helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

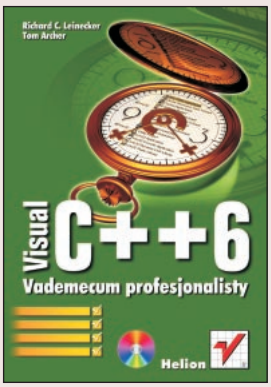
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Visual C++ 6.
Vademecum profesjonalisty



Autorzy: Richard C. Leinecker i Tom Archer
Tłumaczenie: Marcin Pancewicz
ISBN: 83-7197-260-1
Tytuł oryginału: Visual C++ Bible
Format: B5, stron: 1208
oprawa twarda
Zawiera CD-ROM

Jeśli potrafi to Visual C++, Ty także osiągniesz zamierzony efekt. Szybko opanujesz zdolności potrzebne do tworzenia zaawansowanych, komercyjnych aplikacji dla Windows i sieci WWW – poznasz wszystko, co jest do tego potrzebne, od podstaw tworzenia menu i obsługi myszy, przez technologię ODBC, DAO aż do własnych skryptów. Korzystając z książki zawodowych programistów Ricka Leineckera i Toma Archera – „Visual C++ 6. Vademecum profesjoanlisty", szybko zdobędziesz najświeższe informacje potrzebne do utrzymania się w czołówce najlepszych programistów.
Proponujemy Ci doskonałą lekturę, która pomoże w:

- opanowaniu ogólnych i szczegółowych technik programowania dla Windows – od menu do obsługi wyjątków i operacji wejścia-wyjścia;
- zwiększeniu możliwości obsługi baz danych za pomocą ODBC, klasy baz danych MFC, DAO, ADO oraz OLE DB;
- rozszerzeniu aplikacji o dynamicznie łączone biblioteki DLL oraz biblioteki innych programistów;
- lepszym wykorzystaniu MFC do tworzenia kontrolek ActiveX lub ATL do tworzenia kontrolek o bardzo małej objętości kodu;
- pisaniu dynamicznych aplikacji internetowych, aplikacji korzystających z klasy CHtmlView oraz aplikacji DHTML;
- poznaniu zaawansowanych możliwości pakietu Visual Studio, włącznie z makrami VBScript oraz kreatorami AppWizard tworzonymi przez użytkownika.

Na dołączonej do książki płytce CD-ROM znajdziesz rozbudowane narzędzia programistyczne oraz programy demonstracyjne, między innymi:

- Objective Grid, Objective Grid Lite, Objective Toolkit, Objective Chart, Objective Edit, Objective plug-in oraz Objective Diagram stworzone przez Stingray Software
- Ultimate Grid oraz Ultimate Wizard Factory stworzone przez Dundas Software
- BoundsChecker firmy Compuware NuMega Lab

A także pełne programy zademonstrowane w książce!

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

Spis treści

Część I Podstawy Visual C++

Rozdział 1. Zaczynamy 33

Co nowego pojawiło się w Visual C++ 6.0 34
Program HelloWorld1 35
Program HelloWorld2 38
Kontrolka ATL HelloWorld3 40
Podsumowanie 43

Rozdział 2. Korzystanie z zalet zintegrowanego środowiska programisty 45

Korzystanie ze zintegrowanego środowiska 45
Tworzenie pierwszego projektu 47
Dostosowywanie IDE 50
Przeglądanie aktualnych skrótów 50
Dodawanie, modyfikowanie i usuwanie skrótów 51
Korzystanie z pasków narzędzi 53
Wyświetlanie i ukrywanie pasków narzędzi 53
Dostosowywanie pasków narzędzi 54
Podsumowanie 54

Rozdział 3. Debuggowanie: gdy aplikacja nie działa 55

Co każdy debugger posiadać powinien 56
Debugger zintegrowany z Visual Studio 57
Przygotowanie aplikacji do debuggowania 57
Korzystanie z debuggera podczas działania aplikacji 59
Okna debuggowania 60
Korzystanie z punktów wstrzymania i wykonywanie programu krok po kroku 64
Korzystanie z etykietek danych i okna szybkiego podglądu 67
Użycie okien wątków i wyjątków w trakcie debuggowania 67
Proste techniki debuggowania 69
Korzystanie z okien komunikatów w celu przyspieszenia debuggowania 69
Wyjście debuggera 69
Korzystanie z asercji 70
Zrzut obiektów 71
Klasa CMemoryState i wykrywanie wycieków pamięci 71
Komunikaty śledzenia MFC 72
Zdalne debuggowanie 72

Debuggowanie „just-in-time” 74
Debuggowanie typu „edit and continue” 74
Debuggowanie usług Windows NT 75
Podsumowanie 76

Rozdział 4. Poprawianie wydajności aplikacji 77

Optymalizowanie kodu 78
Wprowadzenie do profilowania 80
Podstawy profilowania aplikacji 82
Działanie Profilera 82
Rodzaje profilowania 83
Zaawansowane ustawienia Profilera 84
Włączanie profilowania w Visual C++ 6.0 84
Sterowanie profilowaniem z poziomu Visual Studia 85
Wybór funkcji do profilowania 86
Działanie programu PREP 86
Profilowanie czasu wykonywania funkcji 89
Profilowanie pokrycia funkcji 89
Opcje profilowania linii 90
Profilowanie pokrycia linii 91
Łączenie wyników działania Profilera 91
Eksportowanie danych z Profilera 92
Analizowanie danych Profilera 92
Zawartość globalnych rekordów informacyjnych 93
Lokalne rekordy informacyjne 94
Analizowanie statyki profilowania 95
Przetwarzanie wyników profilera w Excelu 96
Profilowanie bibliotek łączonych dynamicznie 96
Profilowanie komponentów ActiveX lub kontrolek ActiveX 96
Profilowanie kodu typu inline 98
Profilowanie aplikacji wielowątkowych 98
Profilowanie wydajności aplikacji 99
Użycie programu CAP do wyeliminowania powtórzonych wywołań 99
Profilowanie usług Windows NT 101
Podsumowanie 102

Rozdział 5. Inne narzędzia 103

Spy++ 103
Dostosowywanie menu Tools 107
Inne operacje w menu Tools 108
Przeglądanie klas 109
Przeglądanie kodu źródłowego MFC 111
Podsumowanie 112

Część II Podstawy programowania Windows

Rozdział 6. Przegląd programowania w MFC 115

Co to jest MFC? 116
Filozofia MFC 117
Zalety płynące z wykorzystania MFC 117
Właściwości, właściwości, właściwości 119
Nadająca się do rozbudowy architektura 119

Hierarchia klas MFC 120
Usługi plików 120
Okna 120
Grafika 124
Obsługa baz danych 125
Nie używasz MFC? 125
Podsumowanie 126

Rozdział 7. Menu 127

Tworzenie i edycja menu 128
Tworzenie menu 129
Komunikaty menu w MFC 131
Klawisze akceleratorów 132
Polecenia wprowadzane poprzez klawiaturę 133
Definiowanie klawiszy akceleratorów 134
Wybór odpowiednich klawiszy akceleratorów 135
Korzystanie z kilku tabel akceleratorów 138
Dynamicznie zmieniane menu 140
Zakresy poleceń menu 140
Zmiana menu w czasie wykonania 141
Zmiana istniejącego menu 142
Stałe i tymczasowe mapy uchwytów 145
Menu kontekstowe 147
Podsumowanie 149

Rozdział 8. Mysz i klawiatura 151

Komunikaty wejściowe i stan systemu 152
Obsługa myszy 152
Tworzenie funkcji obsługi komunikatów myszy 153
Konwersja pomiędzy współrzędnymi ekranowymi a współrzędnymi okna 155
Tworzenie programu MFC obsługującego zdarzenia myszy 156
Zdarzenia myszy nie związane z obszarem roboczym 161
Zmiana wyglądu wskaźnika myszy 162
Demonstracyjny program zmieniający kształt wskaźnika myszy 163
Przechwytywanie wskaźnika myszy 166
Ograniczanie ruchów myszy 167
Odczyt danych z klawiatury 168
Klawiatura fizyczna 168
Odzwierciedlanie ogniska wprowadzania 172
Podsumowanie 178

Rozdział 9. Grafika 179

Wprowadzenie do interfejsu urządzeń graficznych 180
Rodzaje grafiki 180
Rodzaje urządzeń GDI 182
Kontekst urządzenia 183
Atrybuty rysunkowe kontekstu urządzenia 184
Komunikat WM_PAINT 187
Współrzędne rysowania 188
Generowanie komunikatu WM_PAINT 188
Rysowanie poza komunikatem WM_PAINT 190

Manipulowanie tekstem 191
Obsługa koloru przez GDI 191
Kolor tekstu 193
Wyrównywanie tekstu 194
Justowanie tekstu 195
Czcionki 196
Czym jest czcionka? 196
Wybieranie obiektów w kontekście urządzenia 196
Wybieranie czcionek magazynowych 197
Wybieranie czcionek niemagazynowych 197
Pióra i pędzle 201
Pióra 202
Pędzle 204
Tryby odwzorowania 206
Operacje rastrowe 207
Funkcje rysujące tekst 209
Obliczanie współrzędnych tekstu 210
Regiony obcinania 215
Podsumowanie 216

Rozdział 10. Bitmapy, palety, DIB-y oraz podwójne buforowanie 217

Pamięć bitmapy i pamięć obrazu 218
Tworzenie obiektów CBitmap 220
Ładowanie i ustawianie zawartości bitmapy 222
Rysowanie obiektów CBitmap na ekranie 223
Operacje rastrowe 226
Program BlitDemo 229
Palety i kolor 234
Palety logiczne 237
Zdarzenia dotyczące palety 239
Funkcja SetSystemPaletteUse() 241
Bitmapy niezależne od urządzenia 241
Anatomia pliku DIB 241
Klasa CDib 243
Program demonstracyjny ShowDIB 249
Podwójne buforowanie 252
Podsumowanie 254

Rozdział 11. Obsługa wyjątków MFC 255

Obsługa wyjątków strukturalnych 256
Składnia obsługi wyjątków 256
Zgłaszanie wyjątków 256
Wychwytywanie wyjątków 257
Porównanie technik obsługi błędów 258
Posługiwanie się kodami błędów 258
Obsługa błędów we właściwym kontekście 259
Poprawa czytelności kodu 261
Zgłaszanie wyjątków przez konstruktory 262
Klasa CException 263
Tworzenie i usuwanie obiektów CException 263
Pobieranie od obiektu CException informacji o błędzie 264
Wychwytywanie wyjątków kilku typów 265

Definiowanie klas wyprowadzonych z klasy CException ... 266
Klasa CFileException ... 267
Przykładowy program CFileException ... 269
Definiowanie własnych klas wyprowadzonych z klasy CException ... 272
Zaawansowane techniki obsługi wyjątków ... 275
Decydowanie, która funkcja powinna wychwycić wyjątek ... 276
Wybór kodu przeznaczonego do umieszczenia w bloku try ... 277
Wybór kodu przeznaczonego do umieszczenia w bloku catch ... 278
Zgłaszanie wyjątków w funkcjach wirtualnych ... 279
Podsumowanie ... 282

Rozdział 12. Kontrolki ... 283

Wprowadzenie ... 283
Tworzenie kontrolek ... 284
Używanie klasy CButton ... 287
Używanie klasy CListBox ... 289
Używanie klasy CEdit ... 293
Używanie klasy CStatic ... 297
Używanie klasy CScrollBar ... 300
Używanie klasy CComboBox ... 303
Program Mini edytora ... 306
Zaawansowane programowanie kontrolek ... 307
Dodawanie interfejsu klawiatury ... 307
Modyfikowanie działania kontrolki ... 308
Przyciski z bitmapami ... 309
Zmiana koloru kontrolki ... 311
Podsumowanie ... 313

Rozdział 13. Modalne i niemodalne okna dialogowe ... 315

Korzystanie z edytora dialogów ... 316
Szablony dialogów ... 316
Tworzenie nowego szablonu projektu ... 317
Edycja szablonu dialogu ... 318
Testowanie dialogu ... 323
Klasa CDialog ... 323
Modalne i niemodalne okna dialogowe ... 323
Tworzenie za pomocą ClassWizarda klas wyprowadzonych z klasy CDialog ... 325
Zatwierdzanie i wymiana danych dialogu ... 326
Tworzenie zmiennych DDX ... 326
Funkcje DoDataExchange oraz UpdateData ... 327
Program demonstracyjny AddressBook ... 329
Edycja przykładowego dialogu ... 329
Klasa CContact ... 331
Przesłaniamy funkcję OnInitDialog ... 332
Obsługa komunikatu WM_DESTROY ... 333
Wyświetlanie danych ... 334
Obsługa komunikatów przycisków ... 334
Budowanie i testowanie programu demonstracyjnego ... 335
Stosowanie w dialogach kontrolek klas pochodnych ... 336
Tworzenie klas kontrolek ... 336
Program AddressBook: CAutoCompleteComboBox ... 340

