helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

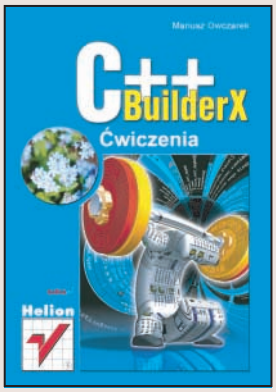
TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++BuilderX. Ćwiczenia

Autor: Mariusz Owczarek
ISBN: 83-7361-676-4
Format: B5, stron: 128



C++BuilderX to narzędzie, które ma umożliwić tworzenie aplikacji w języku C++ dla różnych platform systemowych. Na wszystkich platformach wygląd jego interfejsu jest identyczny, co bardzo ułatwia pracę. Narzędzie to, poza możliwością tworzenia aplikacji w sposób wizualny, udostępnia programistom również rozbudowany edytor kodu źródłowego oraz edytor HTML i XML. C++BuilderX staje się coraz popularniejszym środowiskiem do tworzenia aplikacji wieloplatformowych.

„C++BuilderX. Ćwiczenia" to książka dla tych, którzy chcą poznać to narzędzie i nauczyć się podstaw programowania z użyciem C++BuilderX. Dzięki zawartym w niej wiadomościom dowiesz się, jak korzystać ze środowiska wizualnego i jak stworzyć proste programy dla systemów Windows i Linux.

- Podstawy korzystania z C++BuilderX
- Podstawowe wiadomości o C++
- Aplikacje uruchamiane z poziomu konsoli
- Tworzenie aplikacji z interfejsem graficznym
- Korzystanie z różnych kompilatorów

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

Spis treści

Wstęp **5**

Rozdział 1. Pierwsze kroki w środowisku C++BuilderX **7**

Co jest potrzebne, aby korzystać z C++BuilderX? 7
Pierwsze spotkanie z C++BuilderX 8
Menu i paski narzędzi 8
Okna 10
Podsumowanie 14

Rozdział 2. Aplikacje konsolowe **15**

Ogólna postać programu pisanego w C++ 15
Dyrektywy 17
Dyrektywa #include 17
Dyrektywa #define 18
Dyrektywa #if — kompilacja warunkowa 19
Typy zmiennych 19
Zmienne typu int 20
Zmienne typu float 20
Typ double 21
Typ char 21
Modyfikatory typów 21
Rzutowanie (konwersja) typów 22
Typ wyliczeniowy 23
Operatory 24
Zapis danych do plików i odczyt z nich za pomocą operatora << i >> 26
Wskaźniki 28
Tablice 30
Operatory new i delete 32
Instrukcje 33
Instrukcje selekcji (wyboru) 34
Instrukcje iteracyjne 37
Funkcje 42
Przeciążanie funkcji 43
Niejednoznaczność 45
Przekazywanie argumentów przez wartość i adres 46
Wskaźniki do funkcji 49

Funkcja main() ... 50
Przekazywanie parametrów do funkcji main() ... 50
Struktury i unie ... 52
Struktury ... 52
Unie ... 54
Klasy ... 55
Konstruktory i destruktory ... 57
Przeładowywanie konstruktorów ... 59
Dziedziczenie ... 60
Przeciążanie operatorów ... 64
Wyjątki ... 66
Podsumowanie ... 69

Rozdział 3. Aplikacje okienkowe ... 71
Wstęp ... 71
Pierwszy program GUI — funkcja WinMain() ... 72
Główne okno aplikacji ... 76
Podstawy obsługi kontrolek ... 81
Kalkulator ... 84
Najprostszy edytor tekstu ... 88
Ulepszamy edytor ... 93
Rysowanie w oknie aplikacji ... 96
Odtwarzanie dźwięku z plików .wav ... 99
Podsumowanie ... 103

Rozdział 4. Zarządzanie definicjami obiektów w C++BuilderX ... 105
Class browser ... 105
Okno Structure View ... 110
Podsumowanie ... 112

Rozdział 5. Użytkowanie kilku kompilatorów i projekty makefile ... 113
Kompilacja warunkowa ... 115
Opcje linii komend ... 116
Konfiguracje budowy aplikacji ... 117
Konfiguracje Debug Build i Release Build ... 117
Własne konfiguracje budowy aplikacji ... 118
Dodawanie własnych kompilatorów do środowiska ... 119
Budowanie aplikacji ze zbiorów makefile ... 123
Podsumowanie ... 125

Literatura ... 127

Rozdział 3.

