Wprowadzenie do tworzenia aplikacji okienkowych przy pomocy interfejsu Windows API

Zbigniew Koza
Instytut Fizyki Teoretycznej Uniwersytetu Wrocławskiego
(zkoza@ift.uni.wroc.pl)

Najnowsza wersja tego skryptu znajduje się na stronie
`http://www.ift.uni.wroc.pl/fkomp`

WROCŁAW 2002

Spis treści

1. Pierwszy program dla Windows ... 2

Program tradycyjny ... 2

Pierwsze okienko – standardowe ... 2

Funkcje API i nowe typy danych ... 3

Funkcja WinMain ... 4

Funkcja MessageBox ... 5

Inne predefiniowane okienka dialogowe ... 6

Podsumowanie ... 8

Programy przykładowe ... 8

Zadania ... 9

2. Definiujemy własne okno ... 10

Wstęp ... 10

Program „Moje okno” ... 13

Opis działania programu ... 14

Konstrukcja funkcji WinMain ... 15
Rejestracja klasy okien ... 15
Tworzenie i wyświetlanie okna ... 16
Pętla komunikatów ... 17

Procedura okna ... 17
Konstrukcja procedury okna ... 17
Domyślna procedura okna ... 18
Komunikat WM_PAINT ... 18
Komunikat WM_DESTROY i zakończenie programu ... 20

Standardowe klasy okien ... 20

Podsumowanie ... 21

Programy przykładowe ... 21

Zadania ... 22

3. Interfejs GDI ... 23

Kontekst urządzenia ... 24
Pobieranie kontekstu urządzenia ... 24
Zwalnianie kontekstu urządzenia ... 24
Pierwszy rysunek ... 25

Pióra, pędzle, czcionki… ... 29
Pióra ... 29
Inne obiekty interfejsu GDI używane w kontekstach urządzenia ... 30

Programy przykładowe ... 31

Podsumowanie ... 31

Zadania ... 33

4. Obsługa komunikatów użytkownika ... 34

Prosta aplikacja interaktywna – gra „snake” ... 34
Specyfikacja gry ... 34
Specyfikacja interfejsu użytkownika ... 35

Specyfikacja stanu programu, stałych globalnych i nowych typów zmiennych ... 35
Rozruch programu ... 37
Specyfikacja obsługiwanych komunikatów i implementacja procedury okna ... 38

Programy przykładowe ... 41

Podsumowanie ... 42

Zadania ... 42

Dodatek A Kompilacja programów napisanych w Windows API ... 43

Zintegrowane środowiska programistyczne ... 43
Środowisko MS Visual C++ 6.0 ... 43
Środowisko edytora Dev C++ ... 43

Kompilacja z poziomu wiersza poleceń ... 44

Indeks ... 46

1. Pierwszy program dla Windows

Program tradycyjny

Zgodnie z niepisaną tradycją, pierwszy program w nowym języku programowania powinien wypisywać na ekranie napis „Hello, world”, który w spolszczonej[1] wersji przyjmuje postać „Ahoj, przygodo!”. W języku C++ taki najprostszy program wyglądać może następująco:

```
#include<iostream.h>
void main()
{
   cout << „Ahoj, przygodo!\n”;
}
```

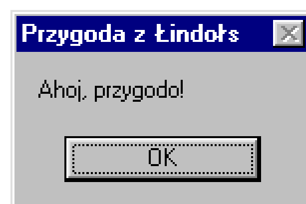
Podstawową cechą powyższego rozwiązania jest to, że napis wyświetlany jest na *standardowym urządzeniu wyjścia* (`cout`), określanym przez system operacyjny, który nadzoruje wykonanie programu. W środowisku systemu operacyjnego UNIX urządzeniem tym jest np. konsola tekstowa lub okienko terminala X. Jednak w przypadku systemu Windows sprawa nie jest tak prosta, gdyż z założenia jest on systemem graficznym, w którym programy uruchamia się nie z konsoli, która mogłaby służyć jako standardowe urządzenie wyjścia, lecz przez „przyciśnięcie” myszką odpowiedniej ikonki lub przycisku na ekranie. Dlatego, aby uruchomić powyższy program, Windows utworzy specjalne okno pełniące rolę „standardowego urządzenia wyjścia”. Okienko jest brzydkie i niefunkcjonalne – jakby jego twórcy celowo usiłowali zniechęcić do niego programistów i użytkowników.

Nie mamy wyboru – musimy przepisać nasz program tak, aby wpasował się w filozofię Windows. Oto nasza pierwsza próba – program ahoj1.cpp.

Pierwsze okienko – standardowe

```
/* ahoj.cpp - wersja dla Windows ze standardowym okienkiem dialogowym */

#include <windows.h>
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR szCmdLine, int iCmdShow)
{
  MessageBoxEx(NULL, „Ahoj, przygodo!”, „Przygoda z Łindołs”,0,0);
  return 0;
}
```

Zanim ten program uruchomimy, musimy go poddać kompilacji. Programy dla Windows kompiluje się troszkę inaczej niż standardowe programy w C/C++. Zagadnienie to omawiam w dodatku A. Efektem wykonania powyższego programu będzie wyświetlenie na ekranie następującego okienka:



Rysunek 1. Okienko *Ahoj, przygodo!*

Na pierwszy rzut oka program ten niemal w ogóle nie przypomina poprzedniej, klasycznej wersji programu „Ahoj, przygodo!”. A przecież oba programy napisano w języku C++!

[1] czyli… nie zawierającej specyficznie polskich liter!

Funkcje API i nowe typy danych

Pierwszą rzeczą, której w programie dla Windows będzie poszukiwał programista używający języka C/C++, jest funkcja `main()`, od której jego zdaniem powinno rozpoczynać się wykonanie programu. I tu niespodzianka – *zamiast* niej programy dla Windows wykorzystują funkcję **`WinMain`**. Należy ona do zestawu ponad tysiąca (!) funkcji wchodzących w skład tzw. interfejsu programowania aplikacji (API, *Application Programming Interface*), czyli zestawu funkcji, których podczas pisania programu można używać w celu komunikowania się z systemem operacyjnym. Programowanie dla Windows to w dużym stopniu sztuka używania funkcji API (nawet jeżeli wykorzystuje się pewne techniki ukrywające API, np. klasy MFC). Ich deklaracje dostępne są poprzez standardowy plik nagłówkowy `windows.h`. Dołączyliśmy go do naszego programu w pierwszej linijce dyrektywą `#include`. Ze względu na rangę funkcji API, ich nazwy wyróżniamy tu kolorem zielonym.

Innym aspektem powyższego programu, który z pewnością zwróci uwagę nowicjusza, jest użycie dość sporej, jak na tak krótki program, ilości tajemniczo brzmiących identyfikatorów określających typy zmiennych, np. **`HINSTANCE`** lub **`LPSTR`**. Zgodnie z konwencją dotyczącą używania w językach C i C++ identyfikatorów składających się z dużych liter, zostały one zdefiniowane jako stałe symboliczne poprzez odpowiednią dyrektywę preprocesora (`#define`) lub instrukcję `typedef`. Ich wprowadzenie wiąże się z próbą zapanowania nad złożonością programów pisanych dla środowiska Windows oraz zapewnienia im przenośności, np. przy przejściu z wersji 16- do 32-bitowej lub z 8-bitowego standardu kodowania znaków (ASCII) do standardu 16-bitowego (Unicode). Większość z tych nowych typów (a są ich dziesiątki), równoważna jest prostym typom języka C. Na przykład **`HINSTANCE`** to `void*`, a **`LPSTR`** to `char*`. Jednak odwzorowanie to może ulec zmianie w przyszłych wersjach systemu.

Windows ukrywa swoje wewnętrzne dane w „prywatnych" strukturach, do których użytkownik uzyskuje dostęp poprzez wskaźniki typu `void*`. Zwie się je „uchwytami" (ang. *handles*). Dostęp do nich możliwy jest *wyłącznie* poprzez funkcje interfejsu API. Sposób ten przypomina hermetyzację danych w obiektach języka C++, zaimplementowany jest jednak środkami języka C – stąd tak powszechne w funkcjach API użycie niemal zupełnie niepotrzebnego w C++ typu `void*`. Jednak posługiwanie się „nagimi" wskaźnikami typu `void*` niesie za sobą duże niebezpieczeństwo ich niewłaściwego użycia. Prędzej czy później funkcji oczekującej uchwytu do okna przekażemy np. uchwyt do czcionki. Błędu tego jednak kompilator nie zauważy, gdyż z jego punktu widzenia oba uchwyty są równie dobre – mają przecież ten sam typ `void*`. Aby uporać się z tym problemem kompilatory programów dla Windows umożliwiają zdefiniowanie specjalnej stałej symbolicznej **`STRICT`** (np. instrukcją `#define STRICT`). Jeżeli zdefiniujemy ją *przed* włączeniem do programu pliku `windows.h`, kompilator zacznie rozróżniać ponad 40 różnych uchwytów! Najlepiej stałą `STRICT` zdefiniować globalnie dla całego programu. W środowisku kompilatorów VC++ można tego dokonać wybierając z menu opcję *Build.Settings.C/C++* i dopisując w polu *Preprocesor_Definitions* wyraz `STRICT`. Ze względu na ważną rolę pełnioną przez identyfikatory typów charakterystycznych dla programów pisanych dla Windows, wyróżniamy je tu kolorem niebieskim.

Mimo wykorzystywania przez Windows kilkudziesięciu różnych identyfikatorów typów, istnieje kilka sposobów, by sobie z nimi poradzić. Po pierwsze, część nowych typów to skrócone nazwy typów standardowych, np. **`UINT`** to `unsigned int`. Inne nazwy stosują się do tzw. notacji węgierskiej, w której typy zmiennych identyfikuje się poprzez pierwsze litery ich nazw, m.in.:

- Nazwy typów związanych z uchwytami rozpoczynają się od litery H, np. **`HINSTANCE`**.
- Litera P lub litery LP umieszczone na początku nazwy oznaczają wskaźnik (ang. ***pointer***) np. **`LPINT`** to `int*`. Pierwotnie litera L oznaczała „long", lecz takie rozróżnienie miało sens tylko w systemach 16-bitowych.
- Analogicznie litery PC lub LPC oznaczają wskaźnik na stałą, np. **`LPCSTR`** to `const char*`.
- Nazwy zawierające ciąg STR odnoszą się do tablicy znaków zakończonych bajtem zerowym.
- Litera W oznacza platformę Unicode, litera T – niezależność od platformy (ANSI lub Unicode), np. **`LPCTSTR`** to adres początku stałej tablicy znaków na aktualnie obowiązującej platformie.

Analogiczne konwencje stosowane są przy doborze nazw zmiennych; np.

- Litery „dw” oznaczają „double word”, co w praktyce oznacza zmienną typu `long int`.
- Nazwy rozpoczynające się literą „n” zarezerwowane są dla zmiennych typu `int`.