Serializacja 342
Uczynienie klas serializowalnymi 343
Implementowanie funkcji wirtualnej Serialize() 343
Odczyt danych z dysku 344
Zapis danych na dysk 344
Dodanie serializacji do programu AddressBook 344
Podsumowanie 350

Rozdział 14. Karty i arkusze właściwości 351

Klasy CPropertySheet oraz CPropertyPage 352
Tworzenie zasobu karty właściwości 353
Tworzenie klasy CPropertyPage 354
Tworzenie i wyświetlanie modalnego arkusza właściwości 354
Tworzenie i wyświetlanie niemodalnego arkusza właściwości 354
Tworzenie i wyświetlanie arkusza właściwości wewnątrz istniejącego dialogu 356
Program demonstrujący modalny arkusz właściwości 357
Karty i arkusze właściwości: rady i techniki 359
Usuwanie standardowych przycisków 359
Zmiana położenia standardowych przycisków 361
Zmiana tytułów standardowych przycisków 362
Wyłączanie zakładek 362
Ponowne włączanie kart właściwości 366
Dynamiczna zmiana tytułów kart właściwości 367
Zmiana czcionki zakładek 370
Użycie mnemonik z obiektami CPropertyPage 370
Podsumowanie 372

Rozdział 15. Zapis i odczyt danych 373

Klasa CString 374
Tworzenie łańcuchów 374
Dostęp do łańcucha znaków 375
Porównania łańcuchów 376
Wydzielanie podłańcuchów 378
Wyszukiwanie podłańcuchów 378
Program StringDemo 379
Klasa CFile 385
Program FileDemo 389
Klasa CCompressedFile 395
Program CompressDemo 396
Kompresja danych 398
Metoda Huffmana 398
LZSS 399
LZW 399
Funkcje klasy CCompressedFile 400
Działanie programu CompressDemo 402
Klasa CSerial dla komunikacji szeregowej 405
Komunikacja szeregowa 405
Klasa CSerial 406
Klasa CRegistry 408
Wartości Rejestru 408
Predefiniowane klucze Rejestru 408
Powszechnie wykorzystywane klucze Rejestru 409
Działanie klasy CRegistry 410
Korzystanie ze schowka 412
Podsumowanie 415

Rozdział 16. Dźwięk ... 417
Odtwarzanie zarejestrowanych dźwięków ... 417
Rzut okiem na klasę CWave ... 418
Odtwarzanie plików dźwiękowych za pomocą Windows API ... 418
Funkcje klasy CWave ... 421
Program WaveDemo ... 422
Odtwarzanie plików MIDI ... 425
Czym jest MIDI? ... 426
Rzut okiem na klasę CMidi ... 426
Funkcje klasy CMidi ... 426
Program MIDIDemo ... 427
CD Audio ... 431
Rzut okiem na klasę CCDAudio ... 432
Media Control Interface ... 432
Funkcje klasy CCDAudio ... 433
Program CDPlayer ... 434
Podsumowanie ... 440

Rozdział 17. Timery oraz przetwarzanie w jałowym czasie aplikacji ... 441
Timery ... 442
Przygotowanie timera do wysyłania komunikatów WM_TIMER ... 442
Przygotowanie timera do wywoływania funkcji zwrotnej ... 444
Program Clock ... 446
Przetwarzanie w jałowym czasie aplikacji ... 452
Program OnIdleDemo ... 454
Podsumowanie ... 457

Część III Architektura

Rozdział 18. Zarządzanie pamięcią ... 461
Zarządzanie pamięcią systemową ... 461
32-bitowy, stronicowany tryb adresowania Intela ... 464
Przestrzeń adresowa procesu w Windows 98 ... 466
Porządkowanie pamięci systemowej ... 470
Prywatna pamięć procesu ... 472
Alokowanie stron ... 473
Pamięć alokowana przez kompilator ... 478
Prywatne sterty Win32 ... 483
Pamięć wspólna ... 490
Pliki odwzorowane w pamięci ... 492
Dynamiczne alokowanie wspólnych stron ... 498
Statyczne alokowanie wspólnych stron ... 502
Podsumowanie ... 503

Rozdział 19. Dokumenty, widoki i SDI ... 505
Projekt typu dokument-widok ... 506
Klasa CDocument ... 507
Deklarowanie klasy dokumentu w aplikacji ... 508
Zmienne składowe klasy CDocument ... 509
Dokumenty a przetwarzanie komunikatów ... 511
Przesłanianie wirtualnych funkcji dokumentu ... 512
Praca ze złożonymi danymi dokumentu ... 513
Zalety klas CCmdTarget oraz CDocItem ... 516

Znaczenie funkcji InitInstance dla dokumentów 518
W jaki sposób aplikacja zarządza dokumentami i widokami 519
Klasa CSingleDocTemplate 519
Okna ramek 520
Zasoby wzorców dokumentów 520
W jaki sposób zasób łańcucha wpływa na wzorzec dokumentu 521
Podział zasobu łańcucha 522
Przegląd standardowych zasobów widoków 523
Czas życia wzorca dokumentu 524
Zaawansowane posługiwanie się wzorcami 525
Praca z kilkoma wzorcami 525
Niszczenie wzorców dodanych funkcją składową AddDocTemplate() 527
Wykorzystanie klasy CView 528
Deklarowanie klasy widoku 528
Funkcje składowe klasy CView 529
Widoki a komunikaty 531
Warianty klasy okna wyprowadzone z klasy CView 532
Aplikacje oparte na klasie CFormView i aplikacje okien dialogowych 533
Powrót do okien ramki 533
Znaczenie funkcji AfxGetMainWnd() 534
Aplikacja jednodokumentowa HexView 535
Podsumowanie 541

Rozdział 20. Układ aplikacji MDI 543

Ponowny przegląd modelu dokument-widok 544
Ponowny przegląd klasy CDocument 546
Zarządzanie bardziej złożonymi kombinacjami dokumentów, widoków i ramek 547
Praca z kilkoma rodzajami dokumentów 547
Użycie kilku rodzajów widoków dokumentu 548
Klasa CMDIFrameWnd 550
Klasa CMDIChildWnd 551
Klasa CMultiDocTemplate 553
Narzut związany z użyciem klasy CDocument 554
Dalsze właściwości aplikacji MDI 554
Program PaintObj 555
Dzielone okna 562
Różnice pomiędzy dzielonymi oknami 563
Specyfika klasy CSplitterWnd 564
Dodatkowe uwagi na temat dynamicznych okien dzielonych 566
Użycie różnych widoków w dynamicznych panelach 567
Użycie obiektu CRuntimeClass 568
Użycie okien dzielonych z widokami powiązanymi z więcej niż jednym dokumentem 569
Użycie statycznych okien dzielonych 570
Tworzenie statycznego okna dzielonego 570
Wspólne paski przewijania 571
Wyznaczanie rozmiaru aktualnego i idealnego 572
Wydajność okien dzielonych 573
Program Dynsplit 573
Subclassing okien potomnych 578
Subclassing okien z pomocą MFC 579
Alternatywy dla architektury dokument-widok 580
Podsumowanie 581

Rozdział 21. Wydruk i podgląd wydruku ... 583
Obsługa wydruku wbudowana w API Windows ... 584
Informacje o drukarce ... 587
Znaczenie charakterystyki tekstu ... 588
Przykład drukowania z użyciem API Windows ... 589
Drukowanie w MFC ... 591
Podział ról przy drukowaniu ... 591
Sekwencja wydruku w MFC ... 592
Dalszy wgląd w domyślny proces drukowania w MFC ... 595
Protokół wydruku ... 596
Przesłanianie funkcji klasy widoku ... 596
Klasa CPrintInfo ... 598
Strony wydruku a strony dokumentu ... 598
Implementacja podziału na strony (paginacji) ... 599
Dodanie funkcji pomocniczych ... 601
Paginacja w czasie wydruku ... 604
Powrót do drukowanie nagłówków i stopek ... 604
Alokowanie zasobów GDI dla wydruku ... 605
Powiększanie drukowanego obrazu ... 606
Architektura podglądu wydruku ... 606
Proces podglądu wydruku ... 607
Modyfikowanie podglądu wydruku ... 607
Rozszerzanie podglądu wydruku w aplikacji ... 608
Klasa CPrintDialog ... 610
Program HexViewMDI ... 611
Podsumowanie ... 615

Rozdział 22. MFC i elementy zaawansowanego interfejsu użytkownika ... 617
Czasem potrzeba większej kontroli ... 618
Pętla modalna ... 619
Wnętrze funkcji RunModalLoop() ... 619
Parametry pętli modalnej ... 622
Przerywanie pętli modalnej ... 623
Tworzenie modalnej aplikacji ... 623
Tworzenie i zamykanie okna modalnego ... 624
Dodawanie funkcji obsługi ... 624
Prosta modalna aplikacja ... 627
Dalsza rozbudowa modalnego okna ... 628
Rysowanie przezroczystej bitmapy na przezroczystym obszarze roboczym ... 632
Wyświetlanie bitmapy ... 633
Wyznaczenie miejsca wklejenia bitmapy ... 636
Zgodne konteksty urządzeń ... 637
Struktura BITMAP ... 638
Składanie wszystkiego razem ... 640
Ograniczenie obszaru kliknięcia ... 642
Podsumowanie ... 643

Rozdział 23. Paski stanu i paski narzędzi ... 645
Tworzenie i wykorzystywanie pasków stanu ... 646
Tworzenie paska stanu ... 646
Dostosowywanie pasków stanu ... 647
Tworzenie i kontrolowanie pasków narzędzi ... 651
Paski kontrolek MFC ... 651
Tworzenie i inicjowanie paska narzędzi ... 652

Korzystanie z kontrolek ReBar 657
Podsumowanie 658

Rozdział 24. Kontrolka widoku drzewa i kontrolka widoku listy 659

CTreeCtrl 660
Podstawy kontrolki widoku drzewa 661
Klasa CTreeCtrl 662
Program demonstracyjny CTreeCtrlDemo 670
CListCtrl 675
Podstawy kontrolki widoku listy 675
Klasa CListCtrl 676
Podsumowanie 681

Rozdział 25. Wątki 683

Wątki 684
Tworzenie wątków roboczych 685
Funkcja wątku 687
Tworzenie wątków interfejsu użytkownika 687
Wstrzymywanie i wznawianie wątków 688
Usypianie wątków 689
Kończenie działania wątku 689
Kończenie działania wątku z wnętrza innego wątku 690
Wątki, procesy i priorytety 693
Klasy priorytetu procesu 695
Dzielenie obiektów MFC pomiędzy wątkami 695
Używanie funkcji biblioteki czasu wykonania C w aplikacjach wielowątkowych 698
Program ThreadDemo1 699
Podsumowanie 703

Część IV Programowanie baz danych

Rozdział 26. ODBC 707

Terminologia związana z ODBC i bazami danych 708
ODBC – potrzeba standardu 710
Standard ODBC 710
Poziomy zgodności ODBC API 711
Poziomy zgodności ODBC SQL 712
ODBC – implementacja 713
Konfigurowanie ODBC 713
Łączenie się ze źródłem danych 715
Pobieranie informacji o źródle danych 716
Przygotowywanie i wykonywanie poleceń SQL 716
Pobieranie danych 717
Odłączanie się od źródła danych 718
Użycie ODBC do pobierania danych 719
Dodawanie obsługi ODBC do projektu Visual C++ 720
Modyfikowanie okna dialogowego programu 720
Dodawanie kodu inicjalizacji i dostępu do bazy danych 721
Użycie ODBC do dynamicznego odpytywania źródła danych 727
Deklaracja klasy CODBCDynamic 727
Działanie klasy CODBCDynamic 729
Użycie klasy CODBCDynamic 734
Podsumowanie 735

Rozdział 27. Klasy baz danych MFC 737
Klasa CDatabase 738
Klasa CRecordset 741
Konstruowanie rekordsetu 745
Otwieranie rekordsetu 746
Odczyt i zapis danych z użyciem RFX 746
Filtrowanie rekordów 747
Sortowanie rekordów zwróconych przez rekordset 749
Poruszanie się w zestawie wyników 750
Zapisywanie rekordów 751
Usuwanie rekordów 751
Użycie klas baz danych MFC 752
Dodanie obsługi klas baz danych MFC 752
Tworzenie interfejsu użytkownika dla programu demonstracyjnego 752
Dodawanie klasy pomocniczej reprezentującej dane użytkownika 753
Tworzenie klasy rekordsetu dla tabeli UserMaster 754
Modyfikowanie pliku nagłówkowego dialogu 754
Modyfikowanie pliku implementującego dialog 754
Parametryzowane rekordsety i kwerendy 761
Tworzenie interfejsu użytkownika dla programu 761
Tworzenie klasy rekordsetu dla tabeli UserMaster 763
Modyfikowanie klasy CRecordset, tak aby akceptowała parametry 763
Dodawanie możliwości wyszukiwania 763
Budowanie aplikacji 765
Tworzenie parametryzowanych kwerend 765
Demonstracyjna baza danych 766
Tworzenie rekordsetu uprawnień 767
Podsumowanie 770