Aplikacje okienkowe

Wstęp

C++BuilderX w przeciwieństwie do C++Buildera 6 nie zawiera elementów VCL. Jesteśmy więc zmuszeni pisać programy z GUI przy użyciu WinAPI. Ma to swoje dobre strony, ponieważ pozwala na lepszą optymalizację kodu, ale jednocześnie zwiększa ilość czasu i pracy potrzebnej do napisania aplikacji. W pierwszej wersji BuilderaX przewidziane jest pisanie aplikacji GUI tylko pod systemem Windows. Dodatkowym utrudnieniem jest brak możliwości tworzenia menu i okien dialogowych w plikach zasobów. Edytor menu i okien dialogowych pojawi się zapewne w drugiej wersji BuilderaX. Tak naprawdę możliwe jest pisanie programów GUI również pod Linuksem, ale należy się w tym celu posłużyć którymś z narzędzi ułatwiających to zadanie. Takim narzędziem jest m.in. pakiet GTK++ dostępny na licencji *open source*.

Ponieważ w drugiej wersji opcja projektów GUI prawdopodobnie będzie dostępna także pod Linuksem, zdecydowałem się nie opisywać tu GTK++ ani innego tego typu pakietu, ale poczekać na rozwiązanie zaproponowanie przez firmę Borland. Jeżeli będziesz, Czytelniku, zainteresowany środowiskiem GTK, to na stronie projektu (*www.gtk.org*) znajdziesz obszerne opisy i tutoriale. Można się z nich nauczyć pracy z tą biblioteką od samych podstaw do skomplikowanych aplikacji.

Mam nadzieję, że także dla Windows ewolucja programu pójdzie w kierunku jakiejś formy biblioteki VCL, która jednak bardzo ułatwia pracę.

Na zakończenie wstępu kilka słów o ćwiczeniach w tym rozdziale. Ponieważ aplikacje w WinAPI są dosyć długie, nie jest możliwe napisanie ich w jednym ćwiczeniu. Dlatego ten rozdział podzielony jest na podrozdziały opisujące jedną aplikację (edytor, kalkulator itd.). Wszystkie ćwiczenia w danym podrozdziale odnoszą się do tego samego kodu źródłowego, aplikacja jest rozwijana w kolejnych podpunktach.

Pierwszy program GUI — funkcja WinMain()

Ćwiczenie 3.1.

Aby stworzyć projekt, którego wynikiem ma być aplikacja okienkowa:

1. Wybieramy z menu *File* opcję *New* i w oknie *Object Gallery* wybieramy *New GUI Application*. Pojawi się kreator projektów GUI.
2. Pierwszym krokiem jest wpisanie nazwy projektu i katalogu, w którym zostaną utworzone pliki programu. W polu nazwy pliku wpiszmy `Hallo`, a następnie kliknijmy *Next>*.
3. Teraz należy wybrać platformę, na której będzie działać nasz projekt. Ponieważ będzie to aplikacja WinAPI, domyślnie wybrany jest Windows. Obowiązują tutaj takie same zasady wyboru kompilatorów, jak dla programów wykonywanych w oknie konsoli.
4. Ostatnim krokiem jest wskazanie, jakie pliki mają wejść w skład projektu. Domyślnie projekt zawiera jeden plik, nazwany *untitled1.cpp* i zawierający kod programu z funkcją `WinMain()`. Jest to odpowiednik funkcji `main()`. Aby pik ten został utworzony, musimy zaznaczyć okienko w kolumnie *Create* obok nazwy pliku. Klikając na przycisk *Add*, możemy otworzyć okno umożliwiające dołączenie innych plików do projektu, podobnie jak to było w aplikacji konsolowej. Mogą to być pliki nagłówkowe, z obrazami graficznymi czy binarne pliki zasobów. Do odłączania plików od projektu służy przycisk *Remove*. Do naszego pierwszego projektu wystarczy plik z kodem programu, klikamy więc opisywane wyżej okienko *Create*, a następnie *Finish*.
5. Piki projektu zostaną utworzone w wybranym przez nas katalogu, a w oknie *Project content* zobaczymy ich listę. Będą to: plik projektu *Hallo.cbx*, plik *untitled1.cpp* z kodem programu i plik *Hallo.exe*, który będzie zawierał gotowy program po kompilacji.