Po drugie, część nowych typów jest ściśle związana z argumentami lub wartościami pewnych funkcji (lub klas funkcji). Na przykład **`LRESULT`** to typ wartości zwracanych przez pewną rodzinę funkcji definiowanych przez programistę zwanych *procedurami okna*. Co prawda aktualnie jest on równoważny typowi `int`, ale używając w deklaracjach procedur okna napisu **`LRESULT`** zamiast `int`, zwiększamy prawdopodobieństwo, że nasz program będzie zgodny z przyszłymi wersjami Windows.

Funkcja WinMain

Jak widzimy, funkcja **`WinMain`** posiada aż cztery obowiązkowe argumenty:

- **`HINSTANCE`** *`hInstance`*,
- **`HINSTANCE`** `hPrevInstance`,
- **`LPSTR`** `szCmdLine`,
- `int iCmdShow`

Pierwszy argument, `hInstance`, jest tzw. uchwytem wystąpienia programu W systemie wielozadaniowym (a więc i w Windows) jednocześnie można uruchomić wiele kopii tego samego programu. Kopie takie (czyli obrazy programu w pamięci operacyjnej) nazywamy *wystąpieniami* lub *egzemplarzami* (ang. *instances*) programu; słowo *program* oznacza zaś kod wykonywalny przechowywany na dysku. Tak więc jeden program może mieć wiele wystąpień; teoretycznie każde z nich będzie posiadało inny uchwyt `hInstance`. Jednak w praktyce numer wystąpienia miał znaczenie w starszych wersjach systemu Windows, a obecnie wszystkie programy otrzymują ten sam uchwyt: `0x00400000`. Jednak wiele funkcji systemu Windows wciąż wymaga podawania tego parametru, dlatego gdzieś trzeba go zapisać i posłusznie, w razie potrzeby, przekazywać go tam, gdzie jest potrzebny.

Drugi parametr funkcji **`WinMain`** również jest pozostałością po 16-bitowych wersjach Windows, zawsze ma wartość `0` i obecnie *nie jest wykorzystywany*.

Trzeci parametr, `szCmdLine`, zawiera adres łańcucha znaków, w którym przechowywane są argumenty wywołania danego programu. Jego typ określa się jako **`LPSTR`**, który obecnie równoważny jest typowi `char*`. Z kolei ostatni argument, `iCmdShow`, informuje program o sposobie, w jaki, zdaniem użytkownika programu, powinno ukazać się główne okno aplikacji tuż po jej uruchomieniu. Na przykład czy ma zajmować cały ekran, czy też powinna pojawić się jako ikonka na pasku zadań. Ponieważ w naszym programie nie tworzymy *własnego* okna, informacje przekazywane przez ten argument możemy zignorować.

Zauważmy, że funkcja **`WinMain`** zwraca do systemu liczbę całkowitą. Zgodnie z dokumentacją, jeżeli program nie tworzy własnego okienka, wartością tą powinno być 0, co wyjaśnia postać ostatniej instrukcji naszego programu. Ale uważny czytelnik zwrócił zapewne uwagę na występujący w deklaracji funkcji **`WinMain`** tajemniczy identyfikator **`WINAPI`**. Oznacza on, że funkcja **`WinMain`** nie jest „zwyczajną” funkcją języka C lub C++, lecz że w specjalny sposób komunikuje się z systemem operacyjnym. W największym uproszczeniu pojawienie się tego identyfikatora można wyjaśnić następująco. System operacyjny może być napisany w dowolnym, a priori nie znanym języku programowania i powinien współpracować z programami napisanymi w różnych językach, a nie tylko w C lub C++. Kompilatory różnych języków stosują różne metody implementacji funkcji – różnice mogą dotyczyć np. kierunku opracowywania ich argumentów (w lewo lub w prawo) lub sposobu zarządzania tzw. stosem funkcji. Identyfikator **`WINAPI`** oznacza więc: „ta funkcja stosuje konwencję obowiązującą w systemie operacyjnym”. Wszystkie funkcje interfejsu API (a więc także użyta w naszym programie funkcja **`MessageBoxEx`**) są typu **`WINAPI`**. Programista powinien deklarować typy *wszystkich* pisanych przez siebie funkcji, które mogą być wywoływane *bezpośrednio* przez system operacyjny, jako **`WINAPI`** lub, w nieco innym kontekście, **`CALLBACK`**. Zwróćmy uwagę na interesujący szczegół: w programach tradycyjnych (nie przeznaczonych dla systemu Windows) system operacyjny może (a nawet musi) bezpośrednio wywołać jedynie funkcję `main()`.

Funkcja MessageBox

Przejdźmy do omówienia funkcji **MessageBoxEx**. Jest to typowa funkcja API i dobrze ilustruje zasady wykorzystywania większości funkcji należących do tego zestawu. Zgodnie z przedstawioną poniżej deklaracją, posiada ona 5 argumentów (jak na funkcję API jest to i tak dość mało):

```
int MessageBoxEx(
HWND hWnd,              // uchwyt okna będącego „właścicielem" („rodzicem") tworzonego właśnie okna
LPCTSTR lpText,         // adres tekstu wyświetlanego w oknie
LPCTSTR lpCaption,      // tytuł tworzonego okna
UINT uType,             // styl tworzonego okna
WORD wLanguageId        // język, w którym wyświetlane będą napisy na przyciskach
);
```

Pierwszy argument identyfikuje okno nadrzędne („rodzica"), które tworzy dane okienko. Ponieważ w naszym programie nie tworzymy własnego okna, które mogłoby „ojcować" okienku tworzonemu funkcją **MessageBox**, wstawiamy tu `NULL`. Odpowiada to następującej konwencji: jeżeli pewna funkcja wymaga podania uchwytu okna nadrzędnego, wartość `NULL` oznacza brak takiego okna.

Drugi argument (typu **LPCTSCR**, czyli `const char[]`) służy do przekazania tekstu, który ma być wyświetlony w okienku. Natomiast trzeci argument określa tytuł okienka. Są to zwyczajne napisy, które, jak widzimy w naszym przykładzie, mogą zawierać polskie litery.

Bardzo ciekawy, i jakże charakterystyczny dla funkcji interfejsu API, jest czwarty argument, który definiuje styl okienka poprzez liczbę całkowitą bez znaku (**UINT**, czyli `unsigned int`). Jest on kombinacją flag (bitów) określających właściwości okienka. Flagi łączymy tradycyjnie – przy pomocy operatora sumy bitowej. Poszczególnym flagom odpowiadają identyfikatory zdefiniowane w pliku `<windows.h>`. Na przykład kombinacja

```
MB_OKCANCEL | MB_ICONQUESTION | MB_HELP
```

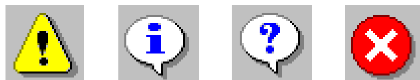
powoduje wyświetlenie w okienku dwóch przycisków: jednego z napisem `OK`, drugiego z napisem `Cancel` (w wersji polskiej: `Anuluj`); dodatkowo wyświetlona zostanie ikonka ze znakiem zapytania oraz klawisz z napisem `Help (Pomoc)`, który będzie również reagował na naciśnięcie klawisza `F1` uruchomieniem systemu pomocy (o ile takowy został dołączony do programu). Przyciski można definiować przy pomocy jednej z następujących flag:

- **MB_ABORTRETRYIGNORE**
- **MB_OKCANCEL**
- **MB_RETRYCANCEL**
- **MB_YESNO**
- **MB_YESNOCANCEL**

Z kolei standardowe ikonki można wyświetlić używając jednej z 8 flag:

- **MB_ICONEXCLAMATION**
- **MB_ICONWARNING**
- **MB_ICONINFORMATION**
- **MB_ICONASTERISK**
- **MB_ICONQUESTION**
- **MB_ICONSTOP**
- **MB_ICONERROR**
- **MB_ICONHAN**

Mimo że flag jest osiem, w moim systemie wyświetlane są tylko cztery różne typy ikon:



Rysunek 2. Ikony dostępne w funkcji `MessageBox`.

Istnieją jeszcze inne flagi (w sumie jest ich obecnie 28), określające dodatkowe właściwości okienka tworzonego przez **MessageBoxEx**; nie będziemy ich tu jednak dokładne omawiać. Ich istnienie ilustruje jednak jeden z podstawowych problemów związanych z programowaniem dla Windows: nie dość, że mamy do dyspozycji kilkaset funkcji z biblioteki API, to jeszcze większość z nich wymaga posługiwania się argumentami o specjalnej postaci, np. flagami. Jak więc można poradzić sobie z tak

ogromną ilością informacji, skoro nie ma mowy, byśmy byli w stanie je wszystkie zapamiętać? Poza kilkoma książkami np. kolejnymi wydaniami podręcznika Ch. Petzolda „Programowanie Windows”, doskonałym źródłem informacji są systemy pomocy dostarczane wraz z kompilatorami. Na przykład system pomocy kompilatora Visual C++ w wersji 6.0 oferuje książki obejmujące (w wersji skompresowanej) ponad 100 milionów znaków, co odpowiada grubo ponad 50 tysiącom stronom maszynopisu. Pomoc ta obejmuje m.in. opis kilku języków programowania (w tym C i C++), informacje o funkcjach API, użytecznych makrodefinicjach preprocesora oraz opis metod pisania programów dla Windows. Dostępnych jest także kilkadziesiąt programów przykładowych ilustrujących możliwości i sposób wykorzystywania funkcji z zestawu API. Bliższe omówienie systemu pomocy kompilatora Visual C++ 6.0 znajduje się w dodatku ***.

Zauważmy na koniec, że zgodnie ze swoją deklaracją, funkcja **MessageBoxEx** zwraca liczbę całkowitą. Służy ona do identyfikacji przycisku naciśniętego przez użytkownika. Może być równa jednej z siedmiu wartości: `IDABORT`, `IDCANCEL`, `IDIGNORE`, `IDNO`, `IDOK`, `IDRETRY` lub `IDYES`. Jak łatwo się domyślić, odpowiadają one przyciśnięciu odpowiednio klawisza ABORT, CANCEL, IGNORE, NO, OK, RETRY lub YES.

Inne predefiniowane okienka dialogowe

W systemie Windows oprócz funkcji `MessageBoxEx` dostępnych jest kilkanaście innych funkcji wyświetlających dobrze znane każdemu użytkownikowi okienka dialogowe. Są to:

`ChooseColor`, `ChooseFont`, `FindText`, `GetFileTitle`, `GetOpenFileName`, `GetSaveFileName`, `PageSetupDlg`, `PrintDlg` i `PrintDlgEx`.

Poniżej przedstawiam jedynie prosty sposób wykorzystania okienka „Otwórz plik”. Program otwiera okienko dialogowe „Otwórz plik”, w którym zaprogramowałem domyślny katalog początkowy, domyślną nazwę pliku i zestaw trzech filtrów nazw plików (w formacie HTML, PS lub PDF) wraz z ich krótkim opisem, który pojawiać się będzie w polu wyboru formatu pliku.

Posługiwanie się tymi funkcjami jest niestety nieco trudniejsze niż w przypadku `MessageBoxEx` – wszystkie one wymagają bowiem przekazania im adresu pewnej rozbudowanej struktury danych (innej dla każdego typu okienek), którą przed ich wywołaniem należy starannie wypełnić odpowiednimi danymi.