Rozdział 28. Programowanie baz danych z użyciem DAO 771
Podstawy DAO 772
Historia DAO 772
Hierarchia DAO 774
Interfejsy DAO 778
Użycie klas MFC DAO 779
Klasa CDaoDatabase 780
Klasa CDaoWorkspace 782
Klasa CDaoRecordset 783
Program demonstrujący użycie MFC DAO 793
Tworzenie demonstracyjnego projektu 793
Tworzenie interfejsu użytkownika dla programu demonstracyjnego 794
Dodawanie klasy pomocniczej reprezentującej dane użytkownika 795
Tworzenie klasy rekordsetu dla tabeli UserMaster 795
Modyfikowanie pliku nagłówkowego dialogu 795
Modyfikowanie pliku implementującego dialog 796
Testowanie aplikacji DaoUserMaintenance 803
Podsumowanie 804

Część V Rozszerzanie aplikacji

Rozdział 29. Praca z bibliotekami DLL ... 807
Podstawy bibliotek DLL ... 808
Biblioteki dynamiczne i statyczne ... 808
Ładowanie bibliotek DLL ... 809
Zwykłe biblioteki DLL w Visual C++ ... 810
Wewnętrzne działanie zwykłych bibliotek DLL ... 811
Dynamiczne ładowanie DLL-i ... 813
Przykłady sytuacji wymagających dynamicznego ładowania bibliotek DLL ... 813
Haki Windows ... 815
Globalne obiekty C++ w bibliotekach DLL ... 822
Biblioteki DLL rozszerzeń MFC ... 829
Działanie bibliotek DLL rozszerzeń MFC ... 829
Eksportowanie klas przez biblioteki rozszerzeń MFC ... 830
Makro AFX_EXT_CLASS ... 831
Użycie zagnieżdżonych bibliotek rozszerzeń MFC ... 831
Eksportowanie zasobów ... 832
Pisanie demonstracyjnej biblioteki DLL rozszerzeń MFC zawierającej dokumenty i widoki ... 832
Podsumowanie ... 837

Rozdział 30. Uzupełnianie programów o wyświetlanie obrazków ... 839
Biblioteki tworzone przez osoby trzecie ... 840
Przegląd biblioteki ImageObject ... 843
Format pliku BMP ... 844
Format pliku GIF ... 845
Format pliku JPEG ... 845
Format pliku PCX ... 845
Format pliku TGA ... 845
Format pliku TIF ... 846
Ładowanie obrazków ... 846
Wyświetlanie obrazków ... 847
Program Display ... 850
Skalowanie, obcinanie i zmiana głębokości koloru ... 852
Tworzenie obrazka z kontekstu urządzenia ... 854
Tworzenie obrazu ze schowka ... 854
Zapisywanie obrazków ... 855
Przetwarzanie obrazów ... 857
Program ProcessImage ... 860
Podsumowanie ... 867

Część VI Programowanie modelu COM

Rozdział 31. Wprowadzenie do ActiveX i projektowania kontrolek ActiveX ... 871
Pochodzenie i zastosowanie ActiveX ... 872
Różne technologie ActiveX ... 874
Serwery automatyzacji ... 874
Sterowniki automatyzacji ... 874
Kontrolki ActiveX ... 875
Obiekty COM ... 875
Dokumenty ActiveX ... 876
Kontenery ActiveX ... 877

Do czego możesz wykorzystać ActiveX....877
Jaki rodzaj komponentu ActiveX jest Ci potrzebny?....878
Użycie serwerów i sterowników automatyzacji....879
Użycie kontrolek ActiveX....880
Używanie obiektów COM....880
Tworzenie komponentów ActiveX z użyciem MFC....881
Użycie biblioteki ATL do tworzenia komponentów ActiveX....882
Użycie biblioteki BaseControl do tworzenia komponentów ActiveX....883
Tworzenie własnej biblioteki....884
Podstawy architektury komponentów ActiveX....884
Serwery automatyzacji ActiveX....885
Kontrolki ActiveX....886
Narzędzia wymagane do tworzenia komponentów ActiveX....887
Kompilator MIDL....887
Mktyplib....887
GUIDGEN....888
RegEdit....888
Serwer rejestracji....888
Ole2View....888
Dodawanie narzędzi do środowiska programowego Visual C++....889
Użycie MFC do tworzenia prostej kontrolki ActiveX....889
Tworzenie podstawowego projektu kontrolki....890
Rejestracja kontrolki....893
Tworzenie metod....894
Właściwości....900
Rysowanie kontrolki....912
Rysowanie standardowe....912
Podsumowanie....914

Rozdział 32. Tworzenie serwerów automatyzacji ActiveX z użyciem MFC....917

Tworzenie podstawowego projektu....918
Dodawanie do aplikacji interfejsu automatyzacji....919
Rejestrowanie serwera....923
Tworzenie roboczego kodu serwera....924
Dodawanie metod....927
Dodawanie właściwości do serwera....932
Generowanie wyjątków OLE....933
Serwery o podwójnym interfejsie....939
Generowanie wyjątków OLE podwójnego interfejsu....947
Tworzenie obiektu serwera w programie C++....952
Częste problemy związane z tworzeniem serwerów OLE poprzez C++....954
Tworzenie dzielonych serwerów....954
Serwer w pojedynczym egzemplarzu....958
Podsumowanie....959

Rozdział 33. Active Template Library....961

Przegląd biblioteki ATL....961
Tworzenie kontrolki ATL....963
Dodanie obiektu COM do projektu....964
Modyfikowanie pliku nagłówkowego klasy CDoublePend....965
Implementowanie funkcji interfejsu....967
Używanie kontroli w C++....971

Typy danych....973
Zamiana na typ BSTR i z typu BSTR....973
Porównywanie łańcuchów BSTR....974
Przeglądanie zawartości łańcucha BSTR w debuggerze....975
Osadzanie kontrolek ActiveX na stronach WWW....975
Podsumowanie....976

Część VII Internet i programowanie HTML

Rozdział 34. Programowanie dla Internetu....979
Klasa CInternetSession....980
Klasa CFtpConnection....981
Pobieranie pliku z serwera FTP....984
Wysyłanie pliku do serwera FTP....988
Inne funkcje FTP....988
Klasa CInternetFile....990
Klasa CFtpFileFind....991
Program FTP....991
Gniazda i klasa CSocket....998
Tworzenie gniazd....999
Łączenie się z gniazdem....1000
Nasłuchiwanie połączenia....1001
Odczyt i zapis danych....1002
Program Sockets....1003
Program SendEmail....1009
Pobieranie pliku HTTP....1014
Program AutoDialer....1015
Podsumowanie....1017

Rozdział 35. Klasa CHtmlView....1019
Tworzenie projektu CHtmlView....1020
Zamiana projektów na projekty korzystające z klasy CHtmlView....1021
Nawigowanie w widoku CHtmlView....1022
Program SimpleBrowser....1024
Program MultiBrowser....1026
Bezpośrednie wykorzystanie kontrolki Web Browser....1030
Tworzenie przeglądarki prowadzącej dziennik nawigacji....1031
Tworzenie przeglądarki blokującej witryny WWW....1032
Podsumowanie....1033

Rozdział 36. Dynamic HTML....1035
Podstawy HTML....1036
Arkusze stylów....1038
Globalne arkusze stylów....1041
Składnia arkusza stylu....1042
Prosta przeglądarka plików....1042
Modele obiektów i zdarzeń....1044
Skrypty....1047
Skrypty dla elementów....1048
Zmiana innych elementów niż elementy tekstowe....1051
Uruchamianie skryptu podczas ładowania dokumentu....1052
Tworzenie dialogów w skryptach....1056
Podsumowanie....1058

Część VIII Zaawansowane korzystanie z Visual Studia

Rozdział 37. Skrypty w Visual Studio ... 1061
Makra w Visual Studio ... 1062
Szybkie makra ... 1062
Tworzenie makr VBScriptu ... 1064
Kod makr VBScriptu ... 1066
Edycja makra VBScriptu ... 1066
Tworzenie pustego makra ... 1067
Użycie makr VBScriptu ... 1069
Ładowanie plików makr ... 1069
Uruchamianie makr VBScriptu ... 1070
VBScript ... 1072
Programowanie VBScriptu ... 1073
Używanie zmiennych ... 1073
Używanie stałych ... 1074
Używanie funkcji i procedur ... 1074
Model obiektów Visual Studia ... 1074
Dostęp do obiektów ... 1074
Użycie obiektu Application ... 1075
Przykładowa aplikacja ... 1078
Uruchamianie przykładowego makra ... 1078
Działanie przykładowego makra ... 1080
Podsumowanie ... 1087

Rozdział 38. Tworzenie własnego AppWizarda ... 1089
Działanie AppWizarda ... 1090
Menedżer AppWizarda ... 1090
Słownik oraz klasa CCustomAppWiz ... 1091
Tworzenie własnego AppWizarda ... 1094
Przykład definiowania domyślnych ustawień projektu ... 1096
Tworzenie demonstracyjnego projektu kreatora SDIAutomationWiz ... 1096
Definiowanie klasyCCustomAppWiz ... 1096
Słowniki makr ... 1098
Bardziej zaawansowany własny AppWizard ... 1099
Dodawanie własnego okna dialogowego ... 1099
Tworzenie własnych plików wzorcowych ... 1101
Zmiana pliku newproj.inf ... 1103
Zmiana plików ClassWizarda ... 1105
Zmiana plików wzorcowych AppWizarda ... 1105
Zmiana pliku CONFIRM.INF ... 1109
Użycie Rejestru do utrwalenia makr ... 1110
Modyfikowanie klasy CAboutWizAppWiz tak, aby używała Rejestru ... 1110
Podsumowanie ... 1113

Dodatki

Dodatek A Zawartość płytki CD-ROM ... 1117

Skorowidz ... 1139

Na CD-ROM-ie ... 1140

Rozdział 11.

Obsługa wyjątków MFC

W tym rozdziale:

- Wyjątki i ich obsługiwanie
- Klasa `CException`
- Klasy MFC wyprowadzone z `CException`
- Klasy `CSimpleException` oraz `CUserException`
- Tworzenie własnych klas wyjątków
- Zaawansowane techniki obsługi wyjątków

Wyjątki to błędy powodujące odejście od normalnego działania programu. Wyjątki mogą być generowane zarówno przez oprogramowanie, jak i sprzęt. Do wyjątków sprzętowych należy na przykład dzielenie przez zero czy przepełnienie wyniku operacji arytmetycznej. Wyjątki programowe zgłaszane są na przykład wtedy, gdy aplikacja nie jest w stanie zaalokować pamięci lub gdy nie uda się otwarcie pliku na dysku. Z wyjątków programowych korzysta na przykład klasa `CFile`. Jeśli ta klasa zostanie użyta do otwarcia pliku, lecz nie da się go otworzyć, *zgłaszany* jest wyjątek typu `CFileException`, zaś obiekt klasy `CFileException` zawiera informacje dotyczące przyczyny błędu. Za wykorzystanie informacji zawartych w otrzymanym obiekcie klasy `CFileException` odpowiada funkcja wywołująca metodę `CFile::Open()`. Dzięki temu obsługa błędów za pomocą wyjątków jest uporządkowana i kontrolowana. ponieważ kompilator Visual C++ Microsoftu implementuje model obsługi wyjątków oparty na ISO WG21/ANSI X3J16, termin „obsługa wyjątków” używany w tym rozdziale może być traktowany także jako obsługa wyjątków w C++.

Należy jednak zdać sobie sprawę że obsługa wyjątków nie powinna być stosowana do obsługi „normalnych” czy „oczekiwanych” błędów, z którymi aplikacja powinna łatwo poradzić sobie w inny sposób. Ponieważ istnieją odpowiednie funkcje do sprawdzenia czy plik występuje na dysku, w poprzednim przykładzie programista powinien oczywiście sprawdzić, czy będzie mógł otworzyć plik. To że otwarcie pliku się nie powiodło, oznacza wystąpienie innych warunków, które najprawdopodobniej doprowadziłyby do tego, że funkcja próbująca otworzyć plik i tak nie mogłaby dalej wykonywać swoich zadań. Dopiero w takim przypadku powinna zostać użyta obsługa wyjątków.

Po poznaniu składni obsługi wyjątków samą obsługę wyjątków porównamy z techniką zgłaszania błędów za pomocą zwracanych kodów. Gdy poznasz już przewagę obsługi wyjątków nad innymi technikami obsługi błędów, zapoznamy się z klasą `CException` oraz przykładami obsługi wyjątków zgłaszanych za pomocą klasy `CFileException`. Ponieważ tylko niektóre funkcje mogą zgłaszać kilka różnych wyjątków, większość przykładów poświęcimy definiowaniu i implementacji wyjątków jednego typu. Na końcu rozdziału zajmiemy się bardziej zaawansowanymi zagadnieniami i technikami związanymi z obsługą wyjątków.