Z czego składa się projekt GUI Windows?

Kliknijmy dwukrotnie plik *untitled1.cpp*. Jego zawartość pojawi się w oknie kodu źródłowego, a w oknie widoku struktury zobaczymy plik nagłówkowy i funkcje istniejące w programie. Na razie jedynym plikiem nagłówkowym jest *windows.h*, a jedyną funkcją `WinMain`. Plik *windows.h* zawiera funkcje WinAPI. Funkcja `WinMain` posiada parametry inne niż funkcja `main` dla aplikacji konsolowych i na niej się na chwilę zatrzymamy. Oto postać `WinMain()`:

```
int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nCmdShow )
```

Pierwszy parametr, `hInstance`, jest uchwytem do aplikacji, której częścią jest dana funkcja `WinMain`. Drugi parametr to uchwyt do poprzedniego uruchomienia danej aplikacji (jeżeli dwie kopie aplikacji działają w tym samym momencie). W środowisku Windows ten

parametr ma zawsze wartość NULL. Jeżeli chcemy sprawdzić, czy uruchomiono wcześniej jakieś kopie programu, musimy to zrobić w inny sposób. Trzeci parametr, lpCmdLine, to łańcuch znakowy zawierający argumenty przekazane do programu w linii komend przy uruchamianiu. Ten mechanizm omówię w dalszej części niniejszego rozdziału. Ostatni parametr zawiera flagi określające sposób wyświetlania głównego okna programu. Funkcja WinMain jest wywoływana przez system przy uruchamianiu aplikacji i zwykle wartość żadnego z tych parametrów nie jest bezpośrednio zadawana przez programistę.

Ćwiczenie 3.2.

Pierwszy program GUI:

Aby nie przedłużać, zabierzmy się za nasz pierwszy program w WinAPI. Będzie to proste okno dialogowe z napisem Witaj Przygodo.

1. W funkcję WinMain wpiszmy instrukcję:

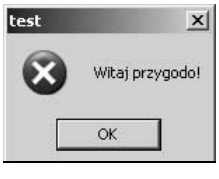
```
MessageBox(NULL,"Witaj przygodo","test",MB_OK | MB_ICONINFORMATION | MB_TASKMODAL);
```

   Teraz cały listing naszego programu wygląda tak:

```
#include <windows.h>
#ifdef __BORLANDC__
  #pragma argsused
#endif
int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow )
{
  MessageBox(NULL,"Witaj przygodo","test",MB_OK | MB_ICONINFORMATION | MB_TASKMODAL);
  return 0;
}
```

2. Wybieramy z menu *Run* opcję *Run Project.*
3. Nastąpi kompilacja, a następnie uruchomienie programu. Na ekranie ujrzymy nasze okienko. Powinno wyglądać tak (rysunek 3.1):

Rysunek 3.1.
Pierwszy program



Jeżeli program nie uruchomił się, sprawdźmy, czy kod źródłowy wygląda dokładnie tak, jak na listingu wyżej.

Jeżeli wszystko przebiegło bez zakłóceń, to właśnie napisałeś i uruchomiłeś, Czytelniku, swój pierwszy program okienkowy w C++BuilderX.

Choć programik jest tak prosty, można wykorzystać go do nauki dwóch rzeczy. Po pierwsze, przyjrzymy się funkcji MessageBox. Po drugie, poznamy mechanizm przekazywania parametrów w linii komend przy uruchamianiu programu.

Funkcja MessageBox, jak już wiemy, tworzy okno komunikatów. Jest to proste okno wyświetlające tekst wiadomości i dające użytkownikowi możliwość reakcji przez naciśnięcie

przycisku. Można sterować rodzajem wyświetlanych przycisków. Po naciśnięciu przycisku okno znika, a funkcja zwraca odpowiednią wartość w zależności od naciśniętego przycisku. Prototyp tej funkcji wygląda następująco:

```
int MessageBox(HWND hWnd,LPCTSTR lpText,LPCTSTR lpCaption,UINT uType)
```

Parametr `hWnd` oznacza uchwyt do okna macierzystego, do tego okna zostanie przesłany komunikat po naciśnięciu przycisku. W naszym przypadku nie ma okna macierzystego, więc ten parametr ma wartość `NULL`. Druga zmienna to łańcuch komunikatu wyświetlany w oknie. `lpCaption` to również łańcuch znakowy wyświetlany w pasku aplikacji. Ostatni parametr, `uType`, to flagi określające, jakie przyciski będą dostępne w oknie, jaka będzie ikona i sposób zachowania się okna.