Oto kod programu:

```
#include <windows.h>
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
      // przygotowujemy sie do otworzenia okienka dialogowego „wybierz plik"
      // najpierw definiujemy kilka napisow, które pojawia się w okienku
      // w poniższej tablicy przekażemy początkową nazwę pliku i otrzymamy nazwę pliku wybranego przez użytkownika
      const int rozmiar_bufora = 512;
      char argument_programu [rozmiar_bufora] = „ index.html";
      // Filtr określa sposób filtrowania rozszerzeń plików w okienku dialogowym;
      // poszczególne opcje oddzielamy znakami '\0'. Opcje umieszczamy parami (tekst, filtr rozszerzenia).
      // Na końcu umieszczamy '\0'.
      const char filtr[] =
         „pliki HTML\0*.html\0Post Script\0*.ps\0Pliki PDF\0*.pdf\0";

      // definiujemy strukturę typu OPENFILENAME i inicjujemy ją zerami
      OPENFILENAME ofn = {0};

      // a teraz wypełniamy te składowe tej struktury, które mają być niezerowe:
      ofn.lStructSize = sizeof(OPENFILENAME);        // obowiązkowa instrukcja
      ofn.lpstrFilter = filtr;                       // instalacja filtra plików
      ofn.lpstrFile = argument_programu + 1;         // tu bedzie wpisana nazwa pliku, po spacji!
      ofn.nMaxFile = rozmiar_bufora - 1;
      ofn.lpstrInitialDir = "d:\\www";               // katalog poczatkowy
      ofn.nFilterIndex = 1;                          // tu: 1 = HTML, 2 = PS, 3 = PDF
      ofn.lpstrTitle = "Otwórz plik HTML, PS lub PDF"; // tytul okienka
      if ( !GetOpenFileName (&ofn) )                 // otwieramy okienko „Otworz"
         return 2;
      // przygotowujemy sie do otworzenia procesu. (czyli programu)...
      STARTUPINFO si = {0};
      si.cb = sizeof(si);
      PROCESS_INFORMATION pi;
      // tablica nazwy_programow przechowuje pełne nazwy 3 programów zainstalowanych w moim komputerze
      const char* nazwy_programow[3] =
      {
         "c:\\program files\\netscape\\netscape 6\\netscp6.exe",
         "c:\\ghostgum\\gsview\\gsview32.exe",
         "c:\\program files\\adobe\\acrobat 5.0\\reader\\acrord32.exe"
      };

      BOOL ok2 = CreateProcess (
         nazwy_programow[ofn.nFilterIndex - 1],  //nazwa programu
         argument_programu,
         0, 0, 0, 0, 0, 0, &si, &pi);
      if (!ok2)
      {
          char bufor[256];
          wsprintf(bufor, „próba otworzenia pliku '%s' nie powiodła sie!\0",
            nazwy_programow[ofn.nFilterIndex-1]);
          MessageBoxEx( NULL, bufor, nazwa_okna2,
            MB_OK | MB_ICONINFORMATION, 0 );
       }
       return 0;
}
```

Szczegółowe informacje o stosowaniu standardowych okienek dialogowych można znaleźć w systemie pomocy *online* kompilatora (w indeksie należy wyszukać hasło „Common Dialog Box Library"); analogicznie można znaleźć informacje o uruchamianiu zewnętrznych aplikacji poprzez wywołanie funkcji `CreateProcess` i manipulowaniu łańcuchami znaków funkcją `wsprintf`.

Podsumowanie

Pisząc programy dla Windows należy pamiętać, że:

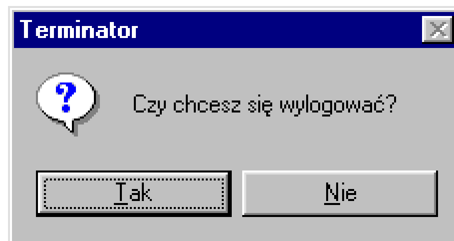
- Rolę funkcji `main` przejmuje funkcja `WinMain`.
- Wszystkie funkcje wywoływane z poziomu systemu operacyjnego, a więc i `WinMain`**,** muszą być deklarowane z modyfikatorem `WINAPI` lub `CALLBACK`.
- Nie wykorzystuje się standardowego wejścia i wyjścia, tj. funkcji `printf`, `scanf` (język C) i obiektów `cout`, `cin` (język C++).
- Szeroko wykorzystuje się zestaw ponad tysiąca funkcji wchodzących w skład tzw. interfejsu API. Implikuje to *konieczność* posiadania dokumentacji interfejsu API – w wersji papierowej (np. podręcznik Ch. Petzolda) lub elektronicznej (np. system pomocy *online* kompilatora Visual C++).
- W wielu przypadkach argumenty funkcji należących do interfejsu API interpretowane są jako zestaw flag o specyficznym znaczeniu.
- Większość deklaracji funkcji (np. interfejsu API) i makrodefinicji (np. flagi) związanych z programowaniem dla Windows jest do programu dołączana poprzez standardowy plik nagłówkowy `<windows.h>`. Dlatego instrukcja `#include <windows.h>` jest jedną z pierwszych instrukcji każdego programu dla Windows.
- Przed kompilacją programów dla Windows pamiętajmy o zdefiniowaniu stałej symbolicznej `STRICT`. Oszczędzi nam to wiele pracy przy odpluskwianiu programu.
- Powtórzmy: najpełniejsze informacje o własnościach funkcji należących do interfejsu API można uzyskać z systemu pomocy kontekstowej kompilatorów. Wymaga to niestety dobrej znajomości języka angielskiego…

Programy przykładowe

1. `0MessageBox` to minimalny program wyświetlający okienko informacyjne.
2. `1LogOut` to rozwiązanie przedstawionego poniżej zadania 1.
3. `2CreateProcess` to prosty program uruchamiający warunkowo uruchamiający zewnętrzną aplikację (tu: klienta poczty).
4. `3GetOpenFile` to program ilustrujący praktyczne wykorzystania okna dialogowego „otwórz plik”.

Zadania

1. Napisz program, który wyświetli poniższe okienko, po czym zakończy sesję użytkownika w przypadku naciśnięcia przycisku Tak. Do wylogowania użytkownika można wykorzystać funkcję **ExitWindowsEx** (EWX_LOGOFF, 0).



Rysunek 3. Proste okienko otrzymane poprzez wywołanie funkcji MessageBox.

2. Wykorzystując system pomocy kompilatora sprawdź, jakie znaczenie mają poszczególne argumenty funkcji **ExitWindowsEx**.
3. W powyższej dyskusji nie omówiliśmy znaczenia ostatniego argumentu funkcji **MessageBoxEx**, czyli wLanguageId. Wykorzystując system pomocy kompilatora sprawdź jego znaczenie i odpowiedz, dlaczego w zdecydowanej większości przypadków można założyć, że równy jest 0?
4. „Pobaw się” programami przykładowymi – np. pozmieniaj napisy, usuń niektóre instrukcje itp. – i zobacz, jaki będzie efekt Twoich działań.
5. Spróbuj wyświetlić jakiekolwiek inne okienko dialogowe, np. „wybierz kolor” lub „wybierz czcionkę”. Nie musisz z tym okienkiem niczego robić – wystarczy, że w ogóle uda ci się takie okienko wyświetlić!

2. Definiujemy własne okno

Wstęp

Tradycyjny program w języku C wykonuje się od początku do końca w sposób ciągły, a występujące w nim sporadycznie momenty „bezczynności" wiążą się zazwyczaj z oczekiwaniem na wprowadzenie przez użytkownika danych z klawiatury. Sytuacje, w których użytkownik ma jakikolwiek wpływ na działanie programu w trakcie jego wykonywania, są więc ściśle zaprogramowane – analizując kod źródłowy, można dokładnie przewidzieć, kiedy takie sytuacje będą miały miejsce, sam zaś użytkownik nie może się uchylić od dostarczenia programowi odpowiednich danych dokładnie wtedy, kiedy zostanie o nie poproszony. Programy takie składają się więc zasadniczo z trzech części:

1. wczytanie wartości początkowych oraz inicjalizacja zmiennych;
2. wykonanie obliczeń;
3. zapisanie wyników i zakończenie programu.

Ten styl programowania dominuje m.in. w zastosowaniach inżynierskich, w których komputer służy do rozwiązania konkretnego problemu numerycznego, a użytkownika w zupełności zadowoli wynik przedstawiony w postaci kilku liczb.

W przypadku programów komunikujących się z użytkownikiem poprzez interfejs graficzny sytuacja ulega radykalnej zmianie. Użytkownikowi wydaje się bowiem, że komputer kreuje na ekranie wirtualną rzeczywistość biegnącą tym samym rytmem, co czas rzeczywisty. Dlatego, po pierwsze, użytkownik oczekuje, że *w każdej* chwili powinien mieć wpływ na działanie programu, przy czym reakcja komputera na polecenia wydane np. przy pomocy myszki powinna być *natychmiastowa*. Na przykład program wyświetlający stronę WWW musi być w każdej chwili przygotowany na to, że użytkownik, po obejrzeniu zaledwie 10% jej zawartości zrezygnuje z obejrzenia reszty i zażąda wyświetlenia zupełnie nowej strony. Lub powiększy do maksimum rozmiar okna przeglądarki, oczekując jednocześnie, że wyświetlany na ekranie obraz natychmiast się do tych nowych rozmiarów dostosuje. Czy oznacza to, że pisząc kod takiej przeglądarki musimy co kilkadziesiąt instrukcji sprawdzać stan klawiatury, myszy i portów zewnętrznych? A jak sobie poradzić z komunikacją między różnymi programami (np. jak spowodować, by można było przy pomocy przeciągnięcia myszą fragmentu tekstu skopiować go z jednego edytora tekstu do drugiego). Ponadto, jak uczy doświadczenie, programy działające w wielozadaniowych graficznych systemach operacyjnych większą część czasu trwają w stanie uśpienia. Na przykład przeglądarka WWW po załadowaniu i wyświetleniu bieżącej strony przechodzi w stan oczekiwania na dalsze instrukcje użytkownika. Czy i taki uśpiony proces miałby co chwilę sam sprawdzać, czy aby nie ma czegoś nowego do zrobienia? Czyż nie powinien raczej dążyć do tego, by w jak najmniejszym stopniu obciążać system, umożliwiając w ten sposób efektywniejszą realizację procesów w danej chwili nie uśpionych?

Aby sprostać tym wyzwaniom program dla Windows wykonywany jest w zupełnie inny sposób niż opisane powyżej programy „tradycyjne". Cykl jego realizacji można opisać następująco:

1. Inicjalizacja (nadanie wartości początkowych zmiennym, wyświetlenie okien),
2. Czekanie w uśpieniu na kolejną komendę (tzw. komunikat),
3. Przetworzenie otrzymanego komunikatu; powrót do punktu 2 lub zakończenie programu.