Obsługa wyjątków strukturalnych

Wszystkie 32-bitowe wersje Windows wspierają formę obsługi wyjątków na poziomie systemu operacyjnego, czyli obsługi wyjątków strukturalnych (SEH, structure exception handling). SEH zapewnia obsługę wyjątków w prawie każdym języku programowania, nawet jeśli język bezpośrednio nie wspiera obsługi wyjątków. Jednak w dokumentacji MSDN dotyczącej obsługi wyjątków Microsoft twierdzi, że mimo iż w C++ można korzystać z obsługi wyjątków strukturalnych, jednak w programach C++ powinny być stosowane nowe metody obsługi wyjątków.

Więcej informacji na temat obsługi wyjątków strukturalnych znajdziesz w sekcji „Adding Program Functionality" w podręczniku *Visual C++ Programmer's Guide*, dostarczanym wraz z pakietem MSDN.

Składnia obsługi wyjątków

Składnia obsługi wyjątków obejmuje jedynie trzy słowa kluczowe: `try`, `catch` oraz `throw`. W obsłudze wyjątków uczestniczą dwie strony: funkcja serwera *zgłaszająca* (ang. *throw*) wyjątek oraz funkcja klienta *wychwytująca* (ang. *catch*) wyjątek.

Zgłaszanie wyjątków

Gdy funkcja chce zgłosić fakt wystąpienia wyjątku, zgłasza go, używając poniższej składni:

```
throw wyrażenie
```

Sposób użycia instrukcji `throw` jest podobny do użycia instrukcji `return` w C/C++. W C++ *`wyrażenie`* może być dowolnego typu. Na przykład, funkcja może zgłosić wskaźnik do typu `char`:

```
void GetBuffer(char** ppBuffer, unsigned int uiSize)
{
  *ppBuffer = NULL;
  *ppBuffer = new char(uiSize);
```

```
  if( NULL == *ppBuffer){
    throw "Błąd pamięci: Nie udało się zaalokować pamięci.";
  }
}
```

Ponieważ jednak MFC udostępnia klasę podstawową `CException`, w tym rozdziale skoncentrujemy się na zgłaszaniu i wychwytywaniu wyjątków klasy `CException` oraz klas z niej wyprowadzonych. Wkrótce dowiesz się, jak korzystać z tej klasy oraz jak samemu tworzyć klasy pochodne.

Wychwytywanie wyjątków

Ponieważ funkcja wywoływana może zgłosić wyjątek, funkcja wywołująca musi być w stanie na niego zareagować. Dzieje się to poprzez użycie instrukcji wychwycenia. Aby wychwycić wyjątek, musisz ująć odpowiedni fragment kodu w blok `try` określając, jakie rodzaje wyjątków chcesz wychwycić. Wszystkie instrukcje w bloku `try` będą wykonywane jak zwykle, chyba że w wyniku któregoś z wywołań w bloku zdarzy się wyjątek. Gdy któraś z wywoływanych funkcji zgłosi wyjątek, sterowanie zostaje przekazane do pierwszej linii odpowiedniego bloku `catch`. Nawiązując do poprzedniego przykładu, funkcja wywołująca mogłaby wyglądać następująco:

```
#include <iostream.h>

int main()
{
  char *pBuffer;
  try
  {
    GetBuffer(&pBuffer, 512);
    cout << pBuffer;
  }
  catch(char* szException)
  {
  cout << szException;
  }

  return 0;
}
```

Jak pewnie wiesz, gdy funkcja spróbuje wywołać funkcję `cout` z niewłaściwym wskaźnikiem do typu `char`, aplikacja najprawdopodobniej spowoduje jakiś błąd dostępu do pamięci. Jednak w tym przykładzie, jeśli w funkcji `GetBuffer()` nie powiedzie się zaalokowanie pamięci, funkcja `GetBuffer()` zgłosi wyjątek, który zostanie wyłapany w bloku `catch`. Dzięki temu w wypadku gdyby pamięć nie mogła zostać zaalokowana, funkcja `cout` nie będzie wywoływana.

Co się więc stanie, jeśli wywoływana funkcja zgłosi wyjątek, lecz funkcja wywołująca go nie wychwyci? To zależy od aplikacji. Po zgłoszeniu wyjątku sterowanie jest przekazywane zgodnie z zawartością stosu wywołań aż do powrotu do funkcji, która posiada blok `catch` dla wyjątku o zgłoszonym typie. Jeśli funkcja o odpowiednim bloku `catch` nie zostanie znaleziona, działanie aplikacji jest przerywane. Tak więc jeśli napiszesz

funkcję wywołującą inną funkcję mogącą zgłosić wyjątek, ale Twoja funkcja go nie wychwytuje, musisz zapewnić, że jakaś funkcja na stosie wywołań będzie w stanie obsłużyć wyjątek.

Kolejnym ważnym zagadnieniem związanym z wychwytywaniem wyjątków jest wychwytywanie wyjątków różnych rodzajów. W tym celu dodaj po prostu kolejne bloki `catch` dla rodzajów wyjątków, które chcesz wychwycić. Innym sposobem obsługi różnych wyjątków w C++ jest zdefiniowanie bloku `catch` dla klasy bazowej wszystkich klas wyjątków, które mogą zostać zgłoszone, a następnie użycie informacji czasu wykonania o rodzaju klasy C++ (RTTI, runtime type information) w celu określenia rodzaju zgłoszonego wyjątku. W ten sposób Twoja funkcja będzie musiała zawierać tylko jeden blok `catch`. Ponieważ klasy wyjątków w MFC posiadają specjalne wsparcie dla wyznaczania rodzaju klasy, tematem tym zajmiemy się bardziej szczegółowo w sekcji zatytułowanej „Klasa CException" w dalszej części rozdziału.

Teraz gdy wiesz już, jak łatwo jest stworzyć szkielet obsługi wyjątków, zastanówmy się, dlaczego obsługa wyjątków jest lepsza od bardziej tradycyjnej obsługi błędów, wykorzystującej kody zwracane przez funkcje.

Porównanie technik obsługi błędów

Standardowym podejściem przy obsłudze błędów jest zwracanie funkcji wywołującej kod błędu. Wtedy na funkcji wywołującej spoczywa obowiązek analizy zwróconej wartości i odpowiednie zareagowanie. Zwracana wartość może być prostym typem C/C++ lub wskaźnikiem do klasy C++. W bardziej wymyślnych technikach obsługi błędów zwracana jest prosta wartość wskazująca, że wystąpił błąd oraz istnieje globalna funkcja zwracająca bardziej szczegółowe informacje na temat jego natury. Jednak koncepcja jest wciąż ta sama: funkcja wywołująca w jakiś sposób otrzymuje kod błędu i musi go przeanalizować w celu sprawdzenia, czy podjęta operacja się powiodła. Takie podejście ma kilka istotnych wad. Poniżej przedstawiamy kilka sytuacji, w których obsługa wyjątków daje zdecydowane korzyści w porównaniu z korzystaniem ze zwracanych kodów błędów.

Posługiwanie się kodami błędów

Gdy korzystasz z kodów błędów, funkcja wywoływana zwraca kod błędu, zaś samą obsługą błędu zajmuje się funkcja wywołująca. Ponieważ obsługa błędu odbywa się poza zakresem funkcji wywoływanej, nie ma żadnej pewności, że funkcja wywołująca w ogóle sprawdzi zwracany kod. Dla przykładu załóżmy, że napisałeś klasę o nazwie `CCommaDelimitedFile` zawierającą metody przeznaczone do odczytu i zapisu plików o zawartości rozdzielonej przecinkami. Oprócz innych rzeczy, Twoja funkcja musi udostępniać funkcje przeznaczone do otwarcia pliku oraz do odczytu danych z pliku. Przy tradycyjnej metodzie zgłaszania błędów te funkcje zwracałyby zmienne pewnego typu, które musiałyby być sprawdzane, przez funkcję wywołującą w celu upewniania się że wywołanie funkcji zakończyło się sukcesem. Jeśli użytkownik Twojej klasy wywołałby funkcję `CCommaDelimitedFile:`

:Open(), a następnie bez sprawdzenia czy otwarcie pliku powiodło się wywołałaby funkcję CCommaDelimitedFile::Read(), mogłoby to doprowadzić do nieoczekiwanych rezultatów. Jednak jeśli funkcja otwierania pliku zgłaszałaby wyjątek, funkcja wywołująca byłaby zmuszona do zareagowania na niepowodzenie operacji otwarcia pliku. Dzieje się tak, ponieważ za każdym razem gdy zgłaszany jest wyjątek, sterowanie jest przekazywane poprzedniej funkcji ze stosu wywołań aż do natrafienia na funkcję będącą w stanie obsłużyć wyjątek. Oto przykład w jaki sposób mógłby wyglądać kod wywołujący funkcję otwierania pliku:

```
try
{
  CCommaDelimitedFile file;
  file.Open("C:\\test.csv");

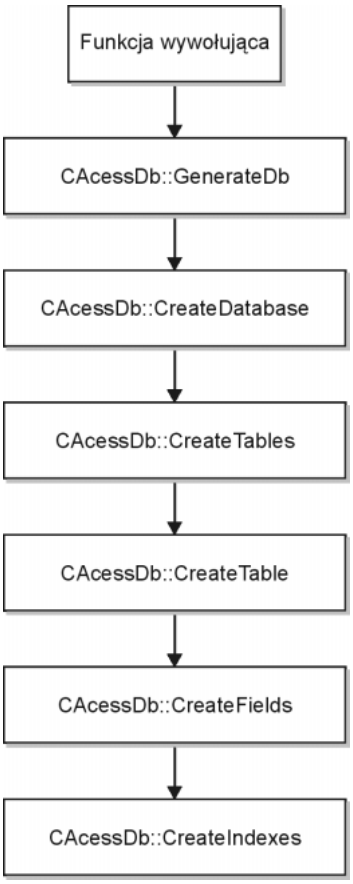
  CString strFirstLine;
  file.Read(strFirstLine);
}
catch(CException *pe)
{
  AfxMessageBox("Wychwycono wyjątek");
}
```

Jak widać, jeśli któraś z funkcji, CCommaDelimitedFile::Open() lub CCommaDelimitedFile::Read(), zgłosi wyjątek, funkcja wywołująca jest zmuszona go obsłużyć. Jeśli funkcja nie wychwyci wyjątku tego typu i nie wychwyci go także żadna inna funkcja ze stosu wywołań aplikacji, działanie aplikacji zostanie przerwane. Zwróć szczególną uwagę na to, że ponieważ obie funkcje, Open() i Read(), zostały umieszczone w tym samym bloku try, w wypadku niepowodzenia przy otwieraniu pliku nie zostanie podjęta próba odczytywania jego zawartości. Dzieje się tak, ponieważ w momencie niepowodzenia funkcji Open() sterowanie zostaje przekazane bezpośrednio do pierwszej instrukcji w bloku catch. Tak więc obsługa wyjątków zapewnia, że niepowodzenie w wywołaniu funkcji nie może zostać zignorowane.

Obsługa błędów we właściwym kontekście

Funkcja wywołująca często nie posiada odpowiednich informacji do obsługi błędu. Właśnie z tego faktu wynika największa zaleta obsługi wyjątków. Jeśli funkcja A wywołuje funkcję B w celu wykonania pewnego bardzo prostego zadania, obsługa ewentualnego błędu może być trywialna. Co jednak zrobić, gdy funkcja B wywołuje funkcję C, która z kolei wywołuje funkcje D oraz E? W jaki sposób funkcja A może zareagować na wystąpienie błędów kilka poziomów wywołań dalej? Spójrzmy na przykład lepiej ilustrujący ten problem.

W tym przykładzie klasa (CAccessDb) jest używania do generowania i manipulowania bazami danych Microsoft Access. Załóżmy, że ta klasa posiada statyczną funkcję o nazwie GenerateDb(). Ponieważ ta funkcja ma być użyta do tworzenia nowych baz danych Accessa, musi wykonać kilka zadań związanych z tym tworzeniem. Musi na przykład fizycznie stworzyć plik bazy danych, wskazane tabele (łącznie z kolumnami i wierszami) oraz zdefiniować potrzebne indeksy i relacje. Funkcja GenerateDb() mogłaby nawet tworzyć domyślnych użytkowników ze standardowymi uprawnieniami. Rysunek 11.1 przedstawia poziomy wywołań funkcji, przez które trzeba przejść w tym przykładzie.