Ćwiczenie 3.3.

Aby zmienić wygląd okna komunikatów:

1. Zmieńmy ostatni parametr funkcji z `MB_OK` na `MB_OKCANCEL`:

```
MessageBox(NULL,"Witaj przygodo","test",MB_OKCANCEL | MB_ICONINFORMATION | MB_TASKMODAL);
```

2. Po uruchomieniu programu okno ma dwa przyciski, *OK* i *Anuluj*:
3. Zamiast `MB_ICONINFORMATION` wpiszmy `MB_ICONSTOP`:

```
MessageBox(NULL,"Witaj przygodo","test",MB_OK | MB_ICONSTOP | MB_TASKMODAL);
```

4. Po kompilacji otrzymamy okno z ikoną błędu krytycznego systemu. Na szczęście to tylko ikona.

Inne możliwe wartości flag zestawiono w tabeli 3.1 i tabeli 3.2.

Tabela 3.1. *Wybór rodzaju i liczby przycisków*

Wartość flagi	Przyciski
`MB_OK`	*OK*
`MB_ABORTRETRYIGNORE`	*Przerwij*, *Ponów próbę* oraz *Ignoruj*
`MB_OKCANCEL`	*OK* oraz *Anuluj*
`MB_RETRYCANCEL`	*Ponów próbę* i *Anuluj*
`MB_YESNO`	*Tak* oraz *Nie*
`MB_YESNOCANCEL`	*Tak*, *Nie* i *Anuluj*

Tabela 3.2. *Wybór rodzaju ikony z boku wiadomości*

Wartość	Ikona
`MB_ICONEXCLAMATION`	ostrzegawczy wykrzyknik
`MB_ICONINFORMATION`, `MB_ICONASTERISK`	„dymek” informacyjny
`MB_ICONQUESTION`	„dymek” ze znakiem zapytania
`MB_ICONSTOP`	błąd krytyczny

Sposób zachowania się okna dialogowego:

- `MB_APPLMODAL` — Okno macierzyste komunikatu pozostaje nieaktywne, dopóki użytkownik nie wybierze jednego z przycisków.
- `MB_SYSTEMMODAL` — Tak samo jak `MB_APPLMODAL`. Dodatkowo okno komunikatu jest zawsze na pierwszym planie. Ta opcja ma zastosowanie do ważnych komunikatów, wymagających natychmiastowej uwagi użytkownika.
- `MB_TASKMODAL` — Tak samo jak `MB_APPLMODAL`, ale jeżeli parametr `hWnd` wynosi `NULL`, zostaną zablokowane wszystkie okna aplikacji, a nie tylko macierzyste.

Zachęcam do eksperymentowania z flagami i obserwacji, jak będzie się zmieniać okno dialogowe.

Ćwiczenie 3.4.

Uruchomienie programu w C++BuilderX z przekazywaniem argumentów z linii komend:

1. Zmieńmy program tak, aby w oknie wiadomości wyświetlał się parametr przekazany w linii komend. Jak już pisałem, parametry te są przekazywane w parametrze `lpCmdLine` funkcji `WinMain`. Wystarczy więc podstawić ten parametr za zmienną `lpText` funkcji `MessageBox`:

```
MessageBox(NULL,lpCmdLine,"test",MB_OK | MB_ICONINFORMATION | MB_TASKMODAL);
```

2. Aby nakazać uruchamianie aplikacji z dodatkową zmienną, wchodzimy w menu *Run* i wybieramy opcję *Configurations*.
3. Ukaże się okno *Project Properties*, wybieramy zakładkę *Run* (jest ona wybrana domyślnie). W oknie *Runtime configurations* powinna być zaznaczona konfiguracja dla naszej aplikacji *Hello*.
4. Klikamy przycisk *Edit*. Pojawi się okno edycji konfiguracji.
5. Znajdujemy okno *Command line argument* i wpisujemy tam cokolwiek. Pogram będzie uruchamiany z argumentem, jaki tam wpisaliśmy.
6. Następnie klikamy *OK* i znowu *OK*, a następnie kompilujemy i uruchamiamy program tak jak wcześniej.
7. Tekst wpisany jako argument pojawi się w oknie wiadomości. Warto zwrócić uwagę na dolne okno pokazujące proces kompilacji. Podczas działania programu będzie tam komenda uruchamiająca nasz program wraz z argumentem.