O ile więc tradycyjny program kończy swoje działanie po wykonaniu wszystkich przewidzianych przez programistę obliczeń, zakończenie programu dla Windows następuje jako reakcja na rozkaz (komunikat) „zakończ działanie". Dopóki użytkownik nie wyda takiego polecenia, program dla Windows może pracować w nieskończoność. Komunikaty docierają do programu poprzez system operacyjny, który m.in. śledzi stan urządzeń zewnętrznych i na bieżąco decyduje, do którego procesu i w jakiej kolejności mają trafiać informacje wygenerowane np. przez mysz lub klawiaturę.

Programowanie dla Windows jest więc z natury **programowaniem defensywnym**: programista musi założyć, że jego program w dowolnym momencie może zostać zasypany gradem różnych komunikatów, które należałoby obsłużyć błyskawicznie, w czasie niezauważalnym dla użytkownika; w prze-

ciwnej sytuacji użytkownik mógłby bowiem odnieść wrażenie, że program się zawiesił. Na szczęście programista nie musi pisać kodu obsługi wszystkich możliwych komunikatów[2]. Przetwarzanie tych spośród nich, których nie chce obsługiwać, zlecić bowiem może specjalnej funkcji systemowej **DefWindowProc**.

W poprzednim rozdziale wypisaliśmy na ekranie napis „Ahoj, przygodo" posługując się funkcją **MessageBoxEx**. Jej możliwości są jednak bardzo ograniczone, a najczęściej wykorzystywana jest ona do zasygnalizowania użytkownikowi sytuacji awaryjnej lub wypisania prostego komunikatu. Stosując ją, nie mamy praktycznie żadnej możliwości formatowania tekstu, wyświetlania własnej grafiki, pasków przewijania, menu i wielu innych elementów graficznego interfejsu użytkownika znanych z aplikacji działajacych w systemie Windows.

Jednakże zaprojektowanie i wyświetlenie *własnego* okna aplikacji wymaga od programisty pewnego nakładu pracy. Zasadnicza trudność polega na zapewnieniu komunikacji tego okna z innymi oknami oraz z systemem operacyjnym i urządzeniami peryferyjnymi. Nie sztuka bowiem wyświetlić na ekranie monitora prostokąt i nazwać go oknem. Jak jednak zapewnić naszemu „rysunkowi" możliwość poprawnego współdziałania z innymi „rysunkami" wyświetlanymi na ekranie monitora oraz np. z klawiaturą i myszką? Jak zapewnić możliwość przesuwania i zmiany rozmiaru naszego okna? Jeżeli w danej chwili kilka okien pokrywa się, do którego z nich powinny trafiać informacje o stanie myszki? Jak zapewnić automatyczne zamknięcie wszystkich okien potomnych w przypadku zamknięcia głównego okna aplikacji? Jak zapewnić odświeżenie części okienka dotychczas przesłoniętej przez inne okienko, które właśnie zostało przesunięte w inną część ekranu? Jak zareagować na tak „subtelne" zdarzenia jak zmiana trybu pracy monitora (nowa rozdzielczość) czy rozpoczęcie zamykania systemu operacyjnego (co z naszymi niezapisanymi danymi?).

Oczywiście tworzeniem okienek i obsługą mechanizmów ich komunikacji ze „światem zewnętrznym" powinien zajmować się system operacyjny. W systemie Windows proces konstruowania okna składa się z trzech podstawowych etapów, realizowanych poprzez wywołania odpowiednich funkcji API.

1. Po pierwsze, definiujemy *klasę okien* (ang. *window class*), a więc swego rodzaju matrycę zawierającą ogólne informacje o wyglądzie okienka i sposobie jego komunikacji z systemem operacyjnym. Matryca ta umożliwia tworzenie w prosty sposób szeregu podobnych do siebie okienek różniących się pewnymi „mniej istotnymi" właściwościami, np. położeniem na ekranie. O utworzeniu nowej klasy należy poinformować system operacyjny – służy do tego funkcja **RegisterClass**. Istnieje też kilka klas predefiniowanych; dzięki nim można utworzyć standardowe okna pomijając rejestrację klasy okna. Najważniejszym elementem klasy okien jest tzw. procedura okna, o której piszę poniżej. Po zakończeniu realizacji programu wszystkie zarejestrowane w nim klasy zostaną automatycznie wyrejestrowane.

   Uwaga: Z punktu widzenia programisty C++ *klasa okien* nie ma nic wspólnego z *klasami języka C++*! Ot, zwykła koincydencja nazw!

2. Do faktycznego utworzenia nowego okna musimy użyć nazwy zarejestrowanej (lub predefiniowanej) klasy. W tym celu wykorzystujemy funkcję **CreateWindowEx**.
3. Jednakże bezpośrednio po wywołaniu funkcji **CreateWindowEx** nowoutworzone okno jest… ukryte (tj. nie jest wyświetlane). Dzięki temu aplikacja może dokonać operacji koniecznych do inicjalizacji wyglądu okna *zanim* zostanie ono wyświetlone. Aby ustalić sposób jego wyświetlania (np. czy ma być minimalizowane lub maksymalizowane) posługujemy się funkcją **ShowWindow**. Natomiast funkcja **UpdateWindow** powoduje aktualizację wyglądu obszaru roboczego okna (czyli wszystkiego z wyjątkiem paska tytułowego, krawędzi itp.).

Okna komunikują się z otoczeniem poprzez tzw. system komunikatów. Komunikaty to, w największym uproszczeniu, liczby całkowite przypisane określonym zdarzeniom. Wysyłane są one do okna w celu poinformowania go o zaistnieniu jakiejś sytuacji mającej potencjalny wpływ na jego stan, dzięki czemu okno może w odpowiedni sposób zmodyfikować swój wewnętrzny stan oraz być może zmienić

[2] Programista nie tylko nie musi, ale i fizycznie nie może obsłużyć wszystkich komunikatów: teoretycznie może ich być bowiem… 2^{32}!

sposób, w jaki jest wyświetlane na ekranie. Wraz z liczbą całkowitą określającą rodzaj zdarzenia okno otrzymuje jeszcze dwie dodatkowe liczby precyzujące charakter zdarzenia, zwane parametrami komunikatu. Na przykład zmianie wielkości okienka towarzyszy przesłanie mu komunikatu o numerze 5 oraz dwóch liczb określających wielkość nowego obszaru roboczego okna i tryb zmiany wielkości okna (np. czy okno jest maksymalizowane lub minimalizowane). Dzięki temu okno może dostosować swój wygląd do swojego nowego rozmiaru. Oczywiście, zamiast używać bezpośrednio liczby 5 należy posługiwać się stałymi symbolicznym dostarczanymi wraz z kompilatorem i określonymi w pliku `<winuser.h>` włączanym do programu poprzez plik `<windows.h>`. W szczególności symboliczna nazwa komunikatu numer 5 to `WM_SIZE` (geneza tej nazwy jest prosta: WM to skrót wyrażenia *Windows Message*, czyli „komunikat Windows", a SIZE to „rozmiar").

Każdy program może otworzyć wiele okien jednocześnie. Ponieważ okna mogą nie nadążać z przetwarzaniem komunikatów, system operacyjny dla każdego programu (dokładniej: dla każdego wątku programu) tworzy tzw. ***kolejkę komunikatów*** (*message queue*). Zazwyczaj definiując funkcję `WinMain`, umieszcza się w niej pętlę `while`, która pobiera z kolejki kolejne komunikaty i rozsyła je do odpowiedniego okna. Rozsyłaniem komunikatów zajmuje się należąca do interfejsu API funkcja `DispatchMessage`.

Dochodzimy do fundamentalnego problemu związanego z programowaniem dla Windows – sposobu, w jaki okienka reagują na otrzymywane komunikaty. Oczywiście o obsłudze komunikatów powinien decydować programista. Dlatego konstruując swoje okna, *musimy* stworzyć specjalną funkcję, zwaną ogólnie ***procedurą okna*** (ang. *window procedure*), która określać będzie sposób obsługi komunikatów przez okna. Na szczęście nie musimy definiować jej dla każdego okna osobno – procedura okna jest bowiem wspólna dla całej klasy okien. Co więcej, dzięki wchodzącej w skład Windows API funkcji `DefWindowProc`, która zapewnia standardowy sposób obsługi dowolnego komunikatu, nie musimy definiować sposobu reakcji naszych okien na wszystkie możliwe komunikaty (których potencjalnie mogą być… miliony). Ponadto system operacyjny zapewnia odpowiednie procedury okna wszystkim okienkom standardowym (np. generowanym przez funkcję `MessageBoxEx`).

Procedury okien nigdy nie są wywoływane przez nasz program bezpośrednio, lecz wyłącznie za pośrednictwem systemu operacyjnego, np. poprzez funkcję `DispatchMessage`**.** Zgodnie z dyskusją przeprowadzoną w poprzednim rozdziale, procedury okien muszą być definiowane ze atrybutem `CALLBACK`. Dodajmy jeszcze, że oprócz funkcji `WinMain` i procedur okien programista może zdefiniować wiele innych funkcji wywoływanych, w odpowiednim kontekście, przez system operacyjny (a więc w pewnym sensie stanowiących jego rozszerzenie). Możliwości takiej nie ma oczywiście programista piszący „klasyczną" aplikację w standardowym języku C lub C++. Jedyną funkcją wywoływaną z poziomu systemu operacyjnego jest w nich bowiem funkcja `main`. Możliwość wykorzystywania funkcji typu `CALLBACK` jest podstawową cechą wyróżniającą programowanie dla Windows, a posługiwanie się nimi należy do podstawowych czynności każdego programisty Windows.

Istnieją dwa podstawowe sposoby wykorzystywania procedur okien. Jednym z nich jest wysłanie komunikatu do kolejki. Sposób ten gwarantuje jego obsługę *po* przetworzeniu komunikatów znajdujących się wciąż w kolejce. Komunikaty wstawiane są do kolejki przez system operacyjny, który informuje okienko o różnych zdarzeniach zewnętrznych (np. zmianie szerokości okna, przyciśnięciu klawisza ‘A', etc.), mogą być też wysyłane przez aplikacje. Do wstawiania komunikatów do kolejki służy funkcja `PostMessage`. Istnieje jednak i druga możliwość – uruchomienia procedury okna natychmiast, z pominięciem kolejki komunikatów. Służy do tego funkcja `SendMessage`. Często zdarza się, że procedura okna wysyła przy pomocy tej funkcji komunikaty sama do siebie, co skutkuje natychmiastowym, rekurencyjnym wywołaniem tej samej procedury okna. Istnieją jeszcze inne funkcje służące do wysyłania komunikatów. Na przykład funkcja `PostQuitMessage` służy do wstawiania do kolejki komunikatu `WM_QUIT`. Ogólna zasada brzmi: nazwa funkcji wykorzystującej kolejkę zawiera angielskie słowo *Post*, natomiast nazwa funkcji omijającej kolejkę i bezpośrednio wywołującej procedurę okna zawiera słowo *Send*. Podobne rozróżnienie obowiązuje w anglojęzycznej dokumentacji Windows: zdania „X sends a message to Y" i „X posts a message to Y", mimo iż często tłumaczone tak samo („X wysyła komunikat do Y"), mają więc zupełnie odmienne znaczenie.

