

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

Programowanie gier. Kompendium

Autor: Bruno Miguel Teixeira de Sousa
Tłumaczenie: Bochenek Adam, Dobrzański
Jarosław, Dzieniszewski Sławomir
ISBN: 83-7361-119-3
Tytuł oryginału: [Game Programming All In One](#)
Format: B5, stron: 776



Książka „Programowanie gier. Kompendium” dostarczy całej potrzebnej wiedzy, byś stał się twórcą pasjonujących gier komputerowych.

Podzielona na rozdziały zgodnie z poziomem zaawansowania, opisuje kolejno wszystkie aspekty dotyczące programowania gier. Mniej doświadczeni czytelnicy poznać mogą zasady posługiwania się językiem C++ i sprawdzić swe umiejętności pisząc dwie gry tekstowe. Ale to tylko rozgrzewka przed skokiem w krainę DirectX i poznaniem podstawowych składników tej najbardziej rozrywkowej z bibliotek, czyli DirectX Graphics, DirectSound i DirectInput. Twoje umiejętności podniesie z pewnością zaprojektowanie własnych, uniwersalnych modułów, z pomocą których tworzone są przykładowe programy.

Kolejna część książki opisuje w przystępny sposób bardziej zaawansowane tematy, takie jak matematyczne podstawy tworzenia gier, modelowanie fizyczne, sztuczna inteligencja. Na zakończenie dowiesz się, jak napisać prawdziwą, dużą i atrakcyjną grę i... jak ją sprzedać.

Książka zawiera:

- Podstawy C++ i środowiska Visual C++
- Zmienne, operatory, funkcje C++
- Projekty wieloplukowe i preprocesor
- Łańcuchy i wskaźniki
- Programowanie obiektowe: klasy, dziedziczenie
- Strumienie danych
- Projektowanie dużych aplikacji
- Projektowanie bibliotek gier
- Wprowadzenie do programowania w Windows i DirectX
- Grafika w DirectX, DirectInput i DirectSound
- Podstawowe algorytmy i struktury danych
- Matematyczne aspekty programowania gier
- Sztuczna inteligencja w grach
- Modelowanie fizyczne – pisanie gier opartych na prawach fizyki
- Publikowanie gier



Spis treści

O Autorze	17
Wstęp	19
Część I Programowanie w języku C++	21
Rozdział 1. Wprowadzenie do programowania w C++	23
Dlaczego akurat C++?.....	23
Praca z edytorem Visual C++	24
Tworzenie obszaru roboczego	24
Tworzenie projektów	25
Tworzenie i dodawanie plików	26
Pierwszy program: „Witajcie, dobrzy ludzie”	27
Struktura programu w C++	30
Język projektowania programów (PDL).....	31
Kod źródłowy i jego kompilacja.....	32
Obiekty i konsolidacja	32
Plik wykonywalny	33
Komentarze	33
Szukanie błędów	34
Ostrzeżenia.....	36
Podsumowanie	37
Pytania i odpowiedzi	38
Ćwiczenia.....	39
Rozdział 2. Zmienne i operatory	41
Zmienne i pamięć.....	41
Typy zmiennych.....	42
Używanie zmiennych w programach.....	44
Deklarowanie zmiennej	44
Używanie zmiennych.....	45
Inicjowanie zmiennych.....	46
Modyfikatory zmiennych.....	47
Stałe (const)	47
Zmienna typu register	48
Nazewnictwo zmiennych	49
Redefinicja nazw typów	49
Co to jest operator?	50
Operator przyporządkowania.....	50
Operatory matematyczne	50

Operatory jednoargumentowe	51
Operatory dwuargumentowe	52
Złożone operatory przyporządkowania	53
Operatory przesunięcia bitowego.....	54
Operatory porównania.....	54
Operator warunkowy.....	55
Operatory logiczne.....	56
Kolejność operatorów	57
Podsumowanie	58
Pytania i odpowiedzi.....	59
Ćwiczenia.....	59
Rozdział 3. Funkcje i przebieg programu	61
Funkcje: czym są i do czego się przydają?	61
Tworzenie i korzystanie z funkcji.....	63
Deklarowanie prototypu	63
Parametry domyślne.....	66
Zakres zmiennej	67
Zmienne lokalne	68
Zmienne globalne	68
Zmienne statyczne	69
Rekurencja	70
O czym warto pamiętać, korzystając z funkcji?	72
Przebieg programu	73
Bloki kodu i pojedyncze instrukcje.....	73
Instrukcje if, else if i else	73
if.....	74
else	75
Pętle while, do ... while i for	76
while	76
do ... while	77
for.....	78
Przerywanie i kontynuacja pętli.....	80
break	80
continue.....	80
Instrukcja switch	82
Liczby losowe	84
Pierwsza gra — „Craps”	87
Cel.....	87
Zasady.....	87
Projekt.....	87
Implementacja.....	88
Podsumowanie	94
Pytania i odpowiedzi.....	94
Ćwiczenia.....	94
Rozdział 4. Projekty wieloplukowe i preprocesor	97
Pliki źródłowe a pliki nagłówek	97
Korzystanie z wielu plików.....	98
Co to jest preprocesor?.....	100
Unikanie podwójnych załączeń	101
Użycie dyrektywy #pragma.....	102
Użycie #ifdef, #define i #endif	102
Makra	104
Inne dyrektywy preprocesora.....	105

Podsumowanie	105
Ćwiczenia.....	105
Rozdział 5. Tablice, wskaźniki i łańcuchy	107
Czym jest tablica?	107
Deklarowanie i używanie tablicy	107
Deklaracja	108
Używanie tablic	108
Inicjowanie tablicy.....	110
Tablice wielowymiarowe.....	111
Co wskazują wskaźniki?	113
Wskaźniki a zmienne	114
Deklaracja i inicjowanie	114
Używanie wskaźników	114
Wskaźniki a tablice	116
Związek wskaźników z tablicami	116
Przesyłanie tablic do funkcji.....	116
Deklaracja i rezerwacja pamięci dla wskaźnika	118
Rezerwacja pamięci	118
Uwalnianie pamięci	119
Operatory wskaźnikowe.....	121
Manipulowanie pamięcią	124
memset.....	124
memset	125
Łańcuchy	125
Łańcuchy a tablice	125
Korzystanie z łańcuchów	125
Operacje na łańcuchach	126
Podsumowanie	141
Pytania i odpowiedzi	141
Ćwiczenia.....	142
Rozdział 6. Klasy.....	143
Czym jest klasa?.....	143
Nowe typy.....	144
Tworzenie klas	144
Projektowanie	144
Definiowanie.....	145
Implementacja.....	146
Korzystanie z klas	146
Składowe prywatne, publiczne i chronione	146
Tryb prywatny (private).....	147
Tryb publiczny (public)	147
Tryb chroniony (protected).....	148
Który tryb jest najlepszy?	148
Konstruktory i destruktory	149
Konstruktor domyślny	149
Konstruktory ogólne	150
Konstruktor kopiujący i wskazania	150
Destruktor	151
Przeładowywanie operatorów	152
Przykład zastosowania klas — klasa String.....	153
Podstawy dziedziczenia i polimorfizmu	158
Dziedziczenie.....	158
Polimorfizm	163

Wyliczenia	166
Unie.....	167
Składowe statyczne	168
Przydatne techniki korzystania z klas	169
Klasa singleton.....	169
Fabryka obiektów	172
Podsumowanie	176
Pytania i odpowiedzi.....	177
Ćwiczenia.....	177
Rozdział 7. „Potwór” — pierwsza prawdziwa gra	179
ConLib	179
Projektowanie	180
Implementacja.....	182
Tworzenie „Potwora”	192
Cel gry	192
Reguły gry	193
Projektowanie gry.....	193
Implementacja.....	197
Podsumowanie	215
Rozdział 8. Strumienie.....	217
Co to jest strumień?.....	217
Strumienie binarne i tekstowe.....	217
Wejście i wyjście	218
istream.....	218
ostream.....	221
Strumienie plików	223
Otwieranie i zamykanie strumieni	223
Tekst	227
Tryb binarny	232
Modyfikacja gry „Potwór” — zapisywanie i odczyt gry.....	236
Podsumowanie	242
Pytania i odpowiedzi.....	242
Ćwiczenia.....	243
Rozdział 9. Architektura oprogramowania	245
Rola projektowania w tworzeniu programów	245
Strategie projektowe	246
Strategia odgórna	246
Strategia oddolna	246
Strategia odgórna kontra oddolna	247
Kilka podstawowych technik	247
Przykład 1. Operator przyporządkowania zamiast równości	247
Przykład 2. Instrukcje kontra bloki.....	248
Przykład 3. Makra kontra funkcje inline	248
Przykład 4. Prywatne kontra publiczne, przypadek pierwszy	249
Przykład 5. Prywatne kontra publiczne, przypadek drugi	250
Moduły i stosowanie wielu plików	251
Tworzenie modułów w C++	251
Dlaczego warto tworzyć moduły?	252
Konwencje nazewnicze.....	252
Nazywanie funkcji	252
Nazwy zmiennych	253
Identyfikacja	253

Gdzie zdrowy rozsądek wygrywa z projektem	254
Metoda projektowania używana w tej książce.....	254
Podsumowanie	256
Pytania i odpowiedzi.....	256
Ćwiczenia.....	257

Część II Programowanie w środowisku Windows..... 259

Rozdział 10. Projektowanie biblioteki gier: Mirus 261

Opis ogólny	261
Komponenty biblioteki Mirus.....	262
Komponent pomocniczy	262
Komponent Windows	263
Komponent graficzny.....	263
mrScreen.....	264
mrRGBAImage.....	264
mrSurface.....	264
mrTexture	265
mrTemplateSet.....	265
mrAnimation.....	265
mrABO	265
Komponent dźwiękowy	266
mrSoundPlayer	266
mrCDPlayer	266
Komponent komunikacji z użytkownikiem	267
mrKeyboard.....	267
mrMouse.....	267
mrJoystick.....	268
Tworzenie komponentu pomocniczego	268
Deklarowanie typów	268
mrTimer	269
Jak utworzyć plik błędów	274
Jak korzystać z biblioteki Mirus	275
Podsumowanie	275
Pytania i odpowiedzi.....	275

Rozdział 11. Wprowadzenie do programowania w Windows..... 277

Historia Windows	277
Wstęp do programowania w Windows	278
Okna.....	278
Wielozadaniowość.....	279
Windows ma swój własny API.....	280
Kolejki komunikatów	280
Visual C++ a aplikacje Windows	280
Tworzenie aplikacji Windows	281
WinMain kontra Main	283
Tworzenie okna	284
Pętla komunikatów	288
Obsługa komunikatów	289
Tworzenie pętli komunikatów działającej w czasie rzeczywistym	291
Tworzenie ogólnej klasy okna	294
Wykorzystanie szkieletu okna z biblioteki Mirus.....	301
Funkcje okien.....	302
SetPosition	302
GetPosition	303