Wejdźmy jeszcze raz w menu *Run\Configurations*. *Runtime configurations* to wygodne narzędzie do testowania różnych opcji uruchamiania programu. Zamiast zmieniać istniejącą konfigurację, możemy dodać nową przyciskiem *New*. Każda konfiguracja może mieć inne opcje. Gdy mamy więcej niż jedną konfigurację, program zostanie uruchomiony w tej, przy której zaznaczymy okienko w kolumnie *Default*. Do tego narzędzia wrócimy jeszcze w dalszych przykładach.

W następnym podrozdziale napiszemy główne okno aplikacji „z prawdziwego zdarzenia". Będzie ono bazą do umieszczania różnych komponentów.

Główne okno aplikacji

Nasz pierwszy program składał się tylko z jednego okna dialogowego. Teraz napiszemy pierwszą „prawdziwą" aplikację, a raczej jej okno główne. Aplikacja Windows składa się z okien, między którymi przesyłane są komunikaty. Główne okno aplikacji otrzymuje komunikaty z systemu i odpowiednio na nie reaguje. Z oknem związana jest tzw. *funkcja zwrotna* (ang. *callback function*), do której przekazywane są komunikaty. Zawiera ona procedury obsługi tych komunikatów w zależności od ich treści.

Okno jest obiektem klasy `WNDCLASS`. Podczas tworzenia nowego okna musimy wygenerować nowy obiekt. Najpierw zadamy odpowiednie wartości składowych klasy. W naszym przykładzie nazwiemy ją `glowne_okno`. Postaram się maksymalnie krótko objaśnić składowe tej klasy. To ćwiczenie będzie trochę przydługie, ale później będzie nam już łatwiej. Wszystkie linie listingu wpisujemy w funkcję `WinMain()`.

Ćwiczenie 3.5.

Aby utworzyć główne okno aplikacji:

1. Pierwsza linia to deklaracja obiektu klasy `WNDCLASS`. Druga to ustawienie składowej `style`.

```
WNDCLASS glowne_okno;
glowne_okno.style = CS_HREDRAW | CS_VREDRAW;
```

   - `CS_HREDRAW` — oznacza, że okno będzie rysowane od nowa przy zmianie jego szerokości.
   - `CS_VREDRAW` — zapewnia rysowanie okna przy zmianie wysokości.

2. Składowa `lpfnWndProc` wskazuje na funkcję, która będzie przetwarzać komunikaty nadsyłane do okna. Zostaną one przesłane do funkcji (nazywanej tutaj `GlownaFunkcja`), a w niej umieścimy procedury do ich obsługi.

```
glowne_okno.lpfnWndProc = GlownaFunkcja;
```

   Pierwowzorem tej funkcji jest `WindowProc` (linii podanej poniżej nie wpisujemy do programu):

```
LRESULT CALLBACK WindowProc(HWND hwnd,UINT uMsg,WPARAM wParam,    LPARAM lParam);
```

   Komunikat przekazywany jest poprzez zmienną `uMsg` tej funkcji. W programie możemy nadać jej dowolną nazwę. Mimo, że nie deklarujemy tego jawnie, program będzie zakładał, że funkcja typu `LRESULT CALLBACK`, której adres znajduje się w zmiennej `lpfnWndProc`, ma takie same parametry jak `WindowProc`, a tylko inną nazwę. Definicje naszej funkcji zwrotnej wpiszemy zaraz po plikach nagłówkowych, przed `WinMain()`.

```
LRESULT CALLBACK GlownaFunkcja(HWND hwnd, UINT msge, WPARAM wParam, LPARAM lParam)
   {
    return DefWindowProc(hwnd, msge, wParam, lParam);
   }
```

   Na razie będzie ona zwracała wartość funkcji `DefWindowProc`. Jest to domyślna wartość zwracana przez funkcję zwrotną.

