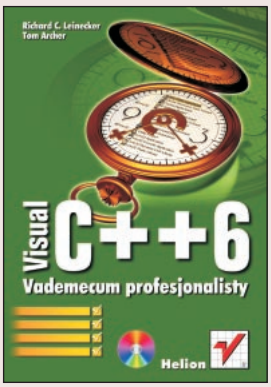
Visual C++ 6.
Vademecum profesjonalisty



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++

Część III Architektura

Część IV Programowanie baz danych

Część V Rozszerzanie aplikacji

Część VI Programowanie modelu COM

Część VII Internet i programowanie HTML

Część VIII Zaawansowane korzystanie z Visual Studia

Dodatki

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:

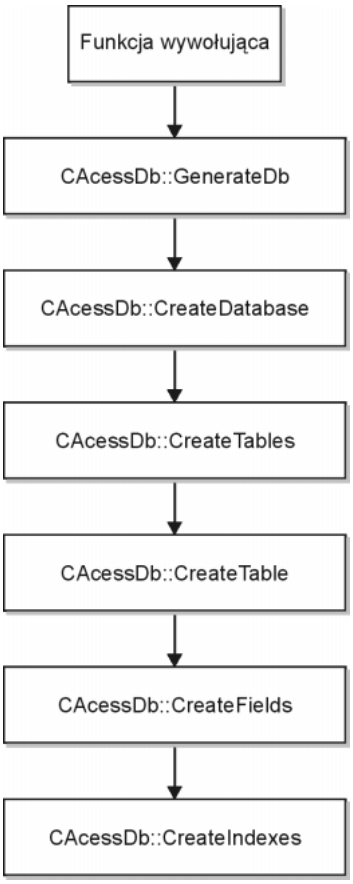
```
try
{
  CCommaDelimitedFile file;
  file.Open("C:\\test.csv");

  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.

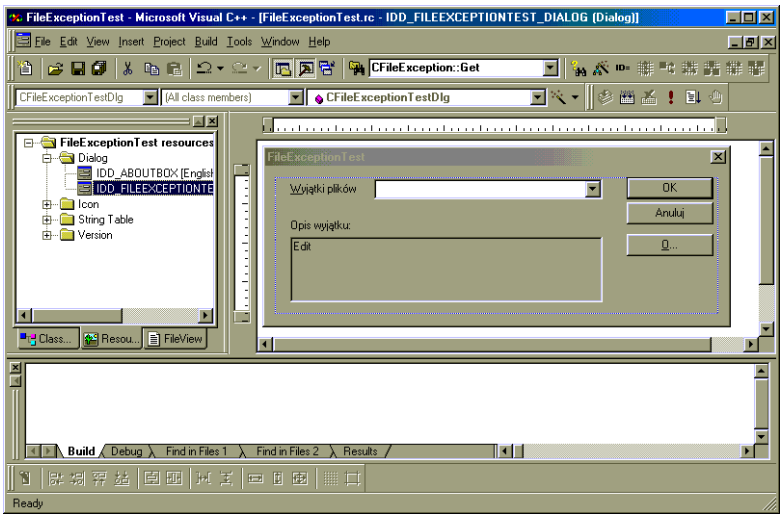
Przeglądanie zasobów w plikach binarnych

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);
  }
```

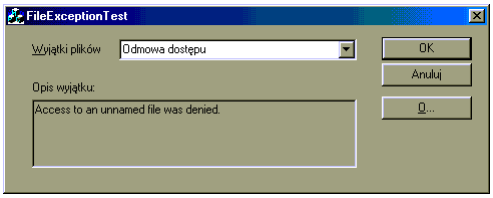
4. Następnie za pomocą ClassWizarda dodaj funkcję obsługi komunikatu `CBN_SELCHANGE` pochodzącego od rozwijanej listy (`IDC_COMBO1`). Ta funkcja wywoła funkcję `DisplayFileException()` za każdym razem, gdy z rozwijanej listy zostanie wybrana przyczyna błędu.

```
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.