SetSize	304
GetSize.....	304
Show	305
Podsumowanie	305
Pytania i odpowiedzi.....	305
Ćwiczenia.....	306
Rozdział 12. Wprowadzenie do DirectX	307
Co to jest DirectX?.....	307
Krótka historia DirectX.....	307
Dlaczego akurat DirectX?.....	308
Komponenty DirectX.....	309
Jak działa DirectX?	310
Warstwa abstrakcji sprzętowej	310
Model COM.....	311
COM i DirectX	312
Jak używać DirectX w Visual C++?	313
Podsumowanie	314
Pytania i odpowiedzi.....	314
Ćwiczenia.....	314
Rozdział 13. Grafika DirectX.....	315
Używane interfejsy	315
Korzystanie z Direct3D: podstawy	317
Powierzchnie, bufory i łańcuchy zamiany	329
Powierzchnie.....	329
Bufory.....	329
Łańcuchy zamiany	330
Renderowanie powierzchni.....	330
Wierzchołki, wielokąty i tekstury	336
Wierzchołki i wielokąty.....	337
Tekstury	337
Z trzech wymiarów na dwa.....	340
Mapy bitowe Windows	347
Struktura mapy bitowej.....	348
Ładowanie mapy bitowej.....	349
Tryb pełnoekranowy i inne tryby kolorów	350
Teoria kolorów i kluczowanie kolorów	351
Teoria kolorów.....	351
Kluczowanie kolorów	353
Pliki Targa.....	354
Struktura pliku Targa.....	354
Ładowanie pliku Targa	355
Animacja i zestawy szablonów	356
Animacja.....	356
Zestawy szablonów.....	356
Wykrywanie kolizji.....	357
Figury opisujące.....	357
Okręgi opisujące	357
Prostokąty opisujące	358
Manipulacja obrazem dwuwymiarowym.....	360
Przesunięcie	360
Skalowanie.....	361
Obrót	361

Kreślenie figur płaskich	364
Linie	364
Prostokąty i inne wielokąty	367
Okręgi	368
Tworzenie biblioteki Mirus.....	368
mrScreen.....	368
mrRGBAImage.....	384
mrSurface.....	395
mrTexture	402
mrTemplateSet.....	409
mrAnimation.....	413
mrABO	419
Podsumowanie	434
Pytania i odpowiedzi	435
Ćwiczenia.....	435
Rozdział 14. DirectInput	437
Wprowadzenie do DirectInput.....	437
Dane niebuforowane.....	438
Dane buforowane.....	438
Klasa mrInputManager	439
Klasa mrKeyboard	441
Klasa mrMouse	453
Klasa mrJoystick.....	464
Podsumowanie	473
Pytania i odpowiedzi.....	474
Ćwiczenia.....	474
Rozdział 15. DirectSound	475
Teoria dźwięku.....	475
Podstawy interfejsu DirectSound.....	476
Klasa mrSoundPlayer.....	477
Klasa mrSound.....	480
Interfejs MCI.....	490
Klasa mrCDPlayer	491
Podsumowanie	496
Pytania i Odpowiedzi	497
Ćwiczenia.....	497
Część III Właściwe programowanie gry	499
Rozdział 16. Wprowadzenie do projektowania gry.....	501
Na czym polega projektowanie gier?.....	501
Ta straszna dokumentacja projektu	502
Dlaczego technika „mam to wszystko w mojej głowie” nie jest dobra?	503
Dwa rodzaje projektów	504
Miniprojekt	504
Kompletny projekt.....	504
Wypełnianie tworzonej dokumentacji projektu	505
Ogólny opis gry	505
Docelowy system operacyjny oraz wymagania gry	505
Opowieść tworząca fabułę gry.....	506
Ogólne założenia grafiki i dźwięku	506
Menu	506

Przebieg gry	506
Opis postaci i bohaterów niezależnych.....	506
Schemat sztucznej inteligencji gry	506
Podsumowanie	507
Przykładowy projekt gry: „Najeźdźcy z kosmosu”	507
Ogólny opis gry	507
Docelowy system operacyjny i wymagania gry	507
Fabuła gry	508
Ogólne założenia grafiki i dźwięku	508
Menu	508
Przebieg gry	509
Opis postaci i bohaterów niezależnych.....	509
Schemat sztucznej inteligencji gry	510
Podsumowanie gry.....	510
Podsumowanie	510
Pytania i Odpowiedzi	511
Ćwiczenia.....	511
Rozdział 17. Algorytmy i struktury danych	513
Znaczenie doboru właściwych struktur danych i algorytmów.....	513
Listy	515
Prosta struktura listy	516
Listy dwustronnie powiązane	524
Listy zapętłone.....	524
Zalety list	525
Wady list.....	526
Drzewa	526
Drzewa uniwersalne.....	527
Konstruowanie drzewa uniwersalnego	530
Trawersowanie drzewa uniwersalnego.....	531
Destruktor drzewa uniwersalnego	533
Zastosowania drzew uniwersalnych	533
Drzewa przeszukiwania binarnego	534
Krótki wykład na temat drzew binarnych.....	534
Czym jest drzewo przeszukiwania binarnego?.....	535
Przeszukiwanie drzewa przeszukiwania binarnego.....	536
Wstawianie elementów do drzewa przeszukiwania binarnego	538
Usuwanie wartości z drzewa przeszukiwania binarnego.....	539
Uwarunkowania związane z wydajnością	546
Zastosowania drzew przeszukiwania binarnego.....	547
Sortowanie danych.....	547
Algorytm bubblesort	548
Algorytm quicksort.....	550
Porównanie sposobów sortowania.....	554
Kompresja danych.....	555
Pakowanie z użyciem kodowania grupowego	555
Kod źródłowy algorytmu kodowania grupowego	556
Podsumowanie	557
Pytania i odpowiedzi.....	558
Ćwiczenia.....	558
Rozdział 18. Matematyczne aspekty programowania gier	559
Trygonometria.....	559
Reprezentacja graficzna oraz wzory	559
Zależności między kątami	562

Wektory.....	563
Dodawanie i odejmowanie wektorów	566
Iloczyn i iloraz skalarny.....	567
Długość wektora	568
Wektor jednostkowy	569
Wektor prostopadły	569
Iloczyn skalarny.....	570
Iloczyn skalarny z wektorem prostopadłym	571
Macierze.....	571
Dodawanie i odejmowanie macierzy	574
Skalarne mnożenie i dzielenie macierzy.....	575
Macierz zerowa i jednostkowa	577
Macierz transponowana	578
Mnożenie macierzy.....	578
Transformacja wektora	580
Prawdopodobieństwo	580
Zbiory	581
Suma zbiorów	581
Iloczyn (część wspólna) zbiorów.....	582
Funkcje.....	582
Rachunek różniczkowy.....	583
Pochodne.....	584
Podsumowanie	584
Pytania i odpowiedzi	585
Ćwiczenia.....	585

Rozdział 19. Sztuczna inteligencja w pigułce 587

Obszary i zastosowania sztucznej inteligencji.....	587
Systemy eksperckie	587
Logika rozmyta	589
Algorytmy genetyczne.....	590
Sieci neuronowe.....	592
Algorytmy deterministyczne.....	593
Ruch losowy	594
Śledzenie.....	595
Wzorce.....	596
Automat skończony.....	598
Logika rozmyta	599
Podstawy logiki rozmytej	599
Macierze rozmyte	601
Metody zapamiętywania	602
Sztuczna inteligencja i gry.....	602
Podsumowanie	603
Pytania i odpowiedzi.....	603
Ćwiczenia.....	604

Rozdział 20. Wstęp do modelowania fizycznego 605

Czym jest fizyka.....	605
Budujemy moduł modelowania fizycznego.....	606
Do czego służy moduł fizyczny.....	606
Założenia modułu	606
mrEntity	606
Podstawowe pojęcia fizyczne	608
Masa.....	608
Czas.....	609

Położenie.....	609
Prędkość.....	610
Przyspieszenie.....	611
Środek ciężkości.....	612
Siła.....	613
Siła liniowa.....	613
Moment obrotowy.....	615
Siła wypadkowa.....	616
Grawitacja.....	617
Prawo powszechnej grawitacji.....	617
Grawitacja na Ziemi i innych planetach.....	618
Symulacja lotu pocisku.....	619
Tarcie.....	621
Pojęcie tarcia.....	621
Rodzaje tarcia.....	621
Obsługa kolizji.....	626
Moment pędu.....	626
Metoda impulsowa.....	627
Symulacja.....	630
Odstęp czasowy rysowania ramek.....	632
Zespoły cząstek.....	637
Projekt zespołu cząstek.....	637
Program demonstracyjny.....	647
Podsumowanie.....	649
Pytania i odpowiedzi.....	650
Ćwiczenia.....	651
Rozdział 21. Przykładowa gra „Breaking Through”	653
Projekt „Breaking Through”.....	653
Ogólne zasady.....	653
Wymagania sprzętowe.....	653
Scenariusz.....	654
Reguły.....	654
Grafika.....	654
Menu.....	655
Przebieg gry.....	656
Składniki aplikacji.....	657
Implementacja gry „Breaking Through”.....	659
btBlock.....	659
btPaddle.....	663
btBall.....	667
btGame.....	673
Okno główne.....	699
Podsumowanie.....	701
Rozdział 22. Publikacja gry	703
Czy Twoja gra warta jest opublikowania?.....	703
Gdzie powinniśmy zapukać?.....	704
Naucz się pukać właściwie.....	704
Kontrakt.....	705
List intencyjny.....	705
Kontrakt.....	706
Kamienie milowe.....	706
Lista błędów.....	707
Dzień wydania.....	707

Gdy nie znajdziesz wydawcy	707
Wywiady	707
Niels Bauer: Niels Bauer Software Design	707
André LaMothe: Xtreme Games LLC	709
Podsumowanie	711
Przydatne adresy	711
Kilka słów na koniec	711

Dodatki..... 713

Dodatek A Zawartość płyty CD-ROM..... 715

Kody źródłowe.....	715
Microsoft DirectX 8.0 SDK	715
Aplikacje	715
Jasc Paint Shop Pro 7	716
Syntrillium Cool Edit 2000.....	716
Caligari TrueSpace 5	716
Gry	716
Gamedrop	716
Smiley	717
Smugglers 2	717

Dodatek B Wykrywanie błędów za pomocą Microsoft Visual C++ 719

Pułapki i praca krokowa.....	719
Pułapki	719
Praca krokowa	720
Zmiana wartości zmiennych podczas działania programu	720
Podglądanie zmiennych	721

Dodatek C System dwójkowy, dziesiętny i szesnastkowy 723

System binarny.....	723
System heksadecymalny	723
System dziesiętny.....	724

Dodatek D Kompendium języka C..... 725

Standardowa biblioteka wejścia-wyjścia	725
Obsługa plików	726
Struktury.....	728
Dynamiczna alokacja pamięci	728

Dodatek E Odpowiedzi do ćwiczeń 731

Rozdział 1.	731
Rozdział 2.	731
Rozdział 3.	732
Rozdział 4.	732
Rozdział 5.	732
Rozdział 6.	733
Rozdział 8.	733
Rozdział 9.	733
Rozdział 11.	734
Rozdział 12.	734
Rozdział 13.	734
Rozdział 14.	735
Rozdział 15.	735
Rozdział 17.	735

Rozdział 18.	735
Rozdział 19.	736
Rozdział 20.	736
Dodatek F Słowa kluczowe C++	737
Dodatek G Przydatne tabele.....	739
Tabela znaków ASCII.....	739
Tablica całek nieoznaczonych	741
Tablica pochodnych.....	741
Moment bezwładności	742
Dodatek H Dodatkowe źródła informacji	743
Tworzenie gier i programowanie	743
Nowości, recenzje, pobieranie plików	744
Biblioteki.....	744
Niezależni twórcy gier	744
Przemysł.....	745
Humor	745
Książki	745
Skorowidz	747

Rozdział 11.

Wprowadzenie do programowania w Windows

System Windows był, jest i będzie. Umiejętność tworzenia i wyświetlania okien oraz znajomość podstaw ich używania jest kluczowa dla każdego programisty DirectX.