3. Przenosimy się z powrotem do wnętrza `WinMain()`. Składowa `cbClsExtra` zawiera liczbę dodatkowych bajtów dodanych do każdej klasy, a `cbWndExtra` liczbę dodatkowych bajtów dodanych do każdej działającej kopii okna. Obie te wartości ustawiamy na zero.

```
glowne_okno.cbClsExtra = glowne_okno.cbWndExtra = 0;
```

4. `hInstance` wskazuje na uchwyt do aplikacji, której częścią jest dane okno. W naszym przypadku uchwyt ten znajduje się w zmiennej `_hInstance`.

```
glowne_okno.hInstance = _hInstance;
```

5. Następna składowa określa ikonę widoczną w pasku aplikacji. Ta składowa musi być inicjalizowana uchwytem do ikony.

```
glowne_okno.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

Tutaj posłużymy się funkcją `LoadIcon`. Funkcja ta pobiera ikonę z pliku *exe*. Posiada ona dwa parametry. Pierwszy to uchwyt do aplikacji, która zawiera ikonę, a drugi to nazwa ikony. Istnieją ikony standardowe. Dla nich parametr pierwszy musi mieć wartość `NULL`. Nazwy ikon standardowych to:

`IDI_APPLICATION`	Domyślna ikona aplikacji
`IDI_ASTERISK`	Asterisk (informacje)
`IDI_EXCL`	Wykrzyknik (ostrzeżenia)
`IDI_HAND`	Ikona ręki (poważne ostrzeżenia)
`IDI_QUESTION`	Znak zapytania (zapytania)
`IDI_WINLOGO`	logo Windows

6. Zmienna `hCursor` identyfikuje kursor, jaki będzie się pojawiał, gdy wskaźnik znajdzie się w obszarze okna. Funkcja `LoadCursor` ma takie same parametry jak `LoadIcon`.

```
glowne_okno.hCursor = LoadCursor(NULL, IDC_ARROW);
```

Tu również występują standardowe kursory (pierwszy parametr `NULL`):

`IDC_APPSTARTING`	Strzałka i mała klepsydra
`IDC_ARROW`	Strzałka
`IDC_CROSS`	Krzyżyk
`IDC_IBEAM`	Kursor tekstowy
`IDC_ICON`	Pusta ikona (tylko Windows NT)
`IDC_NO`	Koło ukośnie przekreślone
`IDC_SIZE`	Strzałka w cztery strony (tylko Windows NT)
`IDC_SIZEALL`	To samo, co `IDC_SIZE`, ale dla innych systemów
`IDC_SIZENESW`, `IDC_SIZENS`, `IDC_SIZENWSE`, `IDC_SIZEWE`	Podwójne strzałki w różnych kierunkach
`IDC_UPARROW`	Pionowa strzałka
`IDC_WAIT`	Klepsydra

7. `hbrBackground` określa sposób wypełnienia tła okna. Inicjalizujemy go nazwą jednego z kolorów systemowych określonych dla składowych interfejsu systemów. Innymi słowy, możemy zażądać, aby okno miało kolor taki, jak np. paski okien, przyciski czy menu. W naszym przykładzie kolor jest ustawiony na standardowy kolor okna. Nazwy kolorów muszą być konwertowane na typ `HBRUSH`. Po zmianie schematu kolorów zmieni się też kolor tła naszego okna.

```
glowne_okno.hbrBackground = (HBRUSH)COLOR_WINDOW;
```

8. Składowa `lpszMenuName` wskazuje na menu stowarzyszone z oknem. Nasze okno na razie nie ma menu, więc parametr inicjalizujemy wartością `NULL`.