Rys. 11.1. *Bez obsługi wyjątków długie ścieżki kodu powodują ogromny wzrost złożoności kodu związanego z obsługą błędów*

Problem polega na tym, że jeśli błąd wystąpi w funkcji `CreateIndexes()`, która z funkcji powinna go obsłużyć? Oczywiście, w pewnym momencie będzie musiała obsłużyć go funkcja oryginalnie wywołująca funkcję `GenerateDb()`, jednak co może ona zrobić? Nie będzie miała pojęcia, jak zareagować na błąd, który wystąpił kilka poziomów wywołań dalej. Funkcja wywołująca nie będzie znała „kontekstu" wystąpienia błędu. Innymi słowy, jedyną funkcją, która mogłaby logicznie przedstawić informacje na temat błędu, jest ta, której wywołanie się nie powiodło. Korzystając ze standardowej obsługi błędów, każda wywoływana funkcja musiałaby sprawdzać wszystkie kody błędów wszystkich wywoływanych funkcji. Oczywistą niedogodnością jest to, że prowadzi to do analizy ogromnej ilości różnych kodów. Oprócz tego ogromnie utrudniona jest wtedy także konserwacja kodu, gdyż po dodaniu do którejś z funkcji nowego kodu zwracanego błędu konieczna jest aktualizacja analizy kodów we wszystkich innych funkcjach pośrednio odwołujących się do zmodyfikowanej funkcji.

Obsługa wyjątków rozwiązuje wszystkie te problemy, gdyż funkcja wywołująca może wychwycić te wyjątki, które chce obsłużyć. W naszym przykładzie, jeśli z klasy `CException` zostałaby wyprowadzona klasa `CAccessDbException`, mogłaby zostać użyta dla wszystkich rodzajów błędów, które mogłyby wystąpić w którejś z funkcji składowych klasy `CAccessDb`. (Klasę `CException` omówimy za moment). Wtedy, jeśli

błąd wystąpiłby w funkcji `CreateIndexes()`, stworzyłaby ona i zgłosiła wyjątek typu `CAccessDbException`. Funkcja wywołująca mogłaby wychwycić ten wyjątek i sprawdzić jego obiekt w celu określenia, co się nie powiodło. Tak więc zamiast obsługiwać wszystkie możliwe kody błędów zwracanych przez funkcję `GenerateDb()` i wszystkie wywoływane przez nią funkcję, funkcja wywołująca ma pewność, że jeśli nie powiedzie się wywołanie *którejkolwiek* z zagnieżdżonych funkcji, i tak zostaną zwrócone poprawne informacje o istocie błędu. Dodatkową zaletą jest to, że ponieważ informacje o błędzie są zawarte w obiekcie klasy, można łatwo dodawać nowe rodzaje błędów, zaś funkcja wywołująca może pozostać bez zmian.

Poprawa czytelności kodu

Przy użyciu obsługi wyjątków znacznie poprawia się czytelność kodu. Powoduje to także bezpośrednie zmniejszenie kosztów konserwacji kodu. Powodem jest składnia obsługi wyjątków, która jest dużo bardziej przejrzysta od składni obsługi zwracanych kodów błędów. Przy użyciu zwracanych kodów błędów do obsługi wartości zwracanych przez funkcję `CAccessDb::GenerateDb()` z poprzedniego przykładu potrzebny byłby mniej więcej taki kod:

```
void CCallingClass::CallingFunction()
{
  if(CAccessDb::GenerateDb())
  {
    ...
  }
  else
  {
    // W jakiś sposób trzeba wyznaczyć funkcję, która się nie
    // powiodła oraz przyczynę błędu
  }
}

RETURN_CODE CAccessDb::GenerateDb()
{
  if (CreatePhysicalDb())
  {
    if (CreateTables())
    {
      if (CreateIndexes())
      {
        return SUCCESS;
      }
      else
      {
        // obsługa błędu
      }
    }
    else
    {
       // obsługa błędu
    }
  }
```

```
  else
  {
    // obsługa błędu
  }
}
```

Gdy do takiego kodu dodasz kilka instrukcji sprawdzających poprawność wskaźników, ilość kodu do sprawdzania zwracanych wartości błędów kilka razy przekroczy objętość zasadniczego kodu programu. Jeśli w swoim edytorze zamiast spacji wyświetlane są tabulatory ustawione na cztery znaki (domyślne ustawienie Visual Studia), pierwszy znak rzeczywistej linii kodu wystąpi dopiero gdzieś w okolicach dwudziestej kolumny! Choć taki program może być oczywiście całkowicie poprawny, jednak jest bardzo trudny w konserwacji. A jak wiadomo, programy trudne w konserwacji aż proszą się o błędy. Spójrzmy, jak wyglądałby ten sam przykład przy zastosowaniu obsługi wyjątków:

```
void CCallingClass::CallingFunction()
{
  try
  {
    CAccessDb::GenerateDb();
    ...
  }
  catch(CAccessDbException *pe)
  {
    pe->DisplayError();
    pe->Delete();
  }
}

vois CAccessDb::GenerateDb() // zgłasza CAccessDbException
{
  CreatePhysicalDb();
  CreateTables();
  CreateIndexes();
}
```

Zwróć uwagę, jak dużo czystsze i elegantsze jest to drugie rozwiązanie. Powodem jest fakt, że kod wykrywania błędów oraz ich obsługi nie miesza się już z zasadniczym kodem programu. Jak widać, ponieważ dzięki obsłudze wyjątków kod stał się o wiele bardziej przejrzysty, dużo łatwiejsza jest także jego konserwacja.

Zgłaszanie wyjątków przez konstruktory

Ponieważ konstruktory nie mogą zwracać wartości, wyjątki są doskonałym sposobem zgłaszania błędu, który wystąpił podczas konstruowania obiektu. Tak więc, jeśli wiesz, że klasa, której obiekt chcesz stworzyć, zgłasza wyjątki, konstrukcję obiektu musisz umieścić w bloku `try`. Obejmuje to także tworzenie obiektów na stosie. Oto dwa przykłady wychwytywania wyjątków zgłaszanych podczas tworzenia obiektów. Pierwszy z nich ilustruje próbę stworzenia obiektu na stercie, a drugi ilustruje próbę stworzenia obiektu na stosie:

```
// przykład obsługi wyjątku zgłaszanego podczas tworzenia
// obiektu na stercie
try
{
  CTest pTest = new CTest;
}
catch(jakiś_rodzaj_wyjątku)
{
  // obsługa wyjątku
}

// przykład obsługi wyjątku zgłaszanego podczas tworzenia
// obiektu na stosie
try
{
  CTest Test;
}
catch(jakiś_rodzaj_wyjątku)
{
  // obsługa wyjątku
}
```

Klasa CException

Jak już wspominaliśmy, obsługa wyjątków w C++ umożliwia wychwytywanie wyjątków prawie każdego typu. Jednak użycie do tego klasy C++ daje tę niewątpliwą zaletę, że można wychwycić cały obiekt zawierający zarówno informacje opisujące przyczynę błędu, jak i funkcje służące do odczytu tych informacji. MFC dostarcza zestawu klas stworzonych właśnie w tym celu. Klasą bazową wszystkich wyjątków MFC jest klasa `CException`. Nie jest ona wcale skomplikowana i składa się jedynie z konstruktora oraz trzech funkcji: `GetErrorMessage()`, `ReportError()` oraz `Delete()`.

Tworzenie i usuwanie obiektów CException

Choć klasa `CException` posiada funkcje służące do odczytu informacji o danym wyjątku, jednak nie posiada żadnych zmiennych składowych zawierających te informacje. Zdecydowano tak, ponieważ zwykle nigdy nie będziesz tworzył bezpośrednich obiektów tej klasy. Zamiast tego powinieneś użyć klas MFC z niej wyprowadzonych lub własnych klas wyprowadzonych z tej klasy i zaimplementować wirtualne funkcje składowe `GetErrorMessage()` oraz `ReportError()`.

Konstruktor klasy `CException` jako jedyny argument przyjmuje wartość typu BOOL (parametr `m_bAutoDelete`) określającą, czy obiekt wyjątku powinien być usuwany automatycznie. Przekazanie wartości `TRUE` wskazuje, że obiekt wyjątku został utworzony na stercie. To powoduje usunięcie obiektu wyjątku w momencie wywołania jego funkcji składowej `Delete()`. Wartość `FALSE` powinieneś przekazać tylko wtedy, gdy obiekt wyjątku jest tworzony na stosie lub gdy jest obiektem globalnym. Tak więc w większości przypadków będziesz przekazywał wartość `TRUE`.

Jak już wspominaliśmy, do usuwania obiektu wyjątku służy jego funkcja składowa `Delete()`. Wywołanie tej funkcji jest bardzo proste, jednak podczas usuwania obiektu wyjątku powinieneś pamiętać o kilku zaleceniach:

- Nigdy nie usuwaj obiektu wyjątku bezpośrednio, za pomocą operatora `new` języka C++. Zamiast tego używaj funkcji `CException::Delete()`.
- Jeśli wychwycony wyjątek ma zostać ponownie zgłoszony lub w jakiś inny sposób wykorzystany poza zakresem bieżącego bloku `catch`, nie usuwaj tego wyjątku.
- Jeśli wychwycony wyjątek nie będzie używany poza zakresem bieżącego bloku `catch`, powinieneś go usunąć (za pomocą funkcji składowej `Delete()`). Ta zasada obowiązuje także w przypadku, gdy wychwytywany jest wyjątek jednego typu i zgłaszany wyjątek innego typu. Jeśli wychwycony wyjątek nie będzie gdzie indziej wykorzystywany, powinien zostać usunięty.



Jeśli wcześniej nie miałeś do czynienia z mechanizmem `try-catch` języka C++, lecz używałeś makr `TRY` i `CATCH` biblioteki MFC, prawdopodobnie zastanawiasz się, dlaczego powinieneś usuwać wychwycone wyjątki. W rzeczywistości obiekt wyjątku zawsze powinien zostać usunięty, jeśli tylko nie jest zgłaszany dalej. Makra MFC usuwają wyjątek „w tle", w sposób niewidoczny dla programisty. Jeśli jednak masz zamiar korzystać z mechanizmu `try-catch` języka C++, sam jesteś odpowiedzialny za przeprowadzanie porządków.

Pobieranie od obiektu CException informacji o błędzie

Klasa `CException` posiada dwie funkcje składowe służące do odczytu informacji o przyczynie wyjątku: `ReportError()` oraz `GetErrorMessage()`. Funkcja `ReportError()` wyświetla okno komunikatu zawierające opis przyczyny wyjątku. Ponieważ większość aplikacji musi formatować komunikaty wyświetlane użytkownikom, ta funkcja zwykle jest używana jedynie podczas usuwania błędów programu. Oto przykład wykorzystania funkcji `ReportError()`:

```
CStdioFile file;
CFileException fe;

try
{
  file.Open("WiemŻeTenPlikNieIstnieje.txt",
    CFile::typeText | CFile::modeReadWrite, &fe);
  file.Close();
}
catch(CFileException* pe)
{
  pe->ReportError();
}
```

W tym przykładzie użyto klasy `CFileException`. Ta klasa jest wykorzystywana przy zgłaszaniu wyjątków związanych z klasą `CFile` oraz jej klasami pochodnymi. Już za chwilę omówimy ją bardziej szczegółowo. Zwróć uwagę, że nie usuwamy obiektu wyjątku. Dzieje się tak, ponieważ obiekt wyjątku został skonstruowany na stosie, więc zostanie usunięty automatycznie, gdy znajdzie się poza zakresem.

Podczas gdy funkcja `ReportError()` wyświetla opis wyjątku, funkcja `GetErrorMessage()` zwraca tekstowy opis błędu. Podczas wywołania tej funkcji aplikacja musi dostarczyć wskaźnik `LPTSTR` do bufora znaków przeznaczonego na opis, o rozmiarze wystarczającym na pomieszczenie najdłuższego opisu. Po powrocie z funkcji można użyć opisu na przykład do sformatowania komunikatu błędu lub do zapisania go do pliku. Korzystając z poprzedniego przykład, oto jak można sformatować opis wyjątku `CFileException` w celu wyświetlenia go użytkownikowi:

```
CStdioFile file;
CFileException fe;

try
{
  file.Open("WiemŻeTenPlikNieIstnieje.txt",
    CFile::typeText | CFile::modeReadWrite, &fe);
  file.Close();
}
catch(CFileException* pe)
{
  TCHAR szErrorMessage [CHAR_MAX];
  if (pe->GetErrorMessage(szErrorMessage,
    _countof(szErrorMessage)))
  {
    CString strErrorMessage;
    strErrorMessage.Format("Wystąpił błąd pliku: %s",
      szErrorMessage);
    AfxMessageBox(strErrorMessage);
  }
}
```

Wychwytywanie wyjątków kilku typów

Funkcje w bloku `catch` muszą czasem wyłapywać wyjątki różnych typów. Na przykład, załóżmy, że stworzyłeś klasę `CValidatePtr` służącą do sprawdzania poprawności wskaźników. Jeśli ta klasa zgłosi wyjątek, powinien on zostać wychwycony w funkcji wywołującej. Co jednak zrobić, jeśli w tym samym bloku `try` wywoływana jest inna funkcja, zgłaszająca wyjątek innego typu? Można zareagować na dwa sposoby. Pierwsza metoda, przedstawiona na listingu 11.1, polega na zadeklarowaniu kilku bloków `catch`, gdzie każdy z bloków `catch` wychwytuje wyjątek odpowiedniego rodzaju.