W rozdziale tym wyjaśnię podstawowe zagadnienia, związane z tworzeniem i manipulacją oknami oraz omówię kilka często używanych funkcji, związanych z programowaniem w Windows. Na koniec stworzymy schemat budowy okien, przeznaczony do wielokrotnego zastosowania w naszych grach.

Historia Windows

System Windows przebył długą drogę od czasu pojawienia się jego pierwszej wersji. Od Windows 1.0 do Windows XP rozrósł się on od prostego interfejsu użytkownika z oknami i menu do najbardziej złożonego systemu operacyjnego wszech czasów.

Programowanie i praca z pierwszymi „wcieleniami” Windows była trudna. Struktura programistyczna została w całości zmodyfikowana w Windows 3.1, który był wybawieniem dla wszystkich programistów Windows.

W 1995 roku Microsoft wprowadził 32-bitowy system — Windows 95. Były to czasy, kiedy Microsoft podbił cały rynek (a w zasadzie cały świat), tworząc system przyjazny użytkownikowi, przyjazny programiście, o dużych możliwościach i atrakcyjnym wyglądzie. Wówczas system Microsoftu nadawał się do tworzenia większości aplikacji, ale nie było łatwo tworzyć w nim gry. Około roku później Microsoft stworzył Game SDK (później przemianowany na DirectX), aby spróbować zachęcić programistów do tworzenia gier w ich nowym systemie.

Wraz z pojawieniem się Windows 98 (a także o wiele lepszej wersji DirectX) Microsoft stworzył idealne rozwiązanie zarówno dla aplikacji, jak i gier. Prawdziwie 32-bitowy

system gwarantował szybkie i stabilne środowisko dla gier. Wyglądał co prawda tak samo i z pozoru niewiele różnił się od Windows 95, jednak wewnątrz był zupełnie inny od swego poprzednika.

Wraz z Windows 95 i 98 Microsoft stworzył też Windows NT (obecnie w swoim piątym wcieleniu zwany Windows 2000), który był stabilnym systemem dla sieci i aplikacji, ale bardzo mało wydajnym dla gier. Dopiero w Windows NT 5 Microsoft przyłożył się, aby uczynić go przyjaznym graczom.

Windows Millennium Edition (ME) bardzo dobrze współpracuje z grami i zwykłymi aplikacjami. Jest przyjazny użytkownikowi i kompatybilny z większością istniejącego sprzętu. Ostatnio Microsoft stworzył Windows XP, który łączy stabilność Windows 2000 z prostotą obsługi Windows 98.

Ogólnie rzecz biorąc, Windows na początku był prostym interfejsem użytkownika. Wkrótce stał się pełnowartościowym systemem operacyjnym, uważanym za najbardziej złożony system, jaki kiedykolwiek stworzono.

Wstęp do programowania w Windows

Ograniczymy się tu do zachowania kompatybilności z Windows 98 i nowszymi wersjami, głównie z uwagi na ich 32-bitową architekturę — co nie znaczy, że pisany przez nas kod nie będzie działał w Windows 95. Windows 95 zawiera dużo kodu 16-bitowego, który przyczynia się do jego niestabilności i powoduje błędy — w Windows 98 nie ma już tych problemów. Poza tym kod, który działa w Windows 98, powinien działać idealnie także w nowszych wersjach systemu. Microsoft stara się zachować kompatybilność z programami stworzonymi dla poprzednich systemów.

Tworząc gry (lub inne programy) w Windows, należy wziąć pod uwagę parę spraw. Niektórymi nie trzeba się specjalnie przejmować, a innymi wręcz przeciwnie.

Okna

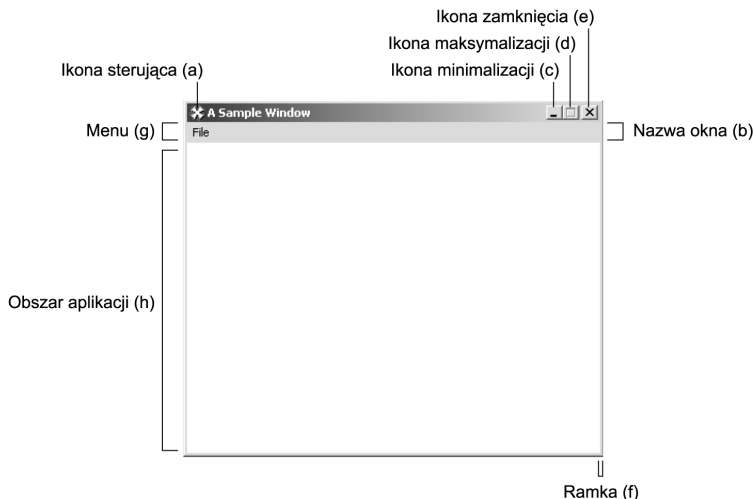
Aplikacje Windows zwykle działają w oknach. Okna składają się z kilku komponentów. Przedstawia je rysunek 11.1.

Rysunek 11.1 przedstawia typowe okno, wykorzystujące najbardziej typowe komponenty. Nie oznacza to, że wszystkie z nich są potrzebne. Oto krótki opis każdego z nich:

- a. po kliknięciu tej ikony wyświetlane jest menu systemowe z podstawowymi funkcjami okna, takimi jak *Przenieś*, *Rozmiar*, *Minimalizuj* itp.;
- b. ten pasek zawiera nazwę okna;
- c. kliknięcie tej ikony minimalizuje okno do paska zadań;
- d. kliknięcie tej ikony maksymalizuje okno do rozmiarów ekranu (o ile to możliwe);
- e. kliknięcie tej ikony zamyka (kończy) aplikację;

Rysunek 11.1.

Typowe okno składa się z kilku komponentów



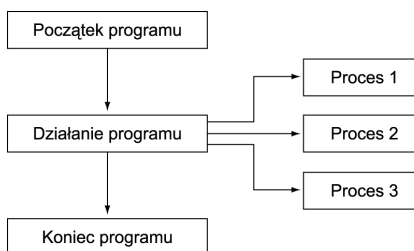
- f. ramka jest używana do zmiany rozmiaru i przedstawienia granicy pomiędzy obszarem okna a innymi aplikacjami na pulpicie;
- g. menu zwykle udostępnia użytkownikowi dodatkowe polecenia w formie grup menu i podmenu;
- h. to obszar, który nas interesuje, czyli obszar aplikacji. Jego zawartość zależy wyłącznie od nas.

Wielozadaniowość

Windows to system wielozadaniowy. Można w nim uruchomić jednocześnie kilka aplikacji. Windows obsługuje dwa rodzaje wielozadaniowości: opartą na procesach i opartą na wątkach. Rysunek 11.2 przedstawia przykład wielozadaniowości.

Rysunek 11.2.

Wielozadaniowość w jednym programie



Jeżeli komputer nie jest systemem wieloprocesorowym, to tak naprawdę nie jest w stanie wykonywać dwóch zadań naraz. Windows emuluje jednak wielozadaniowość, wykonując na zmianę część zadania każdej aplikacji i — dzięki szybkości komputera — tworząc iluzję, że różne zadania są wykonywane w tym samym czasie. Jeżeli na przykład mamy jeden program, który wykonuje 10 obliczeń w każdym cyklu, oraz drugi, który też wykonuje 10 obliczeń w cyklu, Windows jest w stanie wykonać jedno obliczenie w pierwszej aplikacji, potem jedno w drugiej, następnie znowu jedno w pierwszej aplikacji i tak dalej —, aż do ukończenia obliczeń przez obie aplikacje.

Nawet jeśli sami nie musimy zajmować się tym problemem, powinniśmy zapewnić, by nasze gry nie miały wyłącznego dostępu do procesora. Nigdy nie możemy zakładać posiadania 100% mocy obliczeniowej systemu i tego samego nie możemy wymagać również od systemu użytkownika.

Windows ma swój własny API

W odróżnieniu do tworzonych wcześniej aplikacji konsoli nie ma tu bezpośredniej kontroli nad działaniem aplikacji Windows. Posługujesz się interfejsem programowania aplikacji (API), który umożliwia Ci sterowanie widokiem i manipulację oknami.

Będziemy korzystali z Win32 API, czyli 32-bitowej wersji interfejsu programistycznego Windows API. Starszy interfejs, służący do tworzenia aplikacji 16-bitowych, to Win16 API. W nowym API znajdują się setki funkcji, za pomocą których można sterować tworzonymi aplikacjami.

Do końca tego rozdziału i w paru następnych kod będziemy tworzyć wyłącznie za pomocą Win32 API.

Kolejki komunikatów

Inną ogromną różnicą między aplikacjami konsoli a aplikacjami Windows są komunikaty lub kolejki wejściowe. Wszystko, co dzieje się w naszych programach (np. ruch myszy, naciśnięcie klawisza, ładowanie obcych), jest zgłaszane do aplikacji w formie komunikatu.



Kolejka to lista zdarzeń, danych i wszystkich innych elementów, które działają zgodnie z regułą „pierwszy wchodzi, pierwszy wychodzi” (FIFO). Pierwsze dane, jakie wchodzi na listę, opuszczają ją w pierwszej kolejności.

W każdym cyklu poświęconym Twojej aplikacji zobaczysz, czy w kolejce nie ma oczekujących komunikatów. Przejdziesz wówczas do ich obsługi lub je zignorujesz — więcej na ten temat przy okazji programu obsługi komunikatów.

Visual C++ a aplikacje Windows

Przy tworzeniu aplikacji Windows nie będziemy korzystali już z projektu *Win32 Console*. Zastąpimy go projektem *Win32 Application*.

Nauczyliśmy się już tworzyć nowe projekty, ale na wszelki wypadek przypomnę: musisz kliknąć *File, New* i wybrać zakładkę *Projects*. Następnie wybierz *Win32 Application* i podaj nazwę projektu.

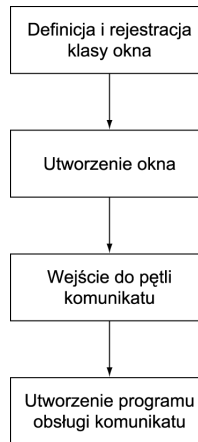
Jeżeli pamiętasz jeszcze aplikacje konsoli, wiesz, że w chwili tworzenia nowego projektu mogłeś zdefiniować kilka wstępnych opcji, pomocnych w jego utworzeniu. Aplikacja Win32 również ma kilka takich opcji. Eksperymentowanie z nimi pozostawiam jednak Tobie.

Teraz dodaj plik źródłowy C++ i możesz zaczynać.

Tworzenie aplikacji Windows

Aplikacje Windows tworzymy w czterech głównych krokach. Ilustruje je rysunek 11.3.