Po tym dość długim wstępie możemy przystąpić do przedstawienia prościutkiego programu wyświetlającego napis „Ahoj, przygodo!” w osobnym oknie Windows. Program ten składa się z funkcji **WinMain**, w której, po zarejestrowaniu klasy okien, tworzymy i wyświetlamy nasze okienko, po czym w pętli `while` pobieramy z systemu komunikaty, które przesyłamy do procedury okna. Oprócz funkcji **WinMain** w naszym programie znajduje się procedura okna określająca sposób reakcji okienka na komunikaty.

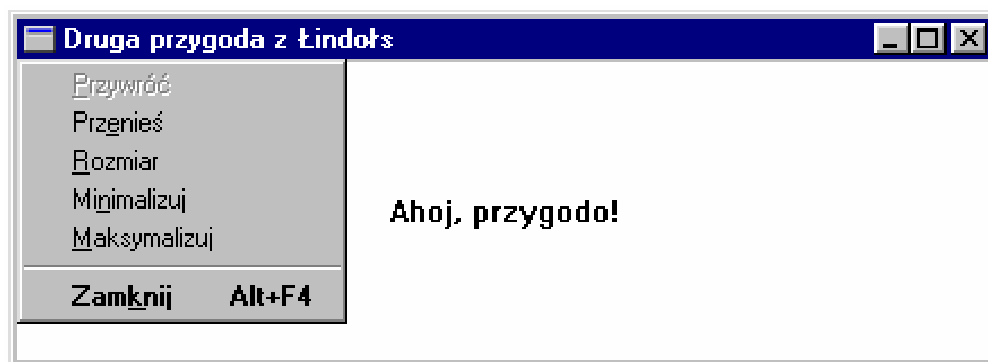
Program „Moje okno”

```
#include <windows.h>
LRESULT CALLBACK ProceduraOkna (HWND, UINT, UINT, LONG); // deklaracja zapowiadająca

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow)
{
    char szClassName[] = „MojeOkno”;
    HWND        hwnd;
    MSG         msg;
    WNDCLASSEX  wndclass;
    wndclass.cbSize          = sizeof(WNDCLASSEX);
    wndclass.style           = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc     = ProceduraOkna;
    wndclass.cbClsExtra      = 0;
    wndclass.cbWndExtra      = 0;
    wndclass.hInstance       = hInstance;
    wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW);
    wndclass.hIcon           = LoadIcon   (NULL, IDI_APPLICATION);
    wndclass.hbrBackground   = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wndclass.lpszMenuName    = NULL;
    wndclass.lpszClassName   = szClassName;
    wndclass.hIconSm         = LoadIcon   (NULL, IDI_APPLICATION);

    RegisterClassEx (&wndclass);
    hwnd = CreateWindowEx ( 0,
          szClassName,
          „Druga przygoda z Łindołs”,
          WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT, CW_USEDEFAULT,
          CW_USEDEFAULT, CW_USEDEFAULT,
          NULL,
          NULL,
          hInstance,
          NULL
    );
    if (hwnd == 0)
    return -1;
    ShowWindow (hwnd, nCmdShow);
    UpdateWindow (hwnd);
    /* Pętla komunikatów: */
    int result;
    while ((result = GetMessage (&msg, NULL, 0, 0) ) != 0)
    {
          if (result == -1) return -1;
          TranslateMessage (&msg);
          DispatchMessage (&msg);
    }
    return msg.wParam;
}
```

```
/***    PROCEDURA OKNA    ***/

LRESULT CALLBACK ProceduraOkna (HWND hwnd, UINT message,
                                UINT wParam, LONG lParam)
{
   switch(message)
   {
      case WM_PAINT:
         {
            PAINTSTRUCT ps;
            RECT        rect;
            HDC hdc = BeginPaint (hwnd, &ps);
            GetClientRect (hwnd, &rect);
            DrawText (hdc, „Ahoj, przygodo!”, -1, &rect,
                      DT_SINGLELINE | DT_CENTER | DT_VCENTER);
            EndPaint (hwnd, &ps);
            return 0;
         }
      case WM_DESTROY:
         {
            PostQuitMessage (0);
            return 0;
         }
   }
   return DefWindowProc (hwnd, message, wParam, lParam);
}
```

Opis działania programu

Uruchomienie tego programu powoduje wyświetlenie okienka przedstawionego poniżej. Jest to w pełni funkcjonalne okienko systemu Windows. Posiada pasek tytułowy, przyciski minimalizacji, maksymalizacji i zamykania okienka, reaguje na podwójne kliknięcie w obszarze paska tytułu, może być przesuwane przy pomocy myszki, posiada obramowanie umożliwiające zmianę jego rozmiaru, reaguje na klawisze systemowe (np. Alt-F4), a po lewej stronie paska tytułowego znajduje się ikonka menu systemowego. Na rysunku przedstawiamy okienko tuż po jej przyciśnięciu myszką. I co najważniejsze – okienko wyświetla napis „Ahoj, przygodo!”, przy czym niezależnie od położenia i wielkości okna napis ten zlokalizowany jest dokładnie w jego środku.



Rysunek 4. *Okienko Ahoj przygodo!*

Konstrukcja funkcji WinMain

Rejestracja klasy okien

Funkcja **WinMain** składa się w naszym programie z dwóch zasadniczych części. W pierwszej z nich rejestrujemy *klasę okna*, po czym wykorzystujemy ją do utworzenia i wyświetlenia na ekranie nowego okna, w drugiej zaś organizujemy *pętlę komunikatów* (zwaną też niekiedy *pompą komunikatów*).

Służąca do rejestracji klasy okna funkcja **RegisterClassEx** wymaga podania adresu struktury typu **WNDCLASSEX** zawierającej niezbędne do rejestracji dane. Zgodnie z poniższą deklaracją, posiada ona aż 12 pól, które pracowicie wypełniamy na początku funkcji **WinMain**.

```
struct WNDCLASSEX {
    UINT      cbSize;          //rozmiar struktury WNDCLASSEX
    UINT      style;           //podstawowy styl okienek danej klasy
    WNDPROC   lpfnWndProc;     //adres procedury okna
    int       cbClsExtra;      //ilość dodatkowych bajtów przydzielanych klasie
    int       cbWndExtra;      //ilość dodatk. bajtów przydzielanych każdemu oknu
    HANDLE    hInstance;       //uchwyt wystąpienia programu
    HICON     hIcon;           //uchwyt ikony
    HCURSOR   hCursor;         //uchwyt kursora
    HBRUSH    hbrBackground;   //uchwyt pędzla używanego do zamalowywania tła
    LPCTSTR   lpszMenuName;    //nazwa menu
    LPCTSTR   lpszClassName;   //nazwa klasy okna
    HICON     hIconSm;         //uchwyt małej ikonki
};
```

Znaczenie poszczególnych składowych jest następujące:

- Składowej `cbSize` zawsze przypisuje się wartość `sizeof(`**WNDCLASSEX**`)`.
- Składowa `style` definiuje ogólny styl okienka, czyli jego podstawowe właściwości. Jest ona konstruowana jako suma bitowa kilku flag, z których warto wymienić cztery: **CS_DBLCLKS**, **CS_NOCLOSE**, **CS_HREDRAW** i **CS_VREDRAW**. Pierwsza z nich powoduje, że okienko będzie reagować na podwójne kliknięcia klawiszami myszki. Druga – uniemożliwi zamknięcie okienka w standardowy sposób (np. kombinacją Alt-F4). Użycie trzeciej lub czwartej flagi powoduje, że po zmianie szerokości (**CS_HREDRAW**) lub wysokości (**CS_VREDRAW**) okna nastąpi automatyczne odświeżenie informacji wyświetlanych w jego obszarze roboczym.
- Składowa `lpfnWndProc` określa adres procedury okna, która obsługiwać będzie komunikaty skierowane do okien danej klasy. Jest to najważniejsza składowa struktury **WNDCLASSEX**.
- Składowe `cbClsExtra` i `cbWndExtra` określają, odpowiednio, ilość dodatkowych bajtów pamięci przydzielanych odpowiednio na potrzeby całej klasy lub poszczególnych okien. W aplikacjach z jednym okienkiem parametry te przyjmują zazwyczaj wartość 0.
- Składowa `hInstance` identyfikuje numer egzemplarza programu. Wielkość tę otrzymujemy z systemu poprzez pierwszy argument funkcji **WinMain**.
- Składowe `hIcon` i `hCursor` określają, odpowiednio, uchwyt do używanej przez aplikację ikony i kursora. Standardową ikonę i standardowy kursor włączamy do aplikacji poprzez wartość funkcji **LoadIcon** i **LoadCursor**, w których jako pierwszy parametr podajemy `NULL`, a jako drugi – odpowiedni identyfikator. Pełna informacja o wszystkich identyfikatorach odpowiadających standardowym ikonom i kursorom dostępna jest poprzez system pomocy kompilatora.
- Składowa `hbrBackground` podaje uchwyt do pędzla, używanego do zamalowywania tła obszaru roboczego okna. Uchwyt pędzla standardowego otrzymujemy poprzez należącą do interfejsu API funkcję **GetStockObject**. Ponieważ funkcja ta może zwracać uchwyty do obiektów różnego typu (nie tylko pędzli, ale i np. piór lub czcionek), użyliśmy operatora rzutowania `(`**HBRUSH**`)`.
- Składowa `lpszMenuName` przechowuje nazwę menu. Wartość `NULL` oznacza brak menu.
- Składowa `lpszClassName` określa nazwę klasy okna. Po rejestracji klasy okna parametr ten będzie służył do jej identyfikacji.

- Składowa hIconSm podaje uchwyt do małej ikonki, tj. ikonki wyświetlanej np. przez program Windows Explorer obok nazw programów.

Tworzenie i wyświetlanie okna

Po zarejestrowaniu klasy okien przystępujemy do utworzenia pierwszego (i jedynego) jego egzemplarza. Wywołujemy w tym celu funkcję **CreateWindowEx**, której deklaracja wygląda następująco:

```
HWND CreateWindowEx(
    DWORD dwExStyle,          // dodatkowy styl tworzonego okna
    LPCTSTR lpClassName,      // nazwa zarejestrowanej (lub predefiniowanej) klasy
    LPCTSTR lpWindowName,     // tytuł okienka. Pojawi się na pasku tytułowym okna
    DWORD dwStyle,            // podstawowy styl okna
    int x,                    // współrzędna lewej krawędzi okna
    int y,                    // współrzędna górnej krawędzi okna
    int nWidth,               // szerokość okna
    int nHeight,              // wysokość okna
    HWND hWndParent,          // uchwyt do okna-rodzica lub okna-właściciela
    HMENU hMenu,              // uchwyt menu lub identyfikator okna, jeśli tworzymy okno potomne
    HINSTANCE hInstance,      // uchwyt wystąpienia programu
    LPVOID lpParam            // wskaźnik do danych użytkownika używanych do inicjalizacji okna
   );
```

Funkcja ta zwraca uchwyt (**HWND**) do nowoutworzonego okna lub 0, jeśli okna nie udało się utworzyć.