Listing 11.1. *Deklarowanie kilku bloków catch*

```
CTestDlg::OnOk()
{
  try
  {
    CValidatePtrException::ValidatePtr(m_pTestPtr,
      RUNTIME_CLASS(CTestPtr));

    m_pTestPtr->DoSomeThing();
  }
  catch(CValidatePtrException* pe)
  {
```

```
    // zrób coś z tym typem wyjątku
    pe->Delete();
  }
  catch(CTestPtrException* pe)
  {
    // zrób coś innego z tym typem wyjątku
    pe->Delete();
  }
}
```

Druga metoda używana do wychwycenia wyjątków różnych typów pochodzących z tego samego bloku `try` polega na użyciu funkcji `IsKindOf()`. Ponieważ klasa `CException` definiuje makra `DECLARE_DYNAMIC` oraz `IMPLEMENT_DYNAMIC`, do wyznaczenia typu zgłoszonego wyjątku może być użyta funkcja `CObject::IsKindOf()`. Listing 11.2 zawiera przykład wykorzystania tej funkcji.

Listing 11.2. *Przykład wykorzystania funkcji IsKindOf()*

```
CTestDlg::OnOk()
{
  try
  {
    CValidatePtrException::ValidatePtr(m_pTestPtr,
      RUNTIME_CLASS(CTestPtr));

    m_pTestPtr->DoSomeThing();
  }
  catch(CException* pe)
  {
    if (pe->IsKindOf(RUNTIME_CLASS(CValidatePtrException)))
    {
      // zrób coś z tym typem wyjątku
      pe->Delete();
    }
    else if (pe->IsKindOf(CTestPtrException))
    {
      // zrób coś innego z tym typem wyjątku
      pe->Delete();
    }
  }
}
```

Definiowanie klas wyprowadzonych z klasy CException

Jak już wyjaśnialiśmy, ponieważ klasa `CException` nie udostępnia metody zdefiniowania przyczyny wyjątku, musisz wyprowadzić z niej klasę pochodną. Oto lista klas wyjątków występujących w MFC, wyprowadzonych z klasy `CException`:

```
CSimpleException
CArchiveException
CFileException
CDaoException
CDBException
COleException
COleDispatchException
CInternetException
```

Choć wyprowadzenie własnej klasy z klasy `CException` nie jest trudne, jednak jedną z największych zalet korzystania z MFC jest korzystanie z już gotowego kodu. A jaka klasa będzie lepsza niż klasa specjalnie wyprowadzona z klasy `CException`? Tak więc, zanim pokażemy, w jaki sposób można wyprowadzić własną klasę wyjątków, spójrzmy, jak MFC robi to samo w przypadku klasy `CFileException`.

Klasa CFileException

Klasa `CFileException` jest używana do obsługi błędów związanych z klasą `CFile` i jej klasami pochodnymi. Wyjątek typu `CFileException` może być zgłoszony w różnych sytuacjach błędów, takich jak odwołanie do nieistniejącego pliku, podanie błędnej ścieżki dostępu czy odwołanie się do już używanego pliku.

Z każdym błędem obsługiwanym przez klasę `CFileException` powinien być powiązany łańcuch znaków, używany w funkcjach `CFileException::Dump()` oraz `AfxThrowFileException()`. Jednak projektanci tej klasy chcieli, by funkcje zgłaszające wyjątek `CFileException` mogły podawać jedynie wartość numeryczną oznaczającą przyczynę wyjątku. Te wartości numeryczne są zdefiniowane jako typ wyliczeniowy wewnątrz samej klasy `CFileException`. Ponieważ każdy element typu wyliczeniowego ma przypisaną wartość, w tej klasie jest zdefiniowana również tablica łańcuchów. Dzięki temu opis przyczyny błędu można znaleźć, używając wartości typu wyliczeniowego jako indeksu do statycznej tablicy napisów. Ponieważ ta metoda może być przydatna także we własnych klasach wyprowadzonych z klasy `CException`, na listingu 11.3 przytaczamy definicję typu wyliczeniowego oraz tablicę napisów.

Listing 11.3. *Struktura enum oraz statyczna tablica napisów*

```
enum {
    none,             // bez błędu
    generic,          // ogólny błąd
    fileNotFound,     // nie odnaleziony plik
    badPath,          // błędna ścieżka
    tooManyOpenFiles, // zbyt wiele otwartych plików
    accessDenied,     // odmowa dostępu
    invalidFile,      // błędny plik
    removeCurrentDir, // próba usunięcia bieżącej kartoteki
    directoryFull,    // zbyt wiele plików w kartotece
    badSeek,          // błąd przy ustawieniu wskaźnika plików
    hardIO,           // błąd sprzętowy
    sharingViolation, // błąd wspólnego dostępu do pliku
    lockViolation,    // próba zablokowania zablokowanego regionu
    diskFull,         // brak miejsca na dysku
    endOfFile         // osiągnięto koniec pliku
};
```

```
static const LPCSTR rgszCFileExceptionCause[] =
{
    "none",
    "generic",
    "fileNotFound",
    "badPath",
    "tooManyOpenFiles",
    "accessDenied",
    "invalidFile",
    "removeCurrentDir",
    "directoryFull",
    "badSeek",
    "hardIO",
    "sharingViolation",
    "lockViolation",
    "diskFull",
    "endOfFile"
};
```

Jak widać, każdej wartości typu wyliczeniowego odpowiada tekstowy opis w statycznej tablicy `rgszCFileExceptionCause`. Tak więc tworząc i zgłaszając obiekt `CFileException`, można to uczynić w poniższy sposób:

```
throw new CFileException(CFileException::fileNotFound);
```

To bardzo dobry przykład sposobu definiowania rodzajów błędów specyficznych dla własnej klasy wyjątku. Jednak istnieje jeszcze lepszy sposób powiązania tekstowego opisu z wyjątkiem. W odniesieniu do poprzedniego przykładu nie podaliśmy jednego drobnego szczegółu: statyczna tablica napisów jest zdefiniowania jedynie dla konfiguracji przeznaczonych dla debuggowania. Innymi słowy, ta statyczna tablica jest zdefiniowana wewnątrz bloku `#ifdef _DEBUG/#endif`. Skąd więc klasa `CFileException` odczytuje opis błędu i w jaki sposób te opisy są powiązane z wartością `int` przekazywaną do konstruktora? Odpowiedź brzmi: za pomocą pliku zasobów. Z powodu konieczności lokalizowania wszystkich komunikatów wyświetlanych użytkownikowi, klasa `CFileException` przechowuje opisy błędów w pliku zasobów. Jeśli rzucisz okiem na definicję funkcji `CFileException::GetErrorMessage()`, zauważysz co następuje:

```
BOOL CFileException::GetErrorMessage(LPTSTR lpszError,
                     UINT nMaxError, PUINT pnHelpContext)
{
    ASSERT(lpszError != NULL && AfxIsValidString(lpszError,
           nMaxError));

    if (pnHelpContext != NULL)
        *pnHelpContext = m_cause + AFX_IDP_FILE_NONE;

    CString strMessage;
    CString strFileName = m_strFileName;
    if (strFileName.IsEmpty())
        strFileName.LoadString(AFX_IDS_UNNAMED_FILE);
    AfxFormatString1(strMessage,
        m_cause + AFX_IDP_FILE_NONE, strFileName);
    lstrcpyn(lpszError, strMessage, nMaxError);

    return TRUE;
}
```

Kilka pierwszych linii sprawdza poprawność adresu argumentu `lpszError` oraz ustawia kontekst pomocy. Następnie pobierana jest nazwa pliku, do którego odnosi się wyjątek. Jeśli do konstruktora nie została przekazana nazwa pliku, z pliku zasobów ładowany jest łańcuch „nienazwany plik".

Zwróć uwagę na linię wywołującą funkcję `AfxFormatString1()`. Ta funkcja formatuje obiekt `CString`, używając łańcucha z zasobów wskazywanego przez identyfikator zasobu przekazywany w drugim argumencie. Jak widać, używany jest napis wskazywany przez identyfikator `AFX_IDP_FILE_NONE`. To co dzieje się w wersji ostatecznej, jest podobne do tego, co dzieje się w wersji dla debuggowania. Jedyna różnica polega na tym, że w wersji dla debuggowania jako indeks do statycznej tablicy napisów używana jest wartość wyliczeniowa. W wersji ostatecznej wartość wyliczeniowa jest dodawana do wartości identyfikatora `AFX_IDP_FILE_NONE` w celu otrzymania wartości identyfikatora zasobu. W ten sposób w przypadku ostatecznych wersji aplikacji korzystających dynamicznie z biblioteki MFC, opisy błędów są zawarte w bibliotece DLL MFC. W przypadku ostatecznych wersji aplikacji statycznie połączonych z biblioteką MFC te same zasoby są wstawiane do pliku aplikacji.



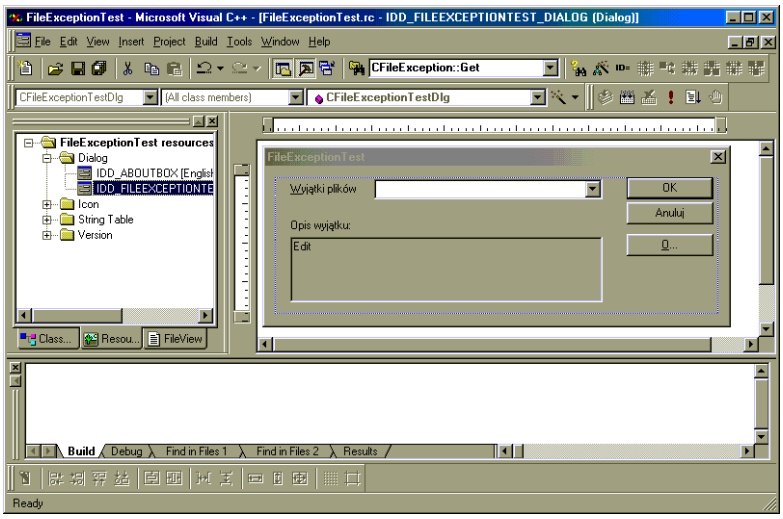
Przeglądanie zasobów w plikach binarnych

Jeśli chcesz przejrzeć zasoby pliku *.DLL* lub *.EXE*, możesz wykorzystać do tego Visual Studio. Na przykład, jeśli chcesz przejrzeć wspomniane zasoby opisów błędów w pliku *MFC42.DLL*, po prostu kliknij przycisk Open na pasku narzędzi, odszukaj plik w folderze *System32* i z rozwijanej listy Open As wybierz pozycję Resources (zasoby).

Przykładowy program CFileException

Oto demonstracyjny program przedstawiający klasę `CFileException` w działaniu. Program znajduje się na dołączonej do książki płytce CD-ROM, w kartotece *\Rozdz11\FileExceptionTest*. Ponieważ widziałeś sposób zdefiniowania klasy `CFileException`, celem programu jest pokazanie, jak łatwe jest zgłaszanie i wychwytywanie wyjątków tego typu.

1. Zacznij od użycia AppWizarda do stworzenia aplikacji opartej na oknie dialogowym; nazwij ją *FileExceptionTest*. Gdy zostaną utworzone pliki projektu, zmodyfikuj okno dialogowe `IDD_FILEEXCEPTIONTEST_DIALOG`, tak aby wyglądało podobnie jak na rysunku 11.2.
2. Po stworzeniu kontrolek okna dialogowego aplikacji użyj ClassWizarda do stworzenia zmiennej typu `CString` o nazwie `m_strFileExceptionDescription` dla pola opisu wyjątku w oknie dialogowym. Następnie stwórz zmienną typu `CComboBox` o nazwie `m_cboFileExceptions` dla rozwijanej listy Wyjątki plików.
3. Otwórz plik *FileExceptionTestDlg.cpp* i pod koniec funkcji `OnInitDialog()`, tuż przed instrukcją `return`, dopisz poniższy kod:



Rys. 11.2. *Okno dialogowe FileExceptionTest po dodaniu nowych kontrolek*

```
static struct
  {
      int iException;
      char szException[42];
  } FILE_EXCEPTIONS[] = {
  CFileException::none,              "Bez błędu",
  CFileException::generic,           "Ogólny błąd",
  CFileException::fileNotFound,      "Nie odnaleziony plik",
  CFileException::badPath,           "Błędna ścieżka",
  CFileException::tooManyOpenFiles, "Zbyt wiele otwartych "plików",

  CFileException::accessDenied,      "Odmowa dostępu",
  CFileException::invalidFile,       "Błędny plik",
  CFileException::removeCurrentDir,
                     "Próba usunięcia bieżącej kartoteki",
  CFileException::directoryFull,     "Zbyt wiele plików
                                     "w kartotece",
  CFileException::badSeek,
                     "Błąd przy ustawianiu wskaźnika plików",
  CFileException::hardIO,            "Błąd sprzętowy",
  CFileException::sharingViolation, "Błąd wspólnego dostępu
                                     "do pliku",
  CFileException::lockViolation,
                     "Próba zablokowania zablokowanego regionu",
  CFileException::diskFull,          "Brak miejsca na dysku",
  CFileException::endOfFile,         "Osiągnięto koniec pliku",
  };

  int iIndex;
  for (int i = 0; i < sizeof FILE_EXCEPTIONS /
       sizeof FILE_EXCEPTIONS[0]; i++)
  {
   iIndex = m_cboFileExceptions.AddString(
   FILE_EXCEPTIONS[i].szException);
   m_cboFileExceptions.SetItemData(iIndex,
                  (DWORD)FILE_EXCEPTIONS[i].iException);
  }
```