Rysunek 11.3.
*Tworzenie aplikacji
Windows*



Najlepiej zacząć od spojrzenia na pełny kod aplikacji Windows, a następnie szczegółowo przeanalizować istotne jego fragmenty. Oto kod:

Listing 11.1. *Aplikacja Windows*

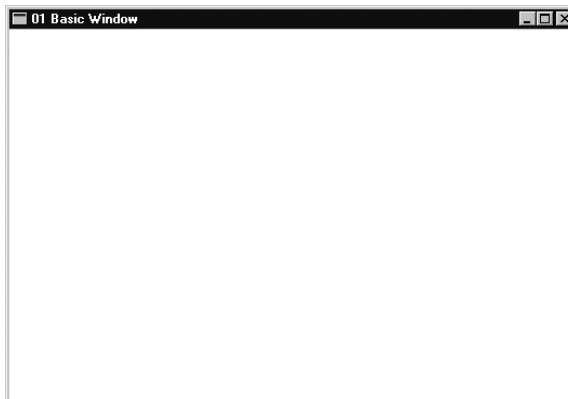
```
1: /* '01 Main.cpp' */
2: #include <windows.h>
3:
4: /* Prototyp programu obsługi komunikatów */
5: LRESULT CALLBACK WndProc (HWND hWnd, UINT iMessage,
6:                           WPARAM wParam, LPARAM lParam);
7:
8: /* "WinMain kontra main" */
9: int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInst,
10:                   LPSTR lpCmdLine, int nShowCmd)
11: {
12:     /* "Klasa Okna" */
13:     WNDCLASS kWndClass;
14:
15:     /* Właściwości wyświetlania */
16:     kWndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
17:     kWndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
18:     kWndClass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
19:
20:     /* Właściwości systemowe */
21:     kWndClass.hInstance = hInstance;
22:     kWndClass.lpfnWndProc = WndProc;
23:     kWndClass.lpszClassName = "01 Basic Window";
24:
25:     /* Właściwości dodatkowe */
```

```
26: kWndClass.lpszMenuName = NULL;
27:
28: kWndClass.cbClsExtra = NULL;
29: kWndClass.cbWndExtra = NULL;
30: kWndClass.style      = NULL;
31:
32: /* Próba rejestracji klasy */
33: if (!RegisterClass (&kWndClass))
34: {
35:     return -1;
36: }
37:
38: /* "Okno" */
39: HWND hWnd;
40: /* Tworzenie okna */
41: hWnd = CreateWindow ("01 Basic Window", "A Blank Window",
42:                     WS_OVERLAPPEDWINDOW | WS_VISIBLE, CW_USEDEFAULT,
43:                     CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
44:                     NULL, NULL, hInstance, NULL);
45: ShowWindow (hWnd, nShowCmd);
46:
47: /* "Pętla komunikatów" */
48: MSG kMessage;
49: /* Wejź do pętli komunikatów i obsługuj komunikaty przesłane do tego
50:    okna */
51: while (GetMessage (&kMessage, hWnd, 0, 0))
52: {
53:     TranslateMessage (&kMessage);
54:     DispatchMessage (&kMessage);
55: }
56:
57: return 0L;
58: }
59:
60: /* "Program obsługi komunikatów" */
61: LRESULT CALLBACK WndProc (HWND hWnd, UINT iMessage,
62:                           WPARAM wParam, LPARAM lParam)
63: {
64:     switch (iMessage)
65:     {
66:     /* Zamknij okno */
67:     case WM_CLOSE:
68:         PostQuitMessage (0);
69:         break;
70:
71:     default:
72:         return DefWindowProc (hWnd, iMessage, wParam, lParam);
73:     }
74:     return 0;
75: }
76:
77:
78:
```

Jeżeli wszystko się powiodło, powinieneś po uruchomieniu zobaczyć okno takie, jak na rysunku 11.4.

Rysunek 11.4.

Okno aplikacji
Windows



Na początku dołączamy plik nagłówka *windows.h* (wiersz 2.). Nagłówek ten zawiera prawie wszystkie funkcje, struktury, stałe itp. interfejsu Win32 API, potrzebne do tworzenia aplikacji Windows. Potem deklarujemy prototyp procedury obsługi komunikatu `WndProc` (wiersze 5. i 6.). Nie przejmuj się na razie tą funkcją — omówimy ją później.

WinMain kontra Main

`WinMain` (wiersz 9.) to odpowiednik funkcji `main` aplikacji konsoli w Windows. Używa ona innej struktury niż `main`. Po pierwsze zwracany typ to `int`. Nie oznacza to, że jesteś zmuszony używać w tej sytuacji `int`, ale powinieneś. Drugą sprawą, jaką zauważyłeś, jest to, że `WinMain` wygląda tak, jakby zwracała dwa typy, co nie jest prawdą. `WINAPI` to konwencja wywołania specyficzna dla aplikacji Windows — taka, jak `static` lub `inline`, z którymi spotkałeś się wcześniej.

Dalej umieszczone są parametry. Pierwszy parametr `HINSTANCE hInstance` to egzemplarz programu. Można go traktować jako identyfikator naszej aplikacji dla systemu operacyjnego. Drugi parametr nie jest używany w 32-bitowych wersjach Windows i ma zawsze wartość `NULL`.



Uchwyt to wskaźnik do wskaźnika, co oznacza, że wskazuje on adres z listy. Uchwyty są potrzebne, ponieważ menedżer pamięci Windows przemieszcza obiekty według swojego uznania i nie mamy normalnego dostępu do pamięci bez pomocy z zewnątrz.

Trzeci parametr, `LPSTR lpCmdLine` to łańcuch z argumentami wiersza poleceń. Działa on trochę inaczej niż w przypadku konsoli. Jeżeli spróbujemy uruchomić taki program:

```
Executable.exe Pierwszy Drugi
```

`lpCmdLine` będzie takim łańcuchem:

```
„Pierwszy Drugi”
```

Jeżeli więc chcemy zinterpretować argumenty z wiersza poleceń, robimy to tak, jakbyśmy interpretowali normalny łańcuch. Ostatni parametr wskazuje, jak należy wyświetlić okno. Może on przyjąć dowolną wartość z tych pokazanych w tabeli 11.1 — będziemy z nich później korzystać.

Tabela 11.1. Stany okna przekazywane do *WinMain*

Wartość	Opis
SW_HIDE	Ukrywa okno
SW_MINIMIZE	Minimalizuje okno
SW_RESTORE	Aktywuje i wyświetla okno w pierwotnym rozmiarze, jeżeli wcześniej było zmaksymalizowane lub zminimalizowane
SW_SHOW	Aktywuje i wyświetla okno
SW_SHOWMAXIMIZED	Aktywuje i wyświetla zminimalizowane okno
SW_SHOWMINIMIZED	Aktywuje i wyświetla okno zmaksymalizowane
SW_SHOWMINNOACTIVE	Aktywuje i wyświetla okno zminimalizowane i jako aktywne
SW_SHOWNA	Aktywuje i wyświetla okno jako aktywne
SW_SHOWNOACTIVE	Aktywuje i wyświetla okno
SW_SHOWNORMAL	Aktywuje i wyświetla okno w jego oryginalnych rozmiarach i pozycji, jeżeli było zminimalizowane lub zmaksymalizowane

Tworzenie okna

Tworzenie okna można również podzielić na dwa etapy: definicję klasy okna i utworzenie samego okna.

Klasa okna

Pierwszym krokiem w definiowaniu klasy okna jest następująca deklaracja zmiennej:

```
WNDCLASS kWndClass;
```

Zmienna ta służyć będzie do określenia atrybutów okna. Struktura `WNDCLASS` posiada kilka składników, z których skorzystamy. Zdefiniowana jest tak:

```
typedef struct _WNDCLASS {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HANDLE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS;
```

Oto objaśnienia kolejnych pól struktury.

`style` określa rodzaj klasy okna. Tym razem nie skorzystamy z tego składnika i ustawimy go na `NULL`. Następny jest `lpfnWndProc` — wskaźnik procedury obsługi komunikatów, wywoływanej przez okno. We wcześniej zadeklarowaliśmy prototyp funkcji i tam właśnie będziemy go używać — przypisujemy go więc do tego pola.

Po procedurze obsługi okna umieszczone są dwa pola, `cbClsExtra` i `cbWndExtra`, które służą do określenia ilości dodatkowych bajtów przeznaczonych do zarezerwowania odpowiednio dla struktury okna i struktury klasy okna. Nie będziemy ich używać, więc nadamy im wartość zero.

Następne pole to pole egzemplarza — `hInstance`. Jest to egzemplarz aplikacji, w której stworzymy okno. Skorzystamy w tym celu z parametru `hInstance` z `WinMain`. Następne pole to uchwyt ikony — `hIcon`. Pole to określa ikonę, jaka ma być wyświetlana na pasku tytułowym. Aby załadować ikonę, używamy funkcji API, która zadeklarowana jest w poniższy sposób:

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

Funkcja ta, o ile jej wywołanie się powiedzie, zwraca uchwyt do ikony, którego używamy w polu klasy okna. Jej pierwszy parametr to egzemplarz, z którego chcemy załadować ikonę. Tutaj użyjemy `NULL`, ponieważ w naszej aplikacji nie będziemy używać ikon. Jeżeli użyjemy `NULL` jako egzemplarza, możemy skorzystać z predefiniowanej ikony. Drugi parametr to łańcuch zakończony znakiem pustym, określający nazwę ikony do załadowania. W tym przypadku używamy predefiniowanej ikony `IDI_APPLICATION`. Tabela 11.2 wymienia kilka innych ikon, z których możemy korzystać.

Tabela 11.2. *Predefiniowane ikony*

Wartość	Opis
<code>IDI_APPLICATION</code>	Domyślna ikona aplikacji
<code>IDI_ERROR</code>	Ikona komunikatu o błędzie
<code>IDI_INFORMATION</code>	Ikona informacji
<code>IDI_WARNING</code>	Ikona ostrzeżenia
<code>IDI_QUESTION</code>	Ikona pytania
<code>IDI_WINLOGO</code>	Ikona z logo Windows

Następna jest informacja o kursorze — `hCursor`, będąca uchwytem do kursora, którego chcemy używać z naszym oknem. Funkcji `LoadCursor` używamy podobnie do `LoadIcon`.

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

Pierwszy parametr to ponownie egzemplarz naszego programu lub `NULL` — jeżeli chcemy użyć któregoś z predefiniowanych kursorów. Tak jest właśnie w tym przypadku. Drugi parametr to nazwa kursora lub nazwa predefiniowanej ikony. Użyjemy tu `IDC_ARROW` — standardowej strzałki, jaką widzimy zwykle w Windows. Tabela 11.3 zawiera listę predefiniowanych kursorów, z jakich można korzystać.

Zostały jeszcze tylko trzy parametry. Kolejny to styl tła — `hbrBackGround`. Tutaj określamy rodzaj pędzla, którym pomalowane jest tło naszego okna. Używając `GetStockObject`, korzystamy z predefiniowanego zbioru obiektów lub pędzli. Oto składnia tej funkcji:

```
HGDIOBJ GetStockObject (int fnObject);
```

Funkcja zwraca uchwyt do obiektu i pobiera typ obiektu jako parametr. Tabela 11.4 zawiera pełną listę pędzli, z jakich możemy wybierać.

Tabela 11.3. *Predefiniowane kursory*

Wartość	Opis
IDC_APPSTARTING	Standardowa strzałka z małą klepsydrą
IDC_ARROW	Standardowa strzałka
IDC_CROSS	Krzyżyk
IDC_HELP	Strzałka i znak zapytania
IDC_IBEAM	Belka “I”
IDC_NO	Przekreślone kółko (zakaz)
IDC_SIZEALL	Strzałka w czterech kierunkach
IDC_SIZENESW	Strzałka dwukierunkowa, wskazująca na północny wschód i południowy zachód
IDC_SIZENS	Strzałka dwukierunkowa, wskazująca na północ i południe
IDC_SIZENWSE	Strzałka dwukierunkowa, wskazująca na północny zachód i południowy wschód
IDC_SIZEWE	Strzałka dwukierunkowa, wskazująca na wschód i na zachód
IDC_UPARROW	Strzałka pionowa
IDC_WAIT	Klepsydra

Tabela 11.4. *Predefiniowane pędzle*

Wartość	Opis
BLACK_BRUSH	Czarny pędzel
DKGRAY_BRUSH	Ciemnoszary pędzel
GRAY_BRUSH	Szary pędzel
HOLLOW_BRUSH	Pędzel przezroczysty
WHITE_BRUSH	Biały pędzel

Następne pole to nazwa menu — `lpszMenuName`. W tym oknie nie będziemy korzystali z menu, dlatego ustawimy jego wartość na `NULL`.