Styl okienka określany jest w dwóch argumentach: dwExStyle i dwStyle. Każdy z nich jest zazwyczaj kombinacją kilku spośród kilkudziesięciu flag. Wraz ze składową style struktury **WNDCLASSEX**, używanej podczas rejestracji klasy okien, całkowicie wyznaczają one styl (czyli podstawowe właściwości) tworzonego okna. Możliwych do wykorzystania w tym celu flag jest więc ponad 60. W szczególności, spośród ponad 25 flag używanych do określania wartości parametru dwStyle funkcji **CreateWindowEx**, warto zwrócić uwagę na następującą dziesiątkę:

- **WS_OVERLAPPED**. Tworzone okienko może zachodzić na inne okienka, posiada pasek tytułowy i jest obramowane.
- **WS_CAPTION**. Tworzy okno z paskiem tytułowym.
- **WS_SYSMENU**. Tworzone okienko będzie posiadało menu systemowe.
- **WS_SIZEBOX**. Okienko będzie posiadało grubą ramkę umożliwiającą zmianę jego wielkości.
- **WS_MAXIMIZEBOX**. Na pasku tytułowym będzie się znajdował przycisk „maksymalizuj”.
- **WS_MINIMIZEBOX**. Na pasku tytułowym będzie się znajdował przycisk „minimalizuj”.
- **WS_OVERLAPPEDWINDOW**. Równoważne użyciu wszystkich przedstawionych powyżej flag.
- **WS_CHILD**. Tworzone okienko jest „dzieckiem” innego okna.
- **WS_VSCROLL**. Powoduje utworzenie pionowego paska przewijania.
- **WS_HSCROLL**. Powoduje utworzenie poziomego paska przewijania

Parametr lpClassName określa nazwę klasy okien, do której należeć będzie nasze okno, a więc pośrednio determinuje jego procedurę okna, czyli sposób przetwarzania docierających do niego komunikatów. Parametr lpWindowName definiuje napis, który pojawi się na pasku tytułowym naszego okienka. Parametry x i y funkcji **CreateWindowEx** określają początkowe położenie lewego górnego wierzchołka okna względem lewego górnego rogi ekranu, któremu odpowiadają wartości x = 0, y = 0. Kolejne dwie wielkości, nWidth i nHeight, determinują początkową szerokość i wysokość okna. Wartości parametrów x, y, nWidth i nHeight podajemy w pikselach. Użycie stałej symbolicznej **CW_USEDEFAULT** oddaje inicjatywę systemowi, który w tym przypadku sam określi wielkość i położenie okienka. Parametr hMenu określa uchwyt do menu (wartość NULL oznacza brak menu), a hInstance – uchwyt wystąpienia programu. Obie te wielkości podawaliśmy już podczas rejestracji klasy okien. Wskaźnik lpParam umożliwia przekazanie procedurze okna dodatkowych informacji,

które mogą być przez nią wykorzystane podczas inicjalizacji okna. My z tej możliwości nie korzystamy, dlatego przyjmujemy `lpParam = NULL`.

Utworzone okno należy jeszcze wyświetlić na ekranie. Korzystamy w tym celu z dwóch omówionych wcześniej funkcji: **`ShowWindow`** i **`UpdateWindow`**.

Pętla komunikatów

Na końcu funkcji **`WinMain`** definiujemy pętlę komunikatów. Składa się ona z pojedynczej instrukcji `while`, w której testujemy wartość zwracaną przez funkcję **`GetMessage`**, pobierającą komunikaty z nadzorowanej przez Windows kolejki. Mimo iż wartość tej funkcji została zdefiniowana przez jej twórców jako **`BOOL`**, zgodnie ze swoim opisem może ona przyjąć **trzy** (!) wartości:

- **`FALSE`** (czyli 0) jeżeli z kolejki pobrano kończący wykonanie programu komunikat **`WM_QUIT`**;
- –1 jeśli podczas realizacji funkcji **`GetMessage`** wystąpił błąd;
- **`TRUE`** (czyli 1) w pozostałych przypadkach.

Funkcja **`GetMessage`** przyjmuje aż cztery argumenty. Pierwszy z nich jest wskaźnikiem do struktury typu **`MSG`**. Poszczególne pola tej struktury wypełniane są przez Windows podczas realizacji funkcji **`GetMessage`** i informują nas o uchwycie okna, do którego skierowany jest komunikat, numerze komunikatu, zawartości dwóch dodatkowych parametrów komunikatu, czasie wstawienia komunikatu do kolejki oraz położeniu kursora (we współrzędnych ekranu, tj. względem jego lewego górnego wierzchołka) w chwili wstawienia komunikatu do kolejki. Drugi argument funkcji **`GetMessage`** umożliwia podanie uchwytu okna, którego komunikaty funkcja ta ma pobierać z kolejki. Wartość `NULL` oznacza, że chcemy pobierać wszystkie komunikaty skierowane do dowolnego okna naszej aplikacji (dokładniej: wątku aplikacji). Natomiast trzeci i czwarty parametr funkcji **`GetMessage`** umożliwia ograniczenie zakresu pobieranych komunikatów do pewnego przedziału wartości. Wstawienie tu dwóch zer oznacza, że funkcja **`GetMessage`** pobierać będzie wszystkie komunikaty.

Po pobraniu komunikatu z kolejki dokonujemy dwóch operacji. Po pierwsze, tradycyjnie wywołujemy funkcję **`TranslateMessage`**. Jej działanie, w największym uproszczeniu, powoduje, że system wyręcza nas w tłumaczeniu stanu klawiatury na komunikaty informujące okno o wysłaniu doń z klawiatury określonego znaku (np. czy użytkownik wprowadził do edytora znak `'a'`, `'A'`, `'ą'` czy `'Ą'`,

Następnie wywołujemy funkcję **`DispatchMessage`**. Jej zadaniem jest wysłanie komunikatu, pobranego przed chwilą z kolejki Windows (funkcją **`GetMessage`**), do odpowiedniej procedury okna.

Procedura okna

Konstrukcja procedury okna

Pozostało nam najważniejsze i najtrudniejsze zadanie – obsługa komunikatów, którymi bombardowane będzie nasze okno. W tym celu definiujemy, *jako osobną funkcję*, procedurę okna. W naszym programie jest to funkcja o nazwie `ProceduraOkna`. Jak każda funkcja użytkownika wywoływana przez system i obsługująca komunikaty, zadeklarowana jest ona z atrybutem **`CALLBACK`**, a jej wartość jest typu **`LRESULT`**, czyli (obecnie) `long int`. Pobiera ona z systemu cztery parametry:

- **`HWND`** `hwnd`. Podaje uchwyt okna, do którego kierowany jest dany komunikat.
- **`UINT`** `message`. Identyfikuje komunikat.
- **`UINT`** `wParam`. Przekazuje dodatkowe informacje związane z komunikatem.
- **`LONG`** `lParam`. Również przekazuje dodatkowe informacje związane z komunikatem.

Parametr `hwnd` informuje nas o tym, które okno tak naprawdę obsługujemy. Jest to informacja niezbędna przy wywoływaniu wielu podstawowych funkcji API. Jest ona szczególnie ważna w aplikacjach, w których tworzymy kilka okien należących do tej samej klasy, gdyż w tym przypadku jedna procedura okna musi je wszystkie obsłużyć niezależnie od tego, w jakim akurat znajdują się stanie.

W naszym prostym przykładzie nie wykorzystujemy wartości parametrów `wParam` i `lParam`, jednak zazwyczaj niosą one bardzo istotne informacje dotyczące komunikatu. Informacje te zależą jednak od konkretnego komunikatu – z każdym razem, gdy przetwarzamy jakiś komunikat, musimy *bardzo uważnie* zaznajomić się ze znaczeniem zawartych w nich informacji. Najwygodniej jest w tym celu posłużyć się systemem pomocy naszego kompilatora. Jak cenne wiadomości przekazywane są za ich pomocą zobaczymy już w następnym rozdziale.

Wartość parametru `message` testowana jest w instrukcji `switch`. Jest to bardzo charakterystyczny sposób konstrukcji procedury okna. Często na jej początku deklarowane są pewne zmienne statyczne, (czyli zachowujące swoje wartości pomiędzy jej kolejnymi wywołaniami), po czym następuje ogromna instrukcja `switch`. Ogromna – bo najczęściej musi obsłużyć dużo więcej niż dwa komunikaty.

Wartości komunikatów (które są zwykłymi liczbami całkowitymi) powinno się określać za pomocą standardowych stałych symbolicznych, np. **`WM_PAINT`** lub **`WM_DESTROY`**. Po obsłużeniu komunikatu należy zwrócić do systemu 0, chyba, że dokumentacja dotycząca danego komunikatu mówi coś innego. Ten drugi przypadek w praktyce spotyka się jednak rzadko, gdyż dotyczy komunikatów systemowych, których obsługę lepiej pozostawić systemowi operacyjnemu.

Domyślna procedura okna

Jeżeli nie chcemy (lub nie potrafimy) obsłużyć pewnych komunikatów, powinniśmy zlecić to należącej do systemu API funkcji **`DefWindowProc`**. Jest to bardzo ważny element konstrukcji procedury okna, gdyż to właśnie **`DefWindowProc`** potrafi odpowiednio obsłużyć komunikaty systemowe.

W naszym przykładowym programie procedura okna przetwarza tylko dwa komunikaty, zrzucając resztę pracy na funkcję **`DefWindowProc`**. Pierwszy z nich, **`WM_PAINT`**, przekazywany jest do procedury okna w sytuacji, gdy wymagane jest odświeżenie informacji wyświetlanych w jego obszarze roboczym. Dzieje się tak na przykład po zmianie rozmiaru okna (o ile podczas rejestracji klasy użyto flag **`CS_HREDRAW`** i **`CS_HREDRAW`**) lub gdy część naszego okna odsłaniana jest na skutek przemieszczenia lub zamknięcia innego okna. Natychmiastowe odświeżenie okna wymusza także wykorzystana w funkcji **`WinMain`** funkcja **`UpdateWindow`**.

Komunikat WM_PAINT

Komunikat **`WM_PAINT`** posiada wiele własności wyróżniających go spośród innych komunikatów. Po pierwsze, traktowany jest on przez Windows jako komunikat o wyjątkowo małym priorytecie. Oznacza to, że jeżeli znajduje się on w kolejce komunikatów, to zawsze na jej końcu. Jeżeli w pewnym momencie do kolejki wstawiany jest nowy komunikat, wygenerowany np. przez myszkę, to „przeskakuje” on komunikat **`WM_PAINT`**, zajmując przedostatnie miejsce w kolejce. Jeżeli do kolejki zostanie wstawiony nowy komunikat **`WM_PAINT`**, a poprzedni wciąż się w niej znajduje, oba połączone będą w jeden „wypadkowy” komunikat **`WM_PAINT`**. Możemy jednak wymusić natychmiastowe odświeżenie zawartości obszaru roboczego okna omijając kolejkę. W tym celu można posłużyć się np. funkcją **`UpdateWindow`**. Ponadto komunikat **`WM_PAINT`** jako jedyny nie może być usunięty z kolejki komunikatów po prostu poprzez wywołanie funkcji **`GetMessage`**.