```
glowne_okno.lpszMenuName = NULL;
```

9. `lpszClassName` określa nazwę klasy okna. Jako nazwę przypiszemy łańcuch znakowy (tytuł), który oczywiście musimy uprzednio zdefiniować. Najpierw opuszczamy więc funkcję `WinMain()` i wpisujemy definicję zmiennej `tytul` bezpośrednio po nagłówku programu, przed funkcją `GlownaFunkcja`:

```
static const char tytul[] = "Okno glowne aplikacji";
```

Teraz wracamy do `WinMain()` i podstawiamy tytuł pod `lpszClassName`:

```
glowne_okno.lpszClassName = tytul;
```

Składowe klasy `WNDCLASS` mogą przybierać różne wartości, a jako wynik otrzymujemy odpowiedni wygląd i zachowanie okna. Wymienienie wszystkich możliwości przekracza zakres tej książki. Ograniczyłem się tylko do tych, które są w danej chwili potrzebne.

10. Po inicjalizacji składowych klasy należy ją zarejestrować za pomocą funkcji `RegisterClass`, a następnie (w przypadku udanej rejestracji) stworzyć okno oparte na naszej klasie.

Okno tworzymy za pomocą funkcji `CreateWindow`. W naszym przypadku użyjemy jej w postaci:

```
if (RegisterClass(&glowne_okno))
   {
   main_hwnd = CreateWindow(tytul, tytul,
               WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
               CW_USEDEFAULT, CW_USEDEFAULT,
               CW_USEDEFAULT, CW_USEDEFAULT,
               HWND_DESKTOP, NULL, _hInstance, NULL);
   }
 else
   main_hwnd = NULL;
```

Pierwszy parametr oznacza nazwę klasy reprezentującej okno. Nie jest to nazwa obiektu klasy, ale nazwa wpisana w zmienną `lpszClassName`. Drugi parametr to łańcuch znakowy, który będzie wyświetlany w pasku okna. Dla uproszczenia wpiszemy tu ten sam tekst, który oznacza nazwę klasy. Następny parametr określa sposób wyświetlania okna.

- `WS_OVERLAPPEDWINDOW` — wskazuje, że okno ma posiadać pasek tytułowy i ramkę.
- `WS_CLIPCHILDREN` — określa, że podczas malowania na oknie nie będzie zamazany obszar ewentualnych okien potomnych tego okna. Następne parametry oznaczają

współrzędne okna oraz jego szerokość i wysokość. Użyjemy tu zmiennej `CW_USEDEFAULT` przyporządkowującej tym zmiennym wartości standardowe zapisane w systemie.

Następny parametr określa uchwyt do okna stojącego wyżej w hierarchii. Ponieważ jest to pierwsze okno w aplikacji, funkcję okna nadrzędnego pełni pulpit. Dlatego podstawiamy tu `HWND_DESKTOP`.

Kolejny parametr ma dwa znaczenia. Dla okna posiadającego pasek i ramkę (tak jak nasze główne okno) oznacza menu stowarzyszone z oknem. Dla okna potomnego (ang. *child window*) oznacza uchwyt do tego okna. Wyjaśnię to na przykładach w dalszych podrozdziałach. Ponieważ nasze okno nie ma menu, parametr ten wynosi `NULL`. Przedostatnia zmienna zawiera uchwyt do aplikacji, której częścią jest okno. W naszym przypadku jest to zmienna `_hInstance`. Ostatni parametr to dodatkowe wartości przekazywane do okna. Na razie ustawimy go na `NULL`.

W przypadku powodzenia funkcja `CreateWindow` zwraca uchwyt do gotowego okna. W przypadku błędu zwracana jest wartość `NULL`.

11. Stworzenie okna nie jest równoznaczne z jego wyświetleniem. Aby to zrobić, należy użyć funkcji `ShowWindow`. Wyświetla ona okno o podanym uchwycie. Jako drugi parametr zawiera zmienne określające sposób wyświetlania. W przypadku, kiedy wyświetlamy pierwsze okno aplikacji, za parametr ten należy podstawić zmienną `nCmdShow` z funkcji `WinMain`. Następnie rysujemy wnętrze okna za pomocą `UpdateWindow`.

```
ShowWindow(main_hwnd, nCmdShow);
UpdateWindow(main_hwnd);
```

12. Skompilujmy i wykonajmy nasz program. Okno na chwilkę ukaże się i zniknie. Do poprawnego działania programu brakuje nam jeszcze obsługi komunikatów.

Teraz napiszemy główną pętlę programu, która będzie odbierała komunikaty nadsyłane przez system i rozsyłała je do funkcji zwrotnych, a później wypełnimy naszą funkcję `GlownaFunkcja` procedurami obsługi podstawowych komunikatów.

Ćwiczenie 3.6.

Przesyłanie i przetwarzanie komunikatów:

1. W dalszym ciągu `WinMain()` napiszmy:

```
while (GetMessage(&msg, NULL, 0, 0) > 0)
    {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
```

`GetMessage` przejmuje komunikaty i wpisuje je do zmiennej `msg`. Drugi parametr `NULL` oznacza, że chcemy przejmować wszystkie komunikaty wysyłane do danej aplikacji. Zamiast `NULL` możemy tu podstawić uchwyt do konkretnego okna, wtedy będą odbierane tylko komunikaty dla tego okna. `TranslateMessage` i `DispatchMessage` zajmują się rozsyłaniem komunikatów do odpowiednich funkcji zwrotnych dla poszczególnych okien. Zauważmy, że pętla będzie działała, dopóki funkcja

`GetMessage` nie zwróci wartości zero. Stanie się to wtedy, jeżeli przechwyci ona komunikat `WM_QUIT`, który oznacza zakończenie aplikacji. Tutaj kończymy funkcję `WinMain`.

2. Pozostało nam jeszcze napisanie funkcji przetwarzającej komunikaty. Na razie będzie ona obsługiwała `WM_PAINT` `WM_CLOSE` i `WM_DESTROY`.

```
LRESULT CALLBACK GlownaFunkcja(HWND hwnd, UINT msge, WPARAM wParam, LPARAM lParam)
   {

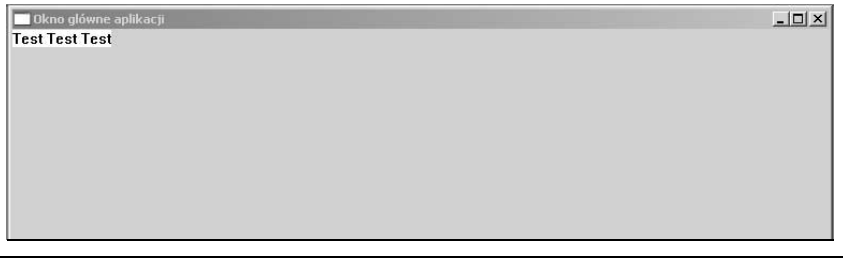
   switch (msge)
     {
     case WM_PAINT :
       {
       PAINTSTRUCT ps;

       BeginPaint(hwnd, &ps);
       TextOut(ps.hdc, 0, 0, "Test Test Test", lstrlen("Test Test Test"));
       EndPaint(hwnd, &ps);
       return 0;
       }
     case WM_CLOSE : { DestroyWindow(hwnd); break; }
     case WM_DESTROY : { PostQuitMessage(0); return 0; }
     }
   return DefWindowProc(hwnd, msge, wParam, lParam);
   }
```

Instrukcja `switch` wywołuje odpowiednią procedurę w zależności od treści komunikatu. Komunikat `WM_PAINT` wysyłany jest przez system w przypadku, kiedy zachodzi konieczność odświeżenia okna, na przykład w chwili, gdy użytkownik zmieni położenie okna lub jego wymiary. Funkcja `BeginPaint` przygotowuje okno do powtórnego narysowania i wypełnia strukturę `ps`, która określa obszar okna. `TextOut` wypisuje w oknie podany ciąg znakowy. Parametrami tej funkcji są kolejno: odnośnik do struktury `ps` określającej granice okna, współrzędne tekstu w oknie, łańcuch znakowy zawierający tekst i długość tego łańcucha. `EndPaint` kończy malowanie okna.

Komunikat `WM_CLOSE` jest wysyłany w przypadku, kiedy użytkownik wydał komendę zamknięcia okna. Wówczas uruchamiana jest funkcja `DestroyWindow`, która kasuje wskazane okno. Funkcja ta wysyła także komunikat `WM_DESTROY`. Ten z kolei aktywuje funkcję `PostQuitMessage`, która wysyła komunikat `WM_QUIT`, który z kolei kończy pętlę przechwytywania wiadomości.

3. Teraz możemy już skompilować i uruchomić nasz program. Ukaże się okno takie jak poniżej (rysunek 3.2).



Rysunek 3.2. *Główne okno aplikacji*

Główne okno aplikacji będzie bazą, którą wykorzystamy do tworzenia aplikacji. W następnym podrozdziale stworzymy w naszej aplikacji kontrolki. Są to elementy umożliwiające komunikację użytkownika z aplikacją. Należą do nich przyciski, pola tekstowe itp.