Ostatnia, lecz niemniej istotna, jest nazwa klasy — `lpszClassName`. Ta nazwa będzie używana przez system Windows w odniesieniu do klasy. Podczas tworzenia okna musimy nadać mu nazwę — w naszym przykładzie będzie to `01 Basic Window`.

W tym momencie przygotowaliśmy naszą klasę do rejestracji. Co teraz? Rejestrujemy!

W wierszu 33. próbujemy zarejestrować klasę za pomocą funkcji `RegisterClass`, której definicja wygląda tak:

```
ATOM RegisterClass (CONST WNDCLASS *lpWndClass);
```

W przypadku powodzenia funkcja zwraca `ATOM`, który identyfikuje klasę okna, lub zero, jeżeli wywołanie się nie powiedzie. Poza sprawdzeniem efektu wywołania nie będziemy używać zwracanego typu, dlatego nie będziemy się tu nim zajmować. Jedyнным parametrem funkcji jest wskaźnik klasy okna, w tym przypadku `&kwndClass`. Funkcja ta rejestruje naszą klasę do dalszego użytku.

Sprawdzamy tu również, czy poprawnie zarejestrowaliśmy klasę okna; jeżeli nie, następuje wyjście z programu i zwrócenie wartości `-1`.

Na tym kończymy fazę deklaracji i rejestracji w procesie tworzenia okna. Jeżeli przebiegła ona pomyślnie, możemy przejść do konstruowania samego okna.

Tworzenie okna

Dotarliśmy do punktu, w którym zaczynamy tworzenie samego okna. Pierwszym krokiem (może nie do końca krokiem) w tworzeniu okna jest deklaracja uchwytu okna:

```
HWND hWnd;
```

Następnie możemy, jak pokazano w wierszach od 41. do 44., utworzyć okno za pomocą następującego kodu:

```
hWnd = CreateWindow ("01 Basic Window", "A Blank Window",  
                    WS_OVERLAPPEDWINDOW | WS_VISIBLE, CW_USEDEFAULT,  
                    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,  
                    NULL, NULL, hInstance, NULL);
```

`CreateWindow` ma wiele parametrów, przyjrzyjmy się więc definicji tej funkcji i kolejno je przeanalizujemy.

```
HWND CreateWindow (LPCTSTR lpClassName,  
                  LPCTSTR lpWindowName,  
                  WORD dwStyle,  
                  int x,  
                  int y,  
                  int nWidth,  
                  int nHeight,  
                  HWND hWndParent,  
                  HMENU hMenu,  
                  HANDLE hInstance,  
                  LPVOID lpParam);
```

`CreateWindow` zwraca uchwyt do utworzonego okna, o ile wywołanie się powiodło lub — w innym wypadku — `NULL`. Zwrócony uchwyt okna może być używany praktycznie w każdej operacji dokonywanej na oknie.

Omówmy teraz parametry. Pierwszy to nazwa klasy — `lpClassName` — tu zawarte będą właściwości okna. Nazwa ta musi być nazwą klasy zarejestrowanej w naszym programie. Używamy więc `01 Basic Window`, ponieważ tak nazywa się klasa, którą zarejestrowaliśmy.

Drugi parametr to nazwa okna — `lpWindowName`. Jest to tekst, który będzie domyślnie widoczny na pasku tytułowym (w naszym przypadku `A Blank Window`.)

Następny jest styl okna — `dwStyle`. Parametr ten określa sposób wyświetlania okna. Stosujemy tu parameter `WS_OVERLAPPEDWINDOW`, tworzący zwykle okno z wszystkimi typowymi składnikami (poza menu) — spójrz na rysunek 11.1). Używamy tu też `WS_VISIBLE` do wymuszenia widoczności okna po uruchomieniu. Oba style łączymy za pomocą operatora `OR`. W tabeli 11.5 przedstawiono kilka typowych stylów okien.

Tabela 11.5. *Style okien*

Wartość	Opis
WS_CHILDWINDOW	Tworzy okno potomne
WS_HSCROLL	Tworzy okno z paskiem przewijania
WS_OVERLAPPEDWINDOW	Tworzy okno z normalnymi komponentami okna
WS_POPUP	Tworzy okno wyskakujące (<i>pop-up</i>)
WS_VISIBLE	Tworzy okno widoczne od początku
WS_VSCROLL	Tworzy okno z pionowym paskiem przewijania

Większości wartości z tabeli 11.5 oraz innych stylów okien można używać razem, łącząc je operatorem OR.

Kolejne 2 parametry — x i y — to pozycja okna na ekranie. Używamy tu `CW_USEDEFAULT`, zezwalając systemowi na wybranie pozycji.

Kolejne parametry są związane z poprzednimi i opisują szerokość oraz wysokość okna. Są to `nWidth` i `nHeight`. Tutaj również wybór wartości powierzamy systemowi Windows, przekazując `CW_USEDEFAULT`.

Następny w kolejności jest uchwyt okna nadrzędnego — `hWndParent`. Nie będziemy z niego co prawda korzystać, ale podajemy tu wartość `NULL`, co stanowi informację dla systemu Windows, że oknem nadrzędnym dla naszego okna jest pulpit.

Następny parametr to uchwyt menu — `hMenu`. Uchwyt ten działa podobnie do uchwyty klasy okna, jednak omówiony dokładniej będzie dopiero w następnym rozdziale — tym razem ustawimy go na wartość `NULL`.

Kolejnym składnikiem jest egzemplarz aplikacji — `hInstance`. Spotkaliśmy się z nim już wcześniej i — tak jak poprzednio — użyjemy tu parametru `hInstance` z `WinMain`.

Na końcu przesyłamy do komunikatu utworzenia okna własne dane — `WM_NCCREATE`. Parametr ten będzie używany w dalszej części rozdziału, podczas budowy klasy okna ogólnego zastosowania; tam też zostanie omówiony.

Skoro utworzyliśmy już okno, skorzystamy z `ShowWindow`, aby wyświetlić je zgodnie z parametrem `nCmdShow` z `WinMain`. Nie jest to krok konieczny, ale powinno się go wykonać, aby sprawdzić, czy Windows poprawnie manipuluje naszymi oknami.

To wszystko. Utworzyliśmy okno i wyświetliliśmy je na ekranie. Teraz przejdziemy do omówienia pętli komunikatów i programu obsługi, aby ukończyć naszą pierwszą aplikację Windows.

Pętla komunikatów

Skoro mamy już okno, trzeba stworzyć pętlę odbierającą komunikaty. Pętla taka jest częścią prawie każdego programu w Windows (choć istnieją pewne zaawansowane techniki pozwalające na jej pominięcie). Uruchomiona aplikacja stale odbiera komunikaty wy-

syłane przez system Windows. Komunikaty te przesyłane są do kolejki komunikatów naszej aplikacji. Kiedy aplikacja jest gotowa do przetworzenia następnego komunikatu, wywołuje funkcję `GetMessage`, która zapisuje komunikat w strukturze `MSG` oraz dokonuje jego translacji i przetworzenia za pośrednictwem procedury obsługi komunikatów. Jako że chcemy, aby aplikacja działała cały czas i przetwarzała wszystkie komunikaty, zastosujemy do powtarzania wszystkich tych kroków pętlę, z której wyjście następuje z chwilą zamknięcia aplikacji przez użytkownika. Krok ten jest widoczny w wierszach od 47. do 54.

Na początku deklarujemy strukturę `MSG` i tworzymy pętlę komunikatów w następujący sposób:

```
MSG kMessage;
while (GetMessage (&kMessage, hwnd, 0, 0))
```

Powstała pętla, która będzie stale wykonywana — aż do momentu opuszczenia aplikacji.

Funkcja `GetMessage` służy do pobrania komunikatu z kolejki komunikatów aplikacji i zapisania go w strukturze `MSG`. Oto jej prototyp:

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

Funkcja ta zwraca zero, jeżeli użytkownik kończy pracę programu, a dokładniej kiedy aplikacja otrzyma komunikat `WM_QUIT`. Pierwszy parametr tej funkcji to wskaźnik struktury `MSG`. Tu przechowywana będzie informacja o komunikacie.

Drugi parametr to uchwyt do okna, gdzie odbieramy komunikat. Użyjemy tu `hwnd`, ponieważ jest to uchwyt do okna, które utworzyliśmy.

Ostatnie dwa parametry to wartości filtrów, które umożliwiają filtrowanie niektórych komunikatów. Nie będziemy ich używali, dlatego obydwóm nadajemy wartość zero.

W pętli trzeba teraz dokonać translacji wszystkich wirtualnych kodów klawiszy na komunikaty znakowe. Nie jest to aż taki ważny i konieczny krok, ale gwarantuje on całkowitą integrację klawiatury z naszym programem. Wykonuje się go, wywołując `TranslateMessage` z adresem komunikatu jako parametrem.

Potem wystarczy jedynie za pomocą `DispatchMessage` przesłać komunikat do programu obsługującego wiadomości. Czynimy to, wywołując `DispatchMessage` z adresem komunikatu jako parametrem.

Ostatni wiersz w `WinMain` to po prostu zwracana wartość funkcji — zero.

Obsługa komunikatów

Jesteśmy już na finalnym etapie tworzenia naszej pierwszej aplikacji Windows — brakuje nam jedynie programu obsługującego komunikaty. Program obsługi komunikatów

to funkcja, która obsługuje wszystkie komunikaty przesłane do naszego okna. Zdażyliśmy już na początku pliku zdefiniować jej prototyp, a teraz skoncentrujemy się na samej funkcji.



Program obsługi komunikatów bywa też zwany po prostu programem obsługi. W systemie Windows i w niektórych dokumentach jest nazywany procedurą okna. Wszystkie te nazwy oznaczają tę samą rzecz.

Kiedy użytkownik naciska klawisz lub porusza myszką, do naszej aplikacji przesyłany jest komunikat. Wówczas musimy dokonać wyboru, czy obsłużymy go sami czy pozostawimy obsługę systemowi Windows. Zwykle obsługujemy jedynie kilka komunikatów z setek możliwych. W tym programie bierzemy jedynie pod uwagę komunikat `WM_CLOSE`, który jest przesyłany do naszej aplikacji za każdym razem, gdy użytkownik próbuje zamknąć program. Kiedy otrzymujemy taki komunikat, obsługujemy go, wysyłając komunikat zamknięcia aplikacji za pomocą funkcji `PostQuitMessage`.

Wróćmy jednak do kodu! Nasza funkcja obsługi komunikatów `WndProc` ma cztery parametry. Pierwszy z nich — `hWindow` — to uchwyt okna, które odebrało komunikat. Drugi parametr — `iMessage` — to rzeczywisty kod komunikatu, który został przesłany do tego okna. Trzeci i czwarty parametr — `wParam` i `lParam` — są parametrami komunikatu. Wróć do nich przy okazji omawiania innych komunikatów.

Wewnątrz funkcji, za pomocą instrukcji `switch`, następuje sprawdzenie, jaki komunikat został przesłany oraz jego obsługa. W naszym prostym programie interesuje nas tylko komunikat `WM_CLOSE`, więc tylko on zostanie obsłużony. Za pomocą poniższego wiersza kodu „prosimy” Windows o zamknięcie naszej aplikacji:

```
PostQuitMessage (0);
```

Funkcja ta ma tylko jeden parametr — kod wyjścia, który będzie przesłany do komunikatu `WM_QUIT`.