Obsługa tego komunikatu ma też specjalne znaczenie z punktu widzenia programisty. Musi być on przygotowany na to, że jego program dosłownie w każdej chwil będzie musiał obsłużyć ten komunikat, aktualizując informacje wyświetlane w okienku. W każdej chwili można bowiem oczekiwać np. zasłonięcia i odsłonięcia części naszego okna przez inne okno, co spowoduje wygenerowanie przez Windows komunikatu **`WM_PAINT`**. Procedura okna musi mieć więc dostęp do wszystkich parametrów koniecznych do wyświetlenia aktualnego stanu okna. Nawet jeżeli w okienku rysujemy spoza kodu obsługującego komunikat **`WM_PAINT`**, musimy mieć absolutną gwarancję, że po otrzymaniu tego komunikatu procedura okna wykonałaby dokładnie taki sam rysunek.

W naszym przykładzie obsługa komunikatu **WM_PAINT** jest bardzo prosta, a zarazem bardzo typowa:

```
case WM_PAINT:
     {
          HDC         hdc;
          PAINTSTRUCT ps;
          RECT        rect;

          hdc = BeginPaint (hwnd, &ps);
          GetClientRect (hwnd, &rect);
          DrawText (hdc, „Ahoj, przygodo!”, -1, &rect,
                    DT_SINGLELINE | DT_CENTER | DT_VCENTER);
          EndPaint (hwnd, &ps);
          return 0;
     }
...
```

Po zdefiniowaniu potrzebnych nam struktur danych wywołujemy funkcję, która zwraca uchwyt typu **HDC**, czyli uchwyt do tzw. kontekstu urządzenia. Kontekst urządzenia omówimy szerzej w następnym rozdziale, tu nadmienimy jedynie, że jest to wielkość niezbędna do wywołania jakiejkolwiek funkcji graficznej systemu Windows. Funkcję **BeginPaint** wolno wywołać wyłącznie w ramach obsługi komunikatu **WM_PAINT**. Jej zadaniem jest wyjęcie go z kolejki komunikatów oraz poinformowanie systemu, że jeżeli istniały dotąd jakiekolwiek powody, by uważać, że nasze okienko powinno otrzymać ten komunikat (czyli zostać odświeżone), to powinien uznać je za niebyłe. Dzięki temu Windows nie będzie bombardował naszego okna serią komunikatów **WM_PAINT**. Ponadto, jeżeli nasz program korzysta z tzw. kursora karetki (służącego do wskazywania miejsca wprowadzania znaków z klawiatury), funkcja **BeginPaint** spowoduje jego schowanie. Podczas realizacji funkcji **BeginPaint** system operacyjny wypełnia strukturę (typu **PAINTSTRUCT**) wskazywaną przez jej drugi argument informacjami umożliwiającymi optymalizację kodu procedury okna. Odpowiednie składowe tej struktury informują bowiem o tym, czy w trakcie wywołania funkcji tło obszaru roboczego okna zostanie zamalowane domyślnym pędzlem, ustalanym podczas rejestracji klasy okien, oraz do jakiego prostokątnego fragmentu obszaru roboczego ograniczone zostanie działanie funkcji graficznych wykorzystujących zwracany przez funkcję **BeginPaint** uchwyt kontekstu urządzenia. To ograniczenie, zwane też obcinaniem (ang. *clipping*), wiąże się z tym, że często zachodzi potrzeba odtworzenia tylko fragmentu okna. Dzieje się tak np. wtedy, gdy jego część zostanie przesłonięta przez okno dialogowe, które po pewnym czasie zniknie, pozostawiając prostokątną „dziurę” wewnątrz głównego okna aplikacji. Aby wypełnić ten obszar treścią, Windows wysyła do odpowiedniego okna komunikat **WM_PAINT**, zaznaczając jednocześnie, że w trakcie przetwarzania go nie należy aktualizować pozostałej części okna, gdyż jest ona wyświetlana poprawnie. Programista, analizując informacje dostarczane przez drugi parametr funkcji **BeginPaint**, może więc dostosować do nich swój kod tak, aby niepotrzebnie nie wywoływać funkcji, które i tak nie będą miały żadnego praktycznego efektu (jeżeli usiłowałyby rysować w obszarze uznanym już za „narysowany”). Istnieją też odpowiednie funkcje służące do „ręcznego” zarządzania wielkością obszaru obcinania, m.in. **InvalidateRect** i **ValidateRect**.

Obsługując komunikat **WM_PAINT**, musimy pamiętać, by po zakończeniu rysowania zwolnić kontekst urządzenia przy pomocy funkcji **EndPaint**. Funkcja ta wyświetli również kursor karetki, o ile był on wyświetlany przed wywołaniem funkcji **BeginPaint**.

Przed wyświetleniem napisu „Ahoj, przygodo!”, przy pomocy funkcji **GetClientRect** sprawdzamy, jaki jest aktualny rozmiar obszaru roboczego naszego okna, czyli obszaru, w którym możemy rysować.

```
GetClientRect (hwnd, &rect);
```

Funkcja ta jako pierwszy argument pobiera uchwyt okna, które ma „obmierzyć”, wyniki zaś swoich obliczeń umieszcza w strukturze typu **RECT** wskazywanej przez drugi argument. Zgodnie z deklaracją tej bardzo często używanej struktury

```
typedef struct _RECT {
     LONG left;          // lewy
     LONG top;           // górny
     LONG right;         // prawy + 1
     LONG bottom;        // dolny +1
}RECT;
```

dwie pierwsze jej składowe oznaczają położenie lewego górnego wierzchołka prostokąta, natomiast składowe `right` i `bottom` – składowe prawego dolnego narożnika. Funkcja **`GetClientRect`** operuje we współrzędnych obszaru roboczego, dlatego składowe `left` i `top` struktury `rect` równe będą 0.

Po wyznaczeniu współrzędnych obszaru roboczego wywołujemy funkcję **`DrawText`**.

```
DrawText (hdc, „Ahoj, przygodo!”, -1, &rect,
                    DT_SINGLELINE | DT_CENTER | DT_VCENTER);
```

Ponieważ funkcja ta będzie dokonywała operacji graficznych, pierwszym jej parametrem jest otrzymany z funkcji **`BeginPaint`** uchwyt kontekstu urządzenia. Drugim argumentem jest wyświetlany napis, a trzeci określa ilość wyświetlanych znaków; wartość –1 oznacza, że należy wyświetlać wszystkie znaki drugiego argumentu aż do napotkania znaku `'\0'` niejawnie kończącego wszystkie standardowe napisy języka C. Czwarty parametr podaje współrzędne prostokąta, wewnątrz którego należy umieścić napis. Ostatni, piąty parametr określa sposób wyświetlania napisu i konstruowany jest jako suma bitowa odpowiednich flag. W naszym przypadku napis będzie wycentrowany w poziomie (**`DT_CENTER`**) i pionie (**`DT_VCENTER`**), i zajmie tylko jedną linię (**`DT_SINGLELINE`**).

Komunikat WM_DESTROY i zakończenie programu

Istnieje jeden komunikat, który **musimy** obsłużyć w procedurze okna: **`WM_DESTROY`**. Jest on wysyłany do okna, gdy zamykamy je naciskając krzyżyk w jego prawym górnym rogu lub naciskając kombinacje klawiszy ALT-F4. Po otrzymaniu tego komunikatu aplikacja może dokonać pewnych operacji koniecznych do prawidłowego zakończenia programu, np. zamknąć pliki, zwolnić zasoby, etc. Następnie wstawiamy do kolejki komunikatów komunikat **`WM_QUIT`**.

```
PostQuitMessage (0);
```

Ponieważ system operacyjny sam z siebie nigdy nie generuje komunikatu **`WM_QUIT`**, powyższa funkcja to *jedyny* sposób, aby przerwać pętlę komunikatów uruchomioną w funkcji **`WinMain`**, a więc i jedyny sposób na zakończenie programu. Bez tej instrukcji moglibyśmy zamknąć wszystkie okna stworzone w naszej aplikacji, lecz mimo to pozostałaby działająca w tle funkcja **`WinMain`**.

Obsługę komunikatów **`WM_PAINT`** i **`WM_DESTROY`** kończymy, zwracając do systemu zero

```
return 0;
```

Komunikaty nieobsłużone w sposób jawny w naszej procedurze okna przekazujemy do domyślnej procedury okna, zwracając na zewnątrz jej wartość:

```
return DefWindowProc (hwnd, message, wParam, lParam);
```

Standardowe klasy okien

Istnieje kilka predefiniowanych klas okien, których nazw możemy użyć w wywołaniu funkcji **`CreateWindowEx`**. Są to: `"`**`BUTTON`**`"`, `"`**`COMBOBOX`**`"`, `"`**`EDIT`**`"`, `"`**`LISTBOX`**`"`, `"`**`MDICLIENT`**`"`, `"`**`RichEdit`**`"`, `"`**`RICHEDIT_CLASS`**`"`, `"`**`SCROLLBAR`**`"` i `"`**`STATIC`**`"`. Na przykład poniższa instrukcja powoduje utworzenie typowego przycisku Windows z napisem „przyciśnij mnie!”; przycisk ma szerokość 300 i wysokość 30 pikseli, jego lewy górny wierzchołek w układzie obszaru roboczego głównego okna aplikacji ma współrzędne (10,20), jest okienkiem podrzędnym („dzieckiem”) okna `hwnd` i przypisaliśmy mu identyfikator `1`.

```
HWND hbutton =  CreateWindowEx (0, "BUTTON", "przyciśnij mnie!",
                WS_CHILD | WS_VISIBLE  | BS_PUSHBUTTON, 10, 20, 300,
                30, hwnd, HMENU(1), hinst, 0);
```