4. Następnie za pomocą ClassWizarda dodaj funkcję obsługi komunikatu `CBN_SELCHANGE` pochodzącego od rozwijanej listy (`IDC_COMBO1`). Ta funkcja wywoła funkcję `DisplayFileException()` za każdym razem, gdy z rozwijanej listy zostanie wybrana przyczyna błędu.

```
void CFileExceptionTestDlg::OnSelchangeCombo1()
{
  try
  {
    DisplayFileException();
    AfxMessageBox("Nigdy nie powinniśmy tu dojść!");
  }
  catch(CFileException* pe)
  {
    char sz[255];
    pe->GetErrorMessage(sz, 255);

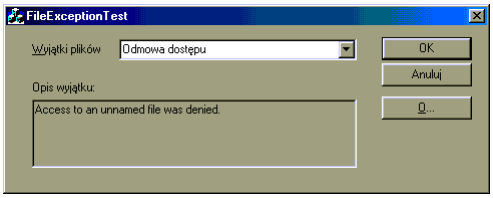
    m_strFileExceptionDescription = sz;
    UpdateData(FALSE);

    pe->Delete();
  }
}
```

5. Teraz zaimplementuj funkcję `DisplayFileException()`. Jak widać, w tej funkcji po prostu używamy danych elementu pozycji listy (ustawionych w funkcji `OnInitDialog()`) w celu wyznaczenia rodzaju błędu, który powinien zostać zgłoszony w wyjątku.

```
void CFileExceptionTestDlg::DisplayFileException()
{
  int iIndex;
  if (CB_ERR != (iIndex = m_cboFileExceptions.GetCurSel()))
  {
    int iFileException =
m_cboFileExceptions.GetItemData(iIndex);
    throw new CFileException(iFileException);
  }
}
```

6. Gdy w tym momencie zbudujesz i uruchomisz program, powinieneś ujrzeć widok podobny do pokazanego na rysunku 11.3.



Rys. 11.3. *Demonstracyjna aplikacja FileExceptionTest ilustruje zgłaszanie i wychwytywanie wyjątków typu CFileException*

Definiowanie własnych klas wyprowadzonych z klasy CException

Teraz, gdy wiesz już, jak zdefiniowana jest klasa `CFileException`, nadszedł czas, aby spróbować wyprowadzić własne klasy z klasy `CException`. Zacznijmy od wyobrażenia sobie rzeczywistego scenariusza, w którym mógłbyś wykorzystać wyjątki. Załóżmy, że opracowujesz system sprzedaży posiadający kilka okien dialogowych dla obsługi klientów. Zgodnie z zasadami obiektowo zorientowanego programowania okno dialogowe obsługi klienta będzie zawierało wskaźnik do klienta tworzonego lub obsługiwanego w tym oknie. Gdy użytkownik zdecyduje się na zapisanie klienta, okno powinno wywołać funkcję `UpdateData(TRUE)`, a następnie funkcję `CCustomer::Save()`.

Analogicznie do techniki MFC polegającej na wykorzystaniu klasy `CFileException` dla wszystkich wyjątków związanych z klasą `CFile`, stwórzmy klasę wyjątków o nazwie `CCustomerException` (wyprowadzoną z klasy `CException`), używaną w powiązaniu z klasą `CCustomer` (listing 11.4).

Listing 11.4. *Klasa wyjątku CCustomerException*

```
const int CUSTOMER_EXCEPTION_STRING_RESOURCE = 1000

class CCustomerException : public CException
{
DECLARE_DYNAMIC(CCustomerException)

public:
  CCustomerException(int iCause);
protected:
  int m_iCause;

public:
  enum
  {
    causeDuplicateCustomer = 0,
    causeInvalidTermsCode,
    causeInvalidSupplierCode
  }
public:
  virtual BOOL GetErrorMessage(LPTSTR lpszError,
    UINT nMaxError, PUINT pnHelpContext = NULL);
};
```

Zwróć uwagę na definicję `const int CUSTOMER_EXCEPTION_STRING_RESOURCE`. Będzie ona używana podobnie jak wartość `AFX_IDP_FILE_NONE` w funkcji `CFileException::GetErrorMessage()`. Innymi słowy, do tej wartości będzie dodawana zmienna składowa `m_iCause`, zaś wynik będzie wykorzystywany podczas działania programu jako identyfikator zasobu zawierającego opis przyczyny błędu.

Funkcje używające klasy `CCustomer` mogą teraz po prostu wychwytywać wyjątki typu `CCustomerException` i wywoływać ich funkcje składowe w celu wyświetlenia lub pobrania przyczyny błędu.

```
CCustomerMaintenanceDlg::OnOK()
{
  try
  {
    if (UpdateData(TRUE))
    {
      m_pCustomer->Save();
      CDialog::OnOK();
    }
  }
  catch(CCustomerException *pe)
  {
    TCHAR szErrorMessage[CHAR_MAX];
    if (pe->GetErrorMessage(szErrorMessage, _countof(szErrorMessage)))
    {
      AfxMessageBox(strErrorMessage);
    }

    pe->Delete();
  }
}
```

Najpierw jednak omówmy kilka problemów powstających przy tym podejściu. Po pierwsze, musiałbyś stworzyć nową klasę wyjątku dla każdej klasy w systemie, która może zgłaszać wyjątki. W bardzo dużym systemie daje to ogromną liczbę klas i plików, które trzeba konserwować. Po drugie, jeśli dla każdego wyjątku zostałaby stworzona inna klasa, różniąca się jedynie zawartością struktury `enum`, z pewnością nie byłoby to zgodne z zasadami programowania obiektowego. Innymi słowy, powinieneś deklarować nowe typy klas tylko wtedy, gdy nie istnieją klasy zawierające potrzebne funkcje. Tak więc musi istnieć lepszy sposób definiowania wyjątków dla całego systemu, nie wymagający tworzenia osobnej klasy dla każdego typu wyjątku. Na szczęście taki sposób istnieje i jest w dodatku bardzo prosty.

Zacznij od przyjrzenia się klasie, która została wyprowadzona z klasy `CException`, czyli klasie `CSimpleException`. Mówi się o niej, że jest wyjątkiem „zależnym od zasobów", gdyż jest oparta na łańcuchach napisów w zasobach. Jednak konieczne jest przejście przez kolejny poziom abstrakcji. Mimo że klasa `CSimpleException` deklaruje zmienną składową zawierającą identyfikator łańcucha w zasobach, jednak nie posiada funkcji ustawiającej wartość tego identyfikatora. Dopiero klasa `CUserException` (wyprowadzona z `CSimpleException`) posiada konstruktor, w którym jako jeden z parametrów jest przekazywany identyfikator łańcucha w zasobach. W ten sposób, gdy zostanie wywołana funkcja `GetErrorMessage()` (dziedziczona z klasy bazowej `CSimpleException`), ładowany jest właściwy zasób reprezentujący przyczynę wyjątku. Listing 11.5 przedstawia implementację funkcji `CSimpleException::GetErrorMessage()`.

Listing 11.5. Implementacja funkcji `CSimpleException::GetErrorMessage()`

```
BOOL CSimpleException::GetErrorMessage(LPTSTR lpszError, UINT
nMaxError,
        PUINT pnHelpContext)
{
    ASSERT(lpszError != NULL && AfxIsValidString(lpszError,
           nMaxError));
```

```
    if (pnHelpContext != NULL)
        *pnHelpContext = 0;

    // Jeśli nie załadowaliśmy naszego łańcucha (na przykład w
    // aplikacji konsoli), zwróć pusty łańcuch i wartość FALSE

    if (!m_bInitialized)
        InitString();

    if (m_bLoaded)
        lstrcpyn(lpszError, m_szMessage, nMaxError);
    else
        lpszError[0] = '\0';

    return m_bLoaded;
}

void CSimpleException::InitString()
{
    m_bInitialized = TRUE;
    m_bLoaded = (AfxLoadString(m_nResourceID,
        m_szMessage, _countof(m_szMessage)) != 0);
}
```

Teraz zamiast tworzyć nową klasę wyjątku dla każdej klasy potrafiącej zgłaszać wyjątki, aplikacja może tworzyć obiekt klasy `CUserException`, podając mu identyfikator łańcucha w zasobach i zgłaszać go jako wyjątek. Aby zobaczyć, jak łatwo to zaimplementować, spójrzmy na kod ilustrujący dwie przykładowe klasy mogące zgłaszać wyjątki: `CCustomer` oraz `CSupplier`, ponieważ `CUserException` wykonuje za nas większość pracy. Wszystko co musisz zrobić to zdefiniowanie przyczyn wyjątków w klasach `CCustomer` i `CSupplier`. Kod mógłby wyglądać następująco:

```
class CCustomer : public CObject
{
...
public:
  enum
  {
    causeDuplicateCustomer = 1000,
    causeInvalidTermsCode = 1001,
    causeInvalidSupplierCode 1002
...
};

class CSupplier : public CObject
{
...
public:
  enum
  {
    causeDuplicateSupplier = 2000,
    causeInvalidDiscountRate = 2001,
...
};
```

Przy tym podejściu, do wychwycenia wszystkich rodzajów wyjątków w aplikacji wystarczy tylko pojedyncza klasa wyjątku. W tym momencie musisz jeszcze tylko stworzyć łańcuchy w zasobach z identyfikatorami zgodnymi z identyfikatorami zdefiniowanymi w strukturach `enum` poszczególnych klas. Funkcja `CCustomerMaintenanceDlg::OnOK()` z poprzedniego przykładu teraz wygląda następująco:

```
CCustomerMaintenanceDlg::OnOK()
{
  try
  {
    if (UpdateData(TRUE))
    {
      m_pCustomer->Save();
      CDialog::OnOK();
    }
  }
  catch(CUserException *pe)
  {
    TCHAR szErrorMessage[CHAR_MAX];
    if (pe->GetErrorMessage(szErrorMessage,
        _countof(szErrorMessage)))
    {
      AfxMessageBox(strErrorMessage);
    }

    pe->Delete();
  }
}
```

Zaawansowane techniki obsługi wyjątków

Jak dotąd poznaliśmy koncepcję stosowania obsługi wyjątków, składnię związaną z ich zgłaszaniem i wychwytywaniem oraz kilka klas MFC sprawiających, że wszystko to jest bardzo proste. Jednak mimo iż poznaliśmy podstawy, pozostaje kilka bardzo ważnych zagadnień.

Powiedzmy, że funkcja A wywołuje funkcję B, która z kolei wywołuje funkcję C. Czy to, że funkcja C może zgłosić wyjątek danego typu, oznacza, że funkcja B powinna go wychwycić, nawet jeśli nie wie lub nie potrafi go obsłużyć? Jak kod powinien zostać podzielony ze względu na bloki `try` i `catch`? Czy ze zgłaszaniem wyjątków z funkcji wirtualnych wiążą się jakieś szczególne sytuacje? Gdy poznaliśmy już podstawy obsługi wyjątków w MFC, zajmijmy się wspomnianymi zagadnieniami, zagłębiając się nieco bardziej w mechanizm działania obsługi wyjątków, dążąc do tego, by nasz kod stał się jeszcze bardziej zwarty.

Decydowanie, która funkcja powinna wychwycić wyjątek

W tym momencie wiesz już, jak wychwycić wyjątek zgłaszany przez wywoływaną funkcję i wiesz także, w jaki sposób sterowanie przechodzi do kolejnych funkcji na stosie wywołań aż do napotkania odpowiedniego bloku `catch`. Pytanie więc brzmi: czy blok `try` powinien wychwytywać wszystkie możliwe wyjątki, które mogą być zgłaszane przez funkcje wywoływane wewnątrz bloku? Odpowiedź brzmi: nie. Jak już wspomnieliśmy, dwie z największych zalet korzystania w aplikacji z obsługi wyjątków wiążą się ze zmniejszeniem ilości kodu i mniejszym wysiłkiem przy jego konserwacji. Ilustruje to listing 11.6.