Teraz, kiedy obsłużyliśmy nasze komunikaty, musimy dodać jeszcze przypadek domyślny (`default`) do instrukcji `switch`, aby pozwolić systemowi Windows obsłużyć te komunikaty, których sami nie obsługujemy. W przypadku domyślnym przesyłamy komunikat funkcją `DefWindowProc` z powrotem do systemu Windows w celu jego przetworzenia. Używamy do tego celu takich samych parametrów, jakie akceptuje nasz program obsługi:

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

Jak widać, zwracamy rezultat tej funkcji, aby system Windows „wiedział”, co zdarzyło się, kiedy przejęliśmy komunikat. Nie musimy się przejmować, w jaki sposób to działa — w systemie Windows jest to bowiem wykonywane automatycznie.

Tym sposobem stworzyliśmy pierwszą aplikację Windows. Nie było to zbyt trudne, prawda? Teraz dopiero zacznie się zabawa: stworzymy pętlę komunikatów działającą w czasie rzeczywistym i dokonamy jej hermetyzacji, przekształcając ją w klasę.

Tworzenie pętli komunikatów działającej w czasie rzeczywistym

Okno, które utworzyliśmy, sprawdzi się w normalnych aplikacjach — takich jak Word czy Notatnik — ale może nie sprawdzić się w grach. Tutaj potrzebujemy pętli, która jest w stanie wykonywać nasz kod zawsze, kiedy nie ma akurat żadnych oczekujących komunikatów. To właśnie pętla działająca w czasie rzeczywistym.

Pseudokod objaśniający działanie takiej pętli wygląda następująco:

```
While Gra się toczy
Begin
  If w kolejce okna oczekują komunikaty
  Begin
    If Komunikat wyjścia
    Begin
      Wyjście
    End
    If Normalny komunikat
    Begin
      Obsługa komunikatu
    End
  End
End
If nie ma komunikatów w kolejce
Begin
  Wykonuj kod gry
End
End
```

Jak przekształcić to na kod? Po pierwsze, musimy usunąć starą pętlę komunikatów i przygotować miejsce dla nowej. Zrobione? Dobrze, kontynuujemy. Z powyższego pseudokodu wynika, że pętla będzie wykonywana do chwili wyjścia, więc pierwszym krokiem jest stworzenie nieskończonej pętli, takiej jak:

```
while (1) {
```

Teraz, gdy jesteśmy już w pętli, musimy określić, czy w kolejce okna oczekują jakieś komunikaty. Realizuje to wywołanie `PeekMessage`. Funkcja ta działa podobnie do `GetMessage`, z tym że zwraca `true`, jeżeli znajdzie jakieś oczekujące komunikaty, lub `false`, jeżeli kolejka jest pusta. Oto jej definicja:

```
BOOL PeekMessage (LPMSG lpMsg,
                  HWND hwnd,
                  UINT wMsgFilterMin,
                  UINT wMsgFilterMax,
                  UINT nRemove);
```

Wkrótce wykorzystamy wartość zwracaną przez tę funkcję, najpierw jednak zajmijmy się jej parametrami. Jak widać, pierwsze cztery parametry na liście są takie same, jak w przypadku funkcji `GetMessage`. Ich rola jest zresztą taka sama, jak w `GetMessage`, więc nie będziemy ich teraz omawiali. Nowością jest ostatni parametr — `nRemove`. Określa on, w jaki sposób należy obsłużyć komunikat. Jeżeli chcemy usunąć z kolejki pobierany

komunikat, podajemy argument `PM_REMOVE`, a jeśli chcemy, by komunikat pozostał w kolejce, zastosujemy argument `PM_NOREMOVE`. Ponieważ nie chcemy pozostawiać komunikatu w kolejce, usuniemy go, stosując następujący zapis:

```
if (PeekMessage (&kMessage, hWnd, 0, 0, PM_REMOVE)) {
```

W ten sposób sprawdzamy, czy w kolejce znajdują się jakieś komunikaty. Jeżeli tak jest, komunikat wysłany jako pierwszy jest kopiowany do `kMessage` i usuwany z kolejki.

Następnie sprawdzamy, czy komunikat to `WM_QUIT`. Oznacza on zakończenie programu. W tym celu sprawdzamy składnik `message` z `kMessage`. Spójrzmy, jak zdefiniowany jest `MSG` (który jest typem `kMessage`):

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

Pierwsze cztery składniki mają takie same zastosowanie jak parametry w funkcji obsługującej komunikaty. Przechowują one odpowiednio: uchwyt do okna, do którego trafił komunikat, rzeczywisty kod komunikatu i parametry komunikatu. Piąty składnik — `time` — jest czasem, w którym do aplikacji został przesłany komunikat, a ostatni składnik — `pt` — przechowuje pozycję kursora w chwili wysłania komunikatu. Nie będziemy bezpośrednio korzystać z tych parametrów (poza kodem komunikatu), dlatego możemy je zignorować.

Zatrzymaliśmy się podczas sprawdzania, czy komunikatem był `WM_QUIT`. Wykonujemy to tak:

```
if (WM_QUIT == kMessage.message) {
```

Co w takim razie, jeżeli `message` jest równe `WM_QUIT`? Wtedy opuszczamy pętlę `while` w zwykły sposób — za pomocą instrukcji `break`:

```
break; }
```

A jeżeli otrzymaliśmy komunikat, ale nie jest to `WM_QUIT`? Musimy wówczas przesłać go do programu obsługi komunikatów w zwykły sposób — za pomocą `TranslateMessage` i `DispatchMessage`:

```
else
{
    TranslateMessage (&kMessage);
    DispatchMessage (&kMessage);
}
}
```

W ten sposób zrealizowaliśmy obsługę komunikatów. Teraz musimy jeszcze dodać fragment kodu umożliwiający wykonanie dowolnych zadań w sytuacji, gdy nie ma komunikatów. Jak to zrobimy? Dodamy warunek `else` do `if (PeekMessage (...))`, który będzie wykonywany, jeżeli `PeekMessage` zwróci `false` (nie ma żadnych komunikatów).

```
else
{
    /* zrób coś */
}
```

Tak powstała pętla komunikatów, działająca w czasie rzeczywistym. Nie było to trudne, prawda? Poniższy listing to pełny kod aplikacji działającej w czasie rzeczywistym:

Listing 11.2. *Aplikacja działająca w czasie rzeczywistym*

```
1: /* '02 Main.cpp' */
2: #include <windows.h>
3:
4: /* Prototyp programu obsługi komunikatów */
5: LRESULT CALLBACK WndProc (HWND hWnd, UINT iMessage,
6:                             WPARAM wParam, LPARAM lParam);
7:
8: /* "WinMain kontra main" */
9: int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInst,
10:                    LPSTR lpCmdLine, int nShowCmd)
11: {
12:     /* "Klasa okna" */
13:     WNDCLASS    kWndClass;
14:
15:     /* Właściwości wizualne */
16:     kWndClass.hCursor      = LoadCursor (NULL, IDC_ARROW);
17:     kWndClass.hIcon        = LoadIcon (NULL, IDI_APPLICATION);
18:     kWndClass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
19:
20:     /* Właściwości systemowe */
21:     kWndClass.hInstance    = hInstance;
22:     kWndClass.lpfnWndProc  = WndProc;
23:     kWndClass.lpszClassName = "02 Real time message loop";
24:
25:     /* Właściwości dodatkowe */
26:     kWndClass.lpszMenuName = NULL;
27:
28:     kWndClass.cbClsExtra = NULL;
29:     kWndClass.cbWndExtra = NULL;
30:     kWndClass.style      = NULL;
31:
32:     /* Próba rejestracji klasy */
33:     if (!RegisterClass (&kWndClass))
34:     {
35:         return -1;
36:     }
37:
38:     /* "Okno" */
39:     HWND hWnd;
40:     /* Tworzenie okna */
41:     hWnd = CreateWindow ("02 Real time message loop",
42:                         "02 Real time message loop",
43:                         WS_OVERLAPPEDWINDOW | WS_VISIBLE, CW_USEDEFAULT,
44:                         CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
45:                         NULL, NULL, hInstance, NULL);
46:     ShowWindow (hWnd, nShowCmd);
47:
```

```
48:  /* "Pętla komunikatów w czasie rzeczywistym" */
49:  MSG kMessage;
50:  /* Wejście do pętli czasu rzeczywistego */
51:  while (1)
52:  {
53:      /* Zapytanie, czy nie ma w kolejce oczekujących komunikatów */
54:      if (PeekMessage (&kMessage, hWnd, 0, 0, PM_REMOVE))
55:      {
56:          /* Jeżeli jest to WM_QUIT, opuść pętlę */
57:          if (WM_QUIT == kMessage.message)
58:          {
59:              break;
60:          }
61:          /* Normalne przetwarzanie komunikatów */
62:          else
63:          {
64:              TranslateMessage (&kMessage);
65:              DispatchMessage (&kMessage);
66:          }
67:      }
68:      /* Brak komunikatów, wykonuj nasz kod */
69:      else
70:      {
71:          /* Nic nie rób ... */
72:      }
73: }
74:
75: return 0L;
76: }
77:
78: /* "Program obsługi komunikatów" */
79: LRESULT CALLBACK WndProc (HWND hWnd, UINT iMessage,
80:                          WPARAM wParam, LPARAM lParam)
81: {
82:     switch (iMessage)
83:     {
84:         /* Zamknij okno */
85:         case WM_CLOSE:
86:             PostQuitMessage (0);
87:             break;
88:
89:         default:
90:             return DefWindowProc (hWnd, iMessage, wParam, lParam);
91:     }
92:     return 0;
93: }
```

Tworzenie ogólnej klasy okna

Teraz, gdy wiemy już, jak utworzyć typowe okno i ponieważ kodu tego będziemy używać wielokrotnie w każdej tworzonej aplikacji Windows, powinniśmy stworzyć własną klasę umożliwiającą wielokrotne stosowanie tego kodu, prawda? Święta prawda!

Oto definicja nagłówka klasy:

Listing 11.3. Nagłówek klasy *mrWindow*

```
1: /* 'mrWindows.h' */
2:
3: /* Nagłówki typów bazowych Mirus */
4: #include "mrDatatypes.h"
5: /* Plik nagłówka Windows */
6: #include <windows.h>
7:
8: /* Dołącz plik tylko raz */
9: #pragma once
10:
11: /* Szkielet okna Mirus */
12: class mrWindow
13: {
14: protected:
15:     WNDCLASS m_kWndClass;
16:     HWND     m_hWindow;
17:     MSG      m_kMessage;
18:
19: public:
20:     /* Konstruktor / Destruktor */
21:     mrWindow (void);
22:     ~mrWindow (void);
23:
24:     /* Funkcje manipulacji oknem */
25:     mrError32 Create (HINSTANCE hInstance, LPSTR szTitle,
26:                     mrInt iWidth = CW_USEDEFAULT,
27:                     mrInt iHeight = CW_USEDEFAULT,
28:                     mrUInt32 iStyle = WS_OVERLAPPEDWINDOW | WS_VISIBLE);
29:     static LRESULT CALLBACK WndProc (HWND hWindow, UINT iMessage,
30:                                     WPARAM wParam, LPARAM lParam);
31:     void Run (void);
32:
33:     /* Funkcje własne */
34:     virtual mrBool32 MessageHandler (UINT iMessage, WPARAM wParam,
35:                                     LPARAM lParam);
36:     virtual mrBool32 Frame (void) = 0;
37:
```

Projekt tej klasy jest dość prosty. Mamy tu funkcję tworzącą okno (*Create*) oraz funkcję uruchamiającą pętlę czasu rzeczywistego (*Run*). Poza tym mamy statyczny program obsługi komunikatów (*WndProc*), który skieruje komunikaty do naszego własnego programu obsługi komunikatów (*MessageHandler*). Poza tym istnieje czysto wirtualna funkcja *Frame*, która jest wywoływana w każdym kadrze, w czasie którego nie ma oczekujących komunikatów. Musi ona zostać zaimplementowana w klasie potomnej.