Podsumowanie

- Przed utworzeniem własnego okna należy zarejestrować nową klasę okien (przy pomocy funkcji `RegisterClass`) lub wykorzystać jedną z klas predefiniowanych (np. `"BUTTON"`).
- Do faktycznego utworzenia okna używamy funkcji `CreateWindowEx`.
- Bezpośrednio po wywołaniu funkcji `CreateWindowEx` okno jest w stanie ukrytym. Aby je wyświetlić na ekranie można wywołać funkcje `ShowWindow` i następnie `UpdateWindow`.
- Dla każdego wątku programu system operacyjny tworzy osobną kolejkę komunikatów.
- System komunikatów zapewnia komunikację okienek ze światem zewnętrznym.
- Komunikaty są wysyłane przez lub za pośrednictwem systemu operacyjnego. Ich obsługą zajmują się specjalne funkcje użytkownika, zwane procedurami okien.
- W przeciwieństwie do tradycyjnych programów, które same zajmują się testowaniem urządzeń zewnętrznych, programy działające w systemie Windows czekają na docierające do nich komunikaty, po czym je przetwarzają. Z tego powodu mówi się, że twórcy programów dla Windows przyjmują postawę defensywną, tak tworząc kod, by odpowiadał na „kanonadę” komunikatów.
- Aby obsłużyć komunikaty, w funkcji `WinMain` organizujemy tzw. pętlę komunikatów. Pobieramy je z kolejki przy pomocy funkcji `GetMessage`, po czym rozsyłamy je do odpowiedniej procedury okna przy pomocy funkcji `DispatchMessage`.
- Jedna procedura okna obsługuje całą klasę okien.
- Procedurę okna definiujemy z atrybutem `CALLBACK`. Przyjmuje ona cztery argumenty: uchwyt okna, do którego skierowany jest komunikat, identyfikator komunikatu oraz dwa pomocnicze parametry zwyczajowo oznaczane jako `wParam` i `lParam`.
- Znaczenie parametrów komunikatu zależy od samego komunikatu. Mogą one odpowiadać liczbom całkowitym, grupom bitów (masek) lub adresom różnych struktur zawierających dodatkowe informacje.
- Główną częścią procedury okna jest zazwyczaj instrukcja `switch`, testująca wartość komunikatu i zapewniająca jego obsługę.
- Komunikat `WM_PAINT` sygnalizuje procedurze okna konieczność odświeżenia zawartości (części) obszaru roboczego okna.
- W dowolnej sytuacji, aby móc rysować, należy wcześniej otrzymać z systemu tzw. uchwyt kontekstu urządzenia. W przypadku obsługi komunikatu `WM_PAINT` uchwyt ten otrzymujemy wywołując funkcję systemową `BeginPaint` („konstruktor malowania”). Po zakończeniu rysowania musimy wywołać funkcję `EndPaint` („destruktor malowania”).
- Aby przerwać pętlę komunikatów i zakończyć działanie funkcji `WinMain`, aplikacja musi umieścić w kolejce komunikat. `WM_QUIT`. Najczęściej programista stosuje w tym celu systemową funkcję `PostQuitMessage`, którą umieszcza w kodzie obsługi komunikatu `WM_DESTROY`.
- Po pomyślnym zakończeniu obsługi komunikatu procedura okna (najczęściej) zwraca jako swoją wartość 0.
- Komunikaty nie obsługiwane w sposób jawny w procedurze okna powinny zostać skierowane do standardowej procedury okna, czyli `DefWindowProc`. W szczególności funkcji tej powinniśmy powierzyć przetwarzanie komunikatów systemowych (chyba, że potrafimy ją wyręczyć…).
- Istnieje wiele standardowych klas okien, unifikujących aplikacje działające w Windows, m.in. `"BUTTON"`, `"COMBOBOX"` i `LISTBOX"`. Por.: dokumentacja systemu.

Programy przykładowe

1. Program `0MojeOkno` – Jest to kod źródłowy programu omawianego w tym rozdziale.
2. Program `1Przyciski_w_oknie` zawiera przykład użycia standardowej klasy okien `"BUTTON"`.

Zadania

1) W kodzie przedstawionego w tym rozdziale programu dokonaj odpowiednich zmian tak, aby:
   - **a)** Tło obszaru roboczego miało kolor jasnoszary (`LTGRAY_BRUSH`),
   - **b)** Kursor miał kształt krzyża (`IDC_CROSS`),
   - **c)** Ikoną programu była ikona Windows (`IDI_WINLOGO`),
   - **d)** W funkcji `WinMain` tworzone były 2 okna klasy `MojeOkno`, pierwsze o rozmiarze 200×200 pikseli, drugie – 300×400 pikseli,
   - **e)** Nie można było zmieniać rozmiaru tych okien (brak możliwości minimalizacji, maksymalizacji lub zmiany rozmiaru poprzez przeciąganie myszką ich boków), ale można było je przesuwać,
   - **f)** Okna miały (różne) tytuły, lecz wyświetlały ten sam napis „Ahoj, przygodo!”,
   - **g)** Okna miały menu systemowe.
2) Najprawdopodobniej zamknięcie jednego z okien utworzonych w powyżej opisanym programie spowoduje natychmiastowe zamknięcie drugiego okna. Dlaczego? Spróbuj tak przepisać procedurę okna, by wyeliminować tę cechę programu.
3) Proszę sprawdzić, co się stanie, jeśli w jakikolwiek sposób „odchudzimy” przedstawiony tu program. Oto kilka możliwych pomysłów
   - **a)** Proszę sprawdzić, co się stanie, gdy zrezygnujemy z napisania własnej procedury okna, przypisując składowej `lpfnWndProc` struktury `wndclass` wartość 0.

     Odpowiedź: program nie wyświetli żadnego okna, lecz natychmiast zakończy swoje działanie – powodem jest to, że funkcja `CreateWindowEx` zakończy się niepowodzeniem (co zasygnalizuje, zwracając 0). Gdybyśmy nie sprawdzali, czy funkcja `CreateWindowEx` zwraca poprawny uchwyt do okna, program dotarłby do pętli `while`, którą wykonywałby w nieskończoność. Program zamieniłby się w „zombi” – jedyną metodą jego zakończenia byłoby usunięcie go za pośrednictwem Menedżera Programów (Ctrl-Alt-Del) lub przez wyłączenie komputera.
   - **b)** Co się stanie, gdy zrezygnujemy z tworzenia pętli komunikatów?

     Odpowiedź: okno się wyświetli, lecz natychmiast zniknie z ekranu. Powód: bez pętli komunikatów funkcja `WinMain` natychmiast zakończy swoje działanie, a to spowoduje automatyczne zwolnienie wszystkich pobranych przez nią zasobów komputera – m.in. czcionek, pędzli i właśnie **okien**.
   - **c)** Co się stanie, gdy zrezygnujemy z obsługi komunikatu `WM_DESTROY`?

     Odpowiedź: Gdy zamkniemy okno (Alt-F4 lub kliknięcie myszką w „krzyżyk” na pasku tytułowym) okno zniknie i może się zdawać, że wszystko jest OK. Okazuje się jednak, że pętla komunikatów wciąż będzie działać. Funkcja `WinMain` nie przerwie swojego działania, a program przejdzie więc w nieprzyjemny stan „zombi” – por. punkt b).
   - **d)** (Uwaga: zadanie niebezpieczne!) Co się stanie, gdy zrezygnujemy z wywołania funkcji `DefWindowProc`, np. zastępując instrukcję `return DefWindowProc (hwnd, message, wParam, lParam);` instrukcją `return 0;` lub `return 1;`?

     Uwaga: Zaczną się dziać różne *dziwne* rzeczy; być może będziesz musiał(a) przeładować system operacyjny!
4) Komunikat `WM_DESTROY` jest wysyłany do okienka w momencie jego destrukcji – gdy okienko otrzyma ten komunikat, nie ma żadnej siły, która mogłaby ten proces powstrzymać. Ale wcześniej okienko otrzymuje komunikat `WM_CLOSE`, informujący, że użytkownik wyraził chęć zamknięcia okna (np. przez naciśniecie klawiszy Alt-F4). Okienko może albo posłusznie spełnić życzenie użytkownika i zamknąć się (instrukcjami `DestroyWindow(hwnd); return 0;`) lub odmówić dokonania samobójstwa (`return 0;`). Do zaprezentowanego tu programu dodaj kod obsługi komunikatu `WM_CLOSE`. Powinien on pytać użytkownika o to, czy na pewno życzy sobie zamknięcia okna (w okienku wyświetlonym funkcją `MessageBoxEx`) i w zależności od otrzymanej informacji wstrzymać bądź kontynuować proces samozagłady.

3. Interfejs GDI

Wiemy już z grubsza, jak przy pomocy funkcji interfejsu API posługiwać się standardowymi okienkami dialogowymi, potrafimy też otworzyć swoje własne okienko. Ale jakże niewiele jeszcze z tym okienkiem potrafimy zrobić: otworzyć, wyświetlić w nim napis i zamknąć. Nadszedł czas, by zapoznać się z bardziej zaawansowanymi i zdecydowanie ciekawszymi technikami programowania w systemie Windows. Pora zapoznać się z podsystemem interfejsu API odpowiedzialnym za operacje graficzne. Podsystem ten nosi nazwę GDI (*Graphics Device Interface*).

Interfejs GDI zawiera mnóstwo funkcji umożliwiających wykonywanie praktycznie dowolnych operacji graficznych na dowolnym urządzeniu obsługiwanym przez Windows. Pomysł jest genialny w swej prostocie – funkcje pisane przez programistów powinny kierować żądania wykonywania odpowiednich operacji graficznych za pośrednictwem pewnej abstrakcyjnej warstwy oddzielającej je od urządzeń fizycznych. I dopiero na poziomie tej abstrakcyjnej warstwy powinno decydować się, czy rozkazy będą tłumaczone na język drukarki, plotera czy karty graficznej naszego komputera. Dzięki temu ta *sama funkcja* może obsługiwać rysowanie na ekranie i drukowanie na papierze, i to niezależnie od typu używanego sprzętu. Tą abstrakcyjną warstwą oddzielającą użytkownika od sprzętu jest właśnie interfejs GDI. Dzięki niemu programista nie musi dołączać do swojego każdego programu setek sterowników drukarek (a później uaktualniać je o nowe modele) ani czynić restrykcyjnych założeń co do rodzaju i trybu pracy karty graficznej; to producent sprzętu odpowiedzialny jest za dostarczenie specjalnego programu, zwanego sterownikiem, odpowiedzialnego za współpracę swojego produktu z programami uruchomionymi pod kontrola systemu Windows.

Jedną z podstawowych koncepcji interfejsu GDI jest ***kontekst urządzenia*** (*device context*). Jest to nasz abstrakcyjny model *konkretnego* urządzenia odpowiedzialnego za wyświetlanie linii, okręgów, napisów itp. W naszym programie możemy potrzebować wielu kontekstów urządzeń, np. każde okienko posiada własny kontekst urządzenia (który przechowuje m.in. informacje o tym, jaka część obszaru roboczego danego okienka jest widoczna na ekranie i nie pozwala nam rysować poza tym obszarem). Szczegółowa postać kontekstów urządzeń jest dla programisty niedostępna (w końcu jest to model abstrakcyjny); programista posługuje się jedynie tzw. uchwytem do kontekstu urządzenia, czyli pewną liczba magiczną, która jednoznacznie identyfikuje każdy kontekst urządzenia.

Oczywiście różne urządzenia mają różne możliwości (wystarczy porównać monitor z drukarką lub prostą drukarkę igłową z laserowym kombajnem), programista może więc w swoim kodzie odpytać używany w danej chwili kontekst urządzenia o podstawowe parametry związanego z nim urządzenia. Programista może tworzyć też bardzo szybkie *logiczne* konteksty urządzeń, niezwiązane z żadnymi urządzeniami fizycznymi, lecz przechowywane w pamięci operacyjnej komputera; w nich to można swobodnie przeprowadzać operacje graficzne, by za chwilę gotowy wynik swej pracy błyskawicznie przesłać do kompatybilnego urządzenia fizycznego.

Elastyczność kontekstów urządzeń ma jednak swoją cenę. Podsystem GDI nie grzeszy bowiem zawrotną prędkością. Co prawda jego możliwości zupełnie wystarczają twórcom arkuszy kalkulacyjnych czy gier karcianych, ale