Listing 11.6. *Zalety korzystania z obsługi wyjątków*

```
#include <iostream.h>

int main()
{
    try
    {
        PerformWork();
        cout << "Program zakończył działanie bez problemów.";
    }
    catch(char* szException)
    {
        cout << szException;
    }

    return 0;
}

void PerformWork()
{
    try
    {
        char* pBuffer;
        GetBuffer(&pBuffer, 512);
        cout << pBuffer;
    }
    catch(char* szException)
    {
        throw szException;
    }
}

void GetBuffer(char** ppBuffer, unsigned int uiSize)
{
    *ppBuffer = NULL;
    *ppBuffer = new char(uiSize);
    if (NULL == *ppBuffer)
    {
        throw "Błąd pamięci: Nie powiodła się alokacja pamięci.";
    }
}
```

Jak widać na listingu 11.6, funkcja `PerformWork()` wychwytuje ewentualny wyjątek zgłaszany przez funkcję `GetBuffer()`, mimo że nie może z tym wyjątkiem zrobić nic innego poza zgłoszeniem go dalej. Funkcja `main()` wychwytuje ponownie zgłoszony wyjątek i rzeczywiście go obsługuje (w tym wypadku po prostu wyświetlając komunikat błędu). Oto kilka powodów, dla których funkcje jedynie zgłaszające wyjątki dalej nie powinny ich wychwytywać:

- Ponieważ funkcja niczego nie robi z wychwyconym wyjątkiem, blok `catch` w takiej funkcji jest zwyczajnie nadmiarowy. Oczywiście, nigdy nie powinno się tworzyć kodu tylko dlatego, że można to robić.
- Jeśli zmieniłby się typ wyjątku zgłaszanego przez funkcję `GetBuffer()`, musiałbyś wrócić i zmienić zarówno funkcję `main()`, jak i `PerformWork()`. Wiąże się to więc ze zwiększonymi kosztami konserwacji aplikacji.
- Wielokrotne dalsze zgłaszanie wyjątków w dużej aplikacji może obniżyć wydajność systemu. Dzieje się tak, ponieważ w celu zwykłego przekazania wyjątku dalej konieczne jest wywoływanie bloku `catch` w każdej funkcji na stosie wywołań.

Ponieważ C++ automatycznie przechodzi w górę stosu wywołań aż do napotkania funkcji będącej w stanie wychwycić wyjątek, funkcje pośrednie mogą ignorować te wyjątki, których nie potrafią przetworzyć. Przypomnij sobie także, że jeśli nie zostanie znaleziona funkcja z odpowiednim blokiem `catch`, działanie aplikacji zostaje przerwane.

Wybór kodu przeznaczonego do umieszczenia w bloku try

Określenie, jaki kod powinien znaleźć się w bloku `try`, często jest jedynie stylistycznym zagadnieniem. Zwykle w bloku `try` umieszcza się tylko ten kod, którego wykonanie może spowodować zgłoszenie wyjątku. Jednak wielu programistów w celu zachowania czytelności umieszcza w bloku `try` prawie cały kod funkcji. Tak więc często zdarza się napotkać kod w rodzaju poniższego:

```
void SomeClass::SomeFunction(CWnd* pWnd)
{
    try
    {
        CString strWindowText;
        pWnd->GetWindowText(strWindowText);

        m_pRecord->Save(strWindowText);

        ...
    }
    catch(CException *pe)
    {
        pe->ReportError();
        pe->Delete();
    }
}
```

Jak widać, w funkcji `SomeClass::SomeFunction()` dwie pierwsze linie kodu definiujące obiekt `CString` i pobierające tytuł okna nie mają żadnego wpływu na to, czy funkcja `Save()` zgłosi wyjątek. Tak więc te dwie linie kodu mogą zostać umieszczone przed blokiem `try`:

```
void SomeClass::SomeFunction(CWnd* pWnd)
{
    CString strWindowText;
    pWnd->GetWindowText(strWindowText);

    try
    {
        m_pRecord->Save(strWindowText);

        ...
    }
    catch(CException *pe)
    {
        pe->ReportError();
        pe->Delete();
    }
}
```

Jednak jak już wspominaliśmy, w rzeczywistości jest to jedynie stylistyczna zmiana. Choć umieszczenie całego kodu wewnątrz bloku `try` może poprawić jego czytelność, jednak obie techniki są równie poprawne.

Wybór kodu przeznaczonego do umieszczenia w bloku catch

Jednym kodem, który powinien pojawić się w bloku `catch`, jest kod, który przynajmniej częściowo przetwarza wyjątek. Na przykład, w przypadku gdy funkcja wychwytuje wyjątek, powinna zrobić co może w celu jego obsłużenia, a następnie zgłosić wyjątek dalej w celu jego dalszej obróbki. Scenariusz ilustrujący takie podejście jest zawarty na listingu 11.7.

Listing 11.7. *Przetwarzanie wyjątku*

```
CMyDialog::OnOK()
{
    try
    {
        m_pDoc->SaveInvoice(*m_pInvoice);
        CDialog::OnOK();
    }
    catch(CException *pe)
    {
        pe->ReportError();
        pe->Delete();
    }
}
```

```
CMyDoc::SaveInvoice(CInvoice const& rInvoice)
{
    try
    {
        m_pDB->Commit();

        m_pDB->SaveInvoiceHdr(rInvoice->GetHeader());
        m_pDB->SaveInvoiceDtl(rInvoice->GetDetail());

        m_pDB->Commit();

        SetModifiedFlag(TRUE);
    }
    catch(CDBException* pe)
    {
        m_pDB->Rollback();

        throw pe;
    }
}
```

W listingu 11.7 do wprowadzania zgłoszeń służy dialog `CMyDialog`. Przed wywołaniem funkcji `OnOK()`, dialog próbuje użyć osadzonego obiektu dokumentu (`m_pDoc`) w celu zapisania zgłoszenia. Jeśli funkcja `SaveInvoice()` zgłosi wyjątek, funkcja `CMyDialog::OnOK()` wychwyci go i wyświetli komunikat błędu. W przeciwnym wypadku okno dialogowe zostaje zamknięte (jako że zgłoszenie zostało zapisane).

Ten rodzaj zgłaszania i wychwytywania widziałeś już w tym rozdziale nie raz. Przyjrzyj się jednak funkcji `CMyDoc::SaveInvoice()`. Ta funkcja wywołuje funkcję `Commit()`, zapisuje dane, a następnie ponownie wywołuje funkcję `Commit()`. Spójrz teraz na blok `catch`. Zanim funkcja ponownie zgłosi wyjątek, wywołuje funkcję `Rollback()` w celu anulowania niepełnych zmian dokonanych w bazie danych. Jest to właśnie przykład funkcji wychwytującej wyjątek, wykonującej pewne wewnętrzne porządki, a następnie ponownie zgłaszającej wychwycony wyjątek.

Zgłaszanie wyjątków w funkcjach wirtualnych

Zgłaszanie wyjątków w funkcjach wirtualnych może być zagmatwane. Jak zawsze w przypadku przesłonięcia funkcji wirtualnej, nigdy nie powinieneś robić niczego, czego nie mógłby obsłużyć wcześniej stworzony kod. Innymi słowy, powiedzmy, że masz deklarację klasy taką jak na listingu 11.8.

Listing 11.8. *Zgłaszanie wyjątku w funkcji wirtualnej*

```
class CTest
{
    ...
public:
    virtual void DoSomething(); // zgłasza wyjątek CTestException

    ...
}
```

```
class CTestException : public CException
{
    ...

public:
    void LogException();

    ...
};

CSomeClass::SomeFunction(CTest* pTest)
{
    ...
    try
    {
        pTest->DoSomething();
    }
    catch(CTestException *pe)
    {
        pe->LogException();
        pe->Delete();
    }
    ...
}
```

Na listingu 11.8 wyjątek został wyprowadzony z klasy bazowej `CException`. Klasa pochodna ma możliwość zapisu wyjątku do pliku w wyniku wywołania funkcji `LogException()`. Tak więc `CSomeClass::SomeFunction()` umieszcza wywołanie funkcji `DoSomething()` w bloku `try` w celu wychwycenia ewentualnego wyjątku `CTestException`.

Co się teraz stanie, jeśli inny programista wyprowadzi nową klasę z klasy `CTest`, przesłoni jej wirtualną funkcję składową `DoSomething()` i zechce zgłosić wyjątek innego typu? Odpowiedź jest prosta. Jeśli wyjątek będzie wyprowadzony z klasy udokumentowanej jako klasa wyjątku zgłaszanego przez funkcję `DoSomething()`, nie ma problemu. Innymi słowy, cały istniejący kod działa tak jak dotychczas. Jeśli jednak nowa implementacja funkcji `DoSomething()` zgłasza wyjątek nie udokumentowany podczas tworzenia kodu korzystającego z oryginalnej funkcji `DoSomething()`, musisz się cofnąć i zmodyfikować cały istniejący kod wywołujący tę funkcję. W przeciwnym razie ryzykujesz załamanie istniejącego kodu w momencie zgłoszenia wyjątku nowego typu. Takie załamanie kodu ilustruje listing 11.9.

Listing 11.9. *Zgłaszanie nieudokumentowanego wyjątku*

```
class CNewTest : public Ctest
{
    ...

public:
    // zgłasza wyjątek CNewTestException
    virtual void DoSomething();
    ...
};
```

```
CSomeClass::SomeFunction(CTest* pTest)
{
    try
    {
        pTest->DoSomething();
    }
    catch(CTestException *pe)
    {
        pe->LogException();
        pe->Delete();
    }
}
```

Teraz, gdy zostanie wywołana funkcja `CSomeClass::SomeFunction()` ze wskaźnikiem do obiektu `CNewTest` i funkcja `DoSomething()` zgłosi wyjątek, nie zostanie on wychwycony. Jedną z największych zalet C++ jest możliwość dodawania nowego kodu bez konieczności przepisywania kodu już istniejącego. Jednak w tym przypadku istniejący kod uległby załamaniu. Powodem jest to, że funkcja `DoSomething()` była udokumentowana jako zgłaszająca wyjątki jednego typu, zaś wersja przesłonięta zgłasza wyjątki innego typu. Na szczęście istnieją dwa sposoby rozwiązania tego problemu.

Gdy przesłaniasz funkcję udokumentowaną jako zgłaszająca wyjątki pewnego typu i chcesz zgłosić inny rodzaj wyjątku, zgłoś wyjątek wyprowadzony z oryginalnie zgłaszanego wyjątku. Na listingu 11.9 oznacza to, że ponieważ funkcja `DoSomething()` była oryginalnie udokumentowana jako zgłaszająca wyjątki typu `CTestException`, funkcja `CNewTest::DoSomething()` powinna zgłaszać wyjątki jedynie tej klasy lub jej klasy pochodnej.

Innym sposobem rozwiązania tego problemu w klasie wywołującej jest ogólne wychwytywanie wyjątków tego typu, który stanowi klasę bazową dla wszystkich wyjątków w systemie. Następnie funkcja wywołująca może użyć funkcji `IsKindOf()` w celu sprawdzenia typu danego wyjątku. To podejście powinno być stosowane zwłaszcza wtedy, gdy projektujesz klasę zawierającą funkcje, o których wiesz, że zostaną przesłonięte. Listing 11.10 przedstawia lepszy sposób obsługi wyjątków niż przykład z listingu 11.9.

Listing 11.10. *Wychwytywanie wyjątku reprezentującego klasę bazową dla wszystkich innych rodzajów wyjątków*

```
class CTest
{
    ...

public:
    // W przypadku błędu funkcja DoSomething() zgłasza
    // wyjątek CTestException (wyprowadzony z CException)
    virtual void DoSomething(); // zgłasza CException

    ...
}

class CTestException : public CException
{
    ...
```

```
public:
    void LogException();

    ...
};

class CNewTest : public Ctest
{
    ...

public:
    // W przypadku błędu funkcja DoSomething() zgłasza
    // wyjątek CNewTestException (NIE wyprowadzony z CException)
    virtual void DoSomething();
    ...
};

CSomeClass::SomeFunction(CTest* pTest)
{
    ...
    try
    {
        pTest->DoSomething();
    }
    catch(CTestException *pe)
    {
        pe->LogException();
        pe->Delete();
    }
    catch(CException *pe)
    {
        TCHAR szErrorMessage[512];
        if (pe->GetErrorMessage(szErrorMessage,
            _countof(szErrorMessage)))
        {
            // Wywołaj globalną funkcję zapisującą
            // wyjątki do pliku dziennika, przekazując
            // jej komunikat szErrorMessage, dzięki czemu
            // do dziennika zostaną zapisane wszystkie wyjątki
        }

        pe->Delete();
    }
}
```

Podsumowanie

Jak widziałeś w tym rozdziale, składnia obsługi wyjątków jest prosta, klasy wyjątków MFC nie są skomplikowane, zaś implementacja obsługi wyjątków we własnej aplikacji sprowadza się po prostu do wcześniejszego zdefiniowania odpowiednich funkcji. To, co sprawia, że obsługa wyjątków jest tak przydatna, to nie prosta składnia czy rozbudowana koncepcja dziedziczenia klas C++. To właśnie konsekwentne używanie wyjątków w aplikacji oraz płynące z tego korzyści sprawiają, że obsługa wyjątków jest ważnym narzędziem przy tworzeniu aplikacji stabilnych i posiadających pełną kontrolę błędów.