MrWindow to klasa czysto wirtualna. Aby ją wykorzystać, należy utworzyć jej klasę potomną.

Aby uprościć korzystanie z klasy, użyjemy parametrów domyślnych dla *Create*. Są to *CW_USEDEFAULT* dla *iWidth* i *iHeight* oraz *WS_OVERLAPPEDWINDOW | WS_VISIBLE* dla *iFlags* — tworzą one standardowe widoczne okno.


```
48:                                     NULL, NULL, hInstance, (void *) this);
49: SetWindowText (m_hWindow, szTitle);
50:
51: return mrNoError;
52: }
```

Kod w `Create` jest prawie taki sam, jak w poprzednich aplikacjach Windows. Wypełniamy klasę okna wszelkimi potrzebnymi informacjami, rejestrujemy klasę i tworzymy okno. Znalazło się tu tylko parę zmian, które chcę omówić.



Tym razem przechowujemy wszystkie zmienne okna (`kWndClass`, `hWindow` i `kMessage`) jako składniki okna, czyli `m_kWndClass`, `m_hWindow` i `m_kMessage`.

Po pierwsze, nazwą klasy okna będzie zawsze `Mirus Window`. Dzięki temu nasza aplikacja będzie miała tylko jedno okno (tak, jak powinno być).



Aplikacja może mieć być uruchomiona w wielu oknach, ale dla osiągnięcia najlepszej wydajności powinieneś zawsze korzystać z jednego okna używającego `Direct3D` i samemu dokonać jego podziału.

Druga zmiana godna uwagi znajduje się w wierszu 48., gdzie ostatni parametr `(void *) this` przekazywany jest do `CreateWindow` zamiast `NULL`. Czy pamiętasz jeszcze, do czego służył ostatni parametr `CreateWindow`? Wykorzystywaliśmy go do przesyłania własnych danych do komunikatu `WM_NCREATE`. Użyjesz go wkrótce w programie obsługi komunikatów. Na razie zapamiętaj jedynie, że przesłany tu został adres Twojego okna.



`WM_NCREATE` to komunikat przesłany do okna tuż przed powrotem sterowania z `CreateWindow` do Twojego programu. Wysyłany jest na ułamek chwili przed rzeczywistym utworzeniem okna.

Ostatnia modyfikacja polega na tym, że nie używamy już `ShowWindow`, tylko `SetWindowText`. Nie używamy `ShowWindow`, ponieważ wymuszamy, aby okno było widoczne, miało pożądaną rozmiar; nie używamy już także `nShowCmd` z `WimnMain`.

`SetWindowText` to funkcja API, która nadaje nazwę oknu. Z pewnych dziwnych powodów `CreateWindow` ma problemy z ustawieniem nazwy okna, kiedy wykonywane jest to w klasach, nawet jeżeli nazwa ta jest stałą. Problem ten powinien zostać naprawiony po instalacji *Service Pack 2* lub kolejnych poprawek, ale nigdy nic nie wiadomo.



Najnowszy *Service Pack* dla Visual C++ można za darmo pobrać ze strony Microsoftu pod adresem <http://msdn.microsoft.com/visualc>.

Pierwszy argument `SetWindowText` to uchwyt do okna, którego nazwę chcemy zmienić, a drugi to łańcuch zawierający nazwę okna.

Listing 11.6. *Klasa mrWindow cd.*

```

54: /* Normalny program obsługi komunikatów - przekierowuje komunikaty do naszego*/
55: LRESULT CALLBACK mrWindow::WndProc (HWND hWnd, UINT iMessage,
56:                                     WPARAM wParam, LPARAM lParam)
57: {
58:     mrWindow * pkWindow = NULL;
59:     mrBool32   bProcessed = mrFalse;
60:
61:     switch (iMessage)
62:     {
63:         /* Okno jest tworzone - ustawienie informacji własnych */
64:         case WM_NCCREATE:
65:             SetWindowLong (hWnd, GWL_USERDATA,
66:                            (long)((LPCREATESTRUCT(lParam))->lpcCreateParams));
67:             break;
68:         /* Komunikat okna - niech obsłuży go nasz program obsługi */
69:         default:
70:             pkWindow = (mrWindow *) GetWindowLong (hWnd, GWL_USERDATA);
71:             if (NULL != pkWindow)
72:             {
73:                 bProcessed = pkWindow->MessageHandler (iMessage, wParam, lParam);
74:             }
75:             break;
76:     }
77:     /* Komunikat nie przetworzony - niech obsłuży go Windows */
78:     if (mrFalse == bProcessed)
79:     {
80:         return DefWindowProc (hWnd, iMessage, wParam, lParam);
81:     }
82:     return 0;
83: }

```

Tu mamy trochę zawiłości. Mimo że jest to program obsługi komunikatów, obsługujemy tu tylko jeden komunikat. Dlaczego? Otóż program obsługi komunikatów musi być funkcją statyczną, a jak wiesz, funkcja statyczna nie ma dostępu do składników klasy. W tym przypadku Twój program obsługi komunikatów nie ma dostępu do żadnego ze składników okna, co nie jest dobre. Z tego powodu musimy użyć małego triku z klasą okna, aby skierować wszystkie komunikaty do naszego własnego programu obsługi — `MessageHandler`.

Zwróćmy uwagę na komunikat `WM_NCCREATE`. Jest on przesyłany w momencie utworzenia okna i na szczęście jeden z parametrów komunikatu — `lParam` — zawiera nasze własne dane, które przesłaliśmy do `CreateWindow` (jak pamiętamy, był to adres naszej klasy).

Co tak naprawdę dzieje się w wierszach 65. i 66.? Używamy `SetWindowLong` do przechowania adresu naszej klasy okna. Oto definicja `SetWindowLong`:

```

LONG SetWindowLong (HWND hWnd,
                   int nIndex,
                   LONG dwNewLong);

```

Funkcja ta jest wykorzystywana do przechowywania naszych własnych danych związanych z oknem. Pierwszy parametr to okno, w którym chcemy przechować informację

— w tym przypadku `m_hWindow`. Drugi parametr to typ danych, jakie chcemy przechować, w tym przypadku dane użytkownika — `GWL_USERDATA`. Ostatni parametr to dane, jakie chcemy przechować — w tym przypadku adres klasy okna. Ale jak pobrać go z `lParam`?

Pierwszy krok to rzutowanie typu `lParam` na strukturę `LPCREATESTRUCT`. Umożliwia to dostęp do pola struktury, przechowującego adres przesłany w `CreateWindow`. Potem dane te trzeba ponownie rzutować na typ `long`. Wykonujemy to tak:

```
(long)((LPCREATESTRUCT(lParam))->lCreateParams)
```

Teraz już wiesz, że pole `lCreateParams` struktury `LPCREATESTRUCT` przechowuje nasze własne dane, przesłane do `WindowCreate` i można się do niego dostać poprzez rzutowanie typu `lParam`. Jaka korzyść wynika z tego dla Ciebie? Będziesz przechowywał adres swojej klasy okna w uchwycie okna, którego możesz następnie użyć wszędzie tam, gdzie uchwyt okna jest znany, co też za chwilę zrobisz.

Ten program obsługi komunikatów jest wywoływany za każdym razem, gdy do Twojego okna został przesłany komunikat, potrzebujesz więc sposobu na przekierowanie komunikatu do własnego programu obsługi. Jak to zrobić? Wystarczy użyć adresu klasy okna, który będzie wskaźnikiem naszej klasy.

Za każdym razem, gdy wysłany zostanie komunikat i nie będzie to `WM_NCCREATE`, zezwolisz na jego obsługę przez własny program obsługi. Na początku pobierasz adres swojej klasy okna za pomocą `GetWindowLong` — odwrotności `SetWindowLong`. Zwraca on dane, jakie zachowałeś w `SetWindowLong`. Oto definicja `GetWindowLong`:

```
LONG GetWindowLong (HWND hWnd,
                   int nIndex);
```

Funkcja zwraca przechowywane dane i ma dwa parametry: uchwyt okna, w którym przechowywane są dane, oraz typ danych — w tym przypadku `GWL_USERDATA`. Używając `GetWindowLong`, można pobrać adres naszej klasy okna i utworzyć wskaźnik do niej, jak widać w wierszu 70.:

```
pkWindow = (mrWindow *) GetWindowLong (hWindow, GWL_USERDATA);
```



Zarówno `SetWindowLong`, jak i `GetWindowLong` pobierają uchwyt okna. Jest to uchwyt, który przesłałeś do programu obsługi komunikatów, czyli uchwyt utworzonego przez Ciebie okna.

`pkWindow` to wskaźnik do klasy `mrWindow`, zadeklarowanej w wierszu 58.

Teraz można już przekierować komunikat do własnej procedury obsługi — metody `MessageHandler`.

Następuje tu również sprawdzenie, czy `MessageHandler` przetworzył komunikat. Jeżeli nie, powinien zwrócić `mrFalse` — wówczas przekazujemy komunikat do obsłużenia go przez Windows w następujący sposób:

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

Listing 11.7. *Klasa mrWindow — pętla komunikatów*

```
85:  /* Pętla czasu rzeczywistego */
86:  void mrWindow::Run (void)
87:  {
88:  while (1)
89:  {
90:      /* Zapytanie o oczekujące w kolejce komunikaty */
91:      if (PeekMessage (&m_kMessage, m_hWindow, 0, 0, PM_REMOVE))
92:      {
93:          /* Jeżeli komunikat to WM_QUIT - wyjdź z pętli */
94:          if (WM_QUIT == m_kMessage.message)
95:          {
96:              break;
97:          }
98:          /* Przetwarzaj komunikat normalnie */
99:          else
100:         {
101:             TranslateMessage (&m_kMessage);
102:             DispatchMessage (&m_kMessage);
103:         }
104:     }
105:     /* Brak komunikatu, wykonuj funkcje Frame */
106:     else
107:     {
108:         Frame ();
109:     }
110: }
111: }
```

Pętla komunikatów jest tutaj dokładnie taka sama, jak pętla działająca w czasie rzeczywistym (z listingu 11.2). Jedyna różnica polega na tym, że tym razem, kiedy nie ma komunikatów, wywołujemy funkcję (Frame). Sprawdzamy, czy oczekują jakieś komunikaty i jeżeli tak, obsługujemy je. Jeżeli komunikatem jest WM_QUIT, następuje wyjście z pętli. Jeżeli nie ma żadnych komunikatów, wywołujemy funkcję Frame, która zwykle zawiera kod do wykonania w danym kadrze gry.

Listing 11.8. *Klasa mrWindow cd — program obsługi komunikatów*

```
113:  /* Nasz program obsługi */
114:  mrBool32 mrWindow::MessageHandler (UINT iMessage, WPARAM wParam,
115:                                     LPARAM lParam)
116:  {
117:  switch (iMessage)
118:  {
119:      /* Zamknij okno */
120:      case WM_CLOSE:
121:          PostQuitMessage (0);
122:          return mrTrue;
123:      break;
124:      /* Nie obsłużony - niech obsłuży go Windows */
125:      default:
126:          return mrFalse;
127:      break;
128:  }
129: }
```

Wreszcie mamy nasz własny program obsługi komunikatów, który działa w podobny sposób, jak program obsługi z listingów 11.1 i 11.2. Komunikaty obsługujemy normalnie, ale tym razem zwracamy wartość informującą statyczny program obsługi komunikatów o tym, czy obsłużyliśmy dany komunikat czy nie.

Tak oto utworzyliśmy ogólny szkielet okna. Wkrótce przekonasz się, jak prosto się z niego korzystać i jak łatwo rozszerzać go o nowe funkcje.

Wykorzystanie szkieletu okna z biblioteki Mirus

Stworzyliśmy szkielet okna, ale nie wiemy jeszcze, jak go używać. Pomówmy teraz o tym, jak wykorzystać nowo stworzoną klasę.

Spójrz na poniższy kod, ilustrujący, jak szybkie i łatwe jest korzystanie ze szkieletu okna:

Listing 11.9. *Wykorzystanie szkieletu okna*

```
1:  /* '03 Main.h' */
2:
3:  /* Nagłówek szkieletu okna Mirus */
4:  #include "mrWindow.h"
5:
6:  /* Własna klasa potomna */
7:  class CustomWindow : public mrWindow
8:  {
9:  public:
10:   /* Konstruktor / Destraktor */
11:   CustomWindow (void) {};
12:   ~CustomWindow (void) {};
13:
14:   /* Funkcje manipulacji oknem */
15:   mrBool32 Frame (void) {return mrTrue;} ;
16: };
17:
18: /* "WinMain kontra main" */
19: int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInst,
20:                    LPSTR lpCmdLine, int nShowCmd)
21: {
22:   /* Nasze okno */
23:   CustomWindow      kWindow;
24:
25:   /* Utworzenie okna */
26:   kWindow.Create (hInstance, "04 Mirus Example");
27:   /* Wejście do pętli komunikatów */
28:   kWindow.Run ();
29:
30:   return 0;
31: }
```

Jak widać, utworzenie okna i pętli komunikatów zajęło 29 wierszy kodu (a w zasadzie tylko 7).

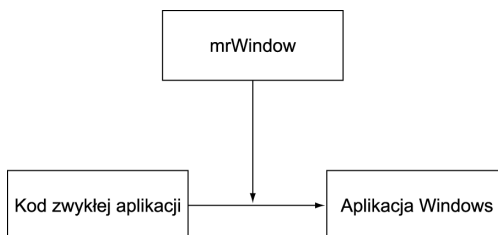
Na początku dołączany jest plik nagłówka biblioteki Mirus *Mirus.h*, który zawiera wszystkie klasy i funkcje biblioteki. Teraz można zdefiniować klasę `CustomWindow`, potomną wobec klasy `mrWindow`, która udostępnia nam wszystkie metody pozwalające na przygotowanie okna do pracy.

Musimy zdefiniować `Frame`, ponieważ — jak pamiętasz — jest to funkcja czysto wirtualna, wywoływana, gdy nie ma oczekujących komunikatów. Jak widać w wierszu 14., sprawiamy, aby na razie zwracana była wartość `mrTrue`.

Po zdefiniowaniu klasy możemy zająć się `WinMain`. Po pierwsze, deklarujemy klasę `CustomWindow` w wierszu 23. Następnie możemy za pomocą `Create` utworzyć okno z tytułem `03 Mirus Example` i poprzez `Run` wejść do pętli komunikatów. Ilustruje to rysunek 11.5.

Rysunek 11.5.

Działanie mrWindow



Funkcje okien

Skoro utworzyliśmy już okno w podstawowej formie, możemy uzupełnić je o pewne cechy funkcjonalne. W części tej opisany został niewielki zbiór funkcji, który w dużym stopniu poprawi funkcjonalność naszego okna.

SetPosition

Pierwsza zaimplementowana metoda służyć będzie do sterowania pozycją okna. Jej prototyp wygląda tak:

```
void SetPosition (mrInt iX, mrInt iY)
```

Przyjmuje ona dwa parametry — pozycję x i y okna. Zdefiniowana jest jako:

```
SetWindowPos (m_hWindow, HWND_TOP, iX, iY, 0, 0, SWP_NOSIZE);
```

`SetWindowPos` umożliwia zmianę rozmiaru i pozycji okna w zależności od parametrów. Jej prototyp zdefiniowany jest następująco:

```
BOOL SetWindowPos (HWND hWnd,
                  HWND hWndInsertAfter,
                  int X,
                  int Y,
```

```
int cx,
int cy,
UINT uFlags);
```

Funkcja ma siedem parametrów. Pierwszy to uchwyt okna, którego rozmiar ustawiamy.

Drugi parametr to uchwyt do sposobu, w jaki okno powinno zostać wstawione do bufora Z — z przodu czy z tyłu. Używamy tu `HWND_TOP`, aby umieścić okno z przodu i tym samym uczynić je widzialnym.



Pamiętaj, że musisz normalnie zdefiniować prototyp w obrębie definicji klasy, a następnie zbudować funkcję, określając zakres jako `void mrWindow::SetPosition (mrInt iX, mrInt iY)`.

Kolejne cztery parametry to odpowiednio współrzędne `x` i `y` oraz szerokość i wysokość okna. Ostatni parametr to opcje `uFlags`, które określają, w jaki sposób `SetWindowPos` ma działać. W naszym przypadku chcemy mieć jedynie możliwość przemieszczania okna, więc nadajemy mu wartość `SWP_NOSIZE`, co wstrzymuje możliwość zmiany rozmiaru okna i powoduje ignorowanie parametrów dotyczących jego rozmiaru.

GetPosition

Funkcja `GetPosition` zwraca strukturę `POINT`, zawierającą współrzędne `x` i `y` okna. Oto jej prototyp:

```
void GetPosition (POINT * pkPosition);
```

`POINT` jest zdefiniowane jako:

```
typedef struct tagPOINT {
    LONG x;
    LONG y;
} POINT;
```

Pola tej struktury to współrzędne `x` i `y`. Treść funkcji wygląda tak:

```
{
    RECT rcWindow;
    POINT pPosition;
    /* Pobierz pozycję okna */
    GetWindowRect (m_hWindow, &rcWindow);

    pPosition.x = rcWindow.left;
    pPosition.y = rcWindow.top;

    memcpy (pkPosition, &pPosition, sizeof (POINT));
}
```



Aby przechowywać pozycję, prześlij do struktury `POINT` wskaźnik. Jeżeli zwrócisz pozycję, nie będziesz mógł przyporządkować jej do żadnej zmiennej, ponieważ `POINT` nie posiada operatora przyporządkowania.

Funkcja pobiera pozycję wszystkich czterech rogów okna i zapisuje je do RECT:

```
GetWindowRect (m_hWindow, &rcWindow)
```

GetWindowRect zwykle pobiera uchwyt do okna jako pierwszy parametr i adres struktury RECT przechowującej wartości. Struktura zdefiniowana jest następująco:

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

Przechowuje ona odpowiednio współrzędne lewego, górnego, prawego i dolnego boku prostokąta.

Pola left i top zawierają odpowiednio współrzędne x i y okna.

SetSize

Funkcja ta pobiera jako parametry szerokość i wysokość okna, a następnie — zgodnie z nimi — zmienia jego rozmiar. Oto jej prototyp:

```
void SetSize (mrInt iWidth, mrInt iHeight);
```

Rozmiar okna ustawiany jest za pomocą SetWindowPos:

```
SetWindowPos(m_hWindow, HWND_TOP, 0, 0, iWidth, iHeight, SWP_NOMOVE);
```

Tym razem jako opcję wybierzemy SWM_NOMOVE, co spowoduje, że SetWindowPos zmieni rozmiar okna i zignoruje parametry dotyczące pozycji.

GetSize

Funkcja ta zwraca strukturę POINT, zawierającą szerokość i wysokość okna. Oto jej prototyp:

```
void GetSize (POINT *pSize);
```

Pole x struktury POINT przechowuje szerokość okna, a pole y jego wysokość.

```
{
    RECT rcWindow;
    POINT pSize;
    /* Pobierz pozycję okna */
    GetWindowRect (m_hWindow, &rcWindow);

    pSize.x = rcWindow.right - rcWindow.left;
    pSize.y = rcWindow.bottom - rcWindow.top;

    memcpy (pSize, &pSize, sizeof (POINT)); }
```

Ponownie używamy GetWindowRect, aby pobrać pozycję i rozmiar okna.

Aby otrzymać szerokość okna, odejmujemy lewą współrzędną od prawej, a wysokość otrzymujemy, odejmując współrzędną górną od dolnej.

Show

Show to funkcja, która ukrywa w sobie funkcję ShowWindow. Pobiera ona stan okna i ustawia je — zgodnie z nim — jako widoczne lub niewidoczne.

```
void Show (mrInt iShow)
```

Oto treść funkcji:

```
{
    /* Zmień stan okna */
    ShowWindow (m_hWindow, iShow);
}
```

Podsumowanie

Nareszcie! Szybki kurs programowania w Windows dał Ci pewnie trochę w kość? Na szczęście kiedy raz zrozumiesz, o co tu chodzi, programowanie w Windows stanie się proste, ponieważ zwykle pracuje się z tym samym lub bardzo podobnym kodem.

Tworząc pierwszą klasę ogólnego zastosowania: mrWindow, stworzyłeś szkielet podstawowego okna, który można wykorzystać w innych projektach za pomocą tylko kilku wierszy kodu.

Pytania i odpowiedzi

- P:** Czym się różni 32-bitowa aplikacja konsoli od 32-bitowej aplikacji Windows?
- O:** Aplikacja konsoli używa interfejsu tekstowego, podobnego do systemów UNIX lub DOS. Aplikacja w oknie ma wszelkie cechy funkcjonalne okien, menu, przycisków i podobnych komponentów systemu Windows.
- P:** Do czego potrzebna jest klasa wirtualna?
- O:** Klasy wirtualnej używamy, aby zmusić użytkownika do stworzenia klasy potomnej i napisania własnej metody Frame. W ten sposób mamy pewność, że wszystko działa dobrze i wszystkie metody są zaimplementowane.
- P:** Dlaczego stosujemy taki skomplikowany kod w programie obsługi komunikatów, zamiast użyć kilku funkcji lub zmiennych globalnych?
- O:** Jak dowiedzieliśmy się już w rozdziale 9., zmienne i funkcje globalne nie powinny być używane, ponieważ nie zapewniają one ukrywania informacji oraz identyfikacji obszarów nazw. Dowolna funkcja może wówczas pomyłkowo zmienić wartość zmiennej globalnej.

Ćwiczenia

1. Jakie jest zadanie funkcji `PostQuitMessage`?
2. Na czym polega działanie pętli obsługi komunikatów czasu rzeczywistego?
3. Czym różni się funkcja `PeekMessage` od `GetMessage`?
4. Dlaczego powinieneś stworzyć zarówno statyczną, jak i niestatyczną funkcję obsługi komunikatów?
5. Dodaj w programie opcję, która spowoduje zamknięcie aplikacji, gdy `Frame` zwróci wartość `mrFalse`.
6. Dodaj do programu kod, który po otrzymaniu komunikatu `WM_CREATE` zmaksymalizuje okno główne.
7. Czy potrafisz wprowadzić w programie taką modyfikację, która zablokuje możliwość zmiany rozmiarów okna przez użytkownika?
8. Teraz proszę zablokować opcje maksymalizacji i minimalizacji okna.
9. Spróbuj dołączyć do programu kod pozwalający użytkownikowi zmieniać kolor tła okna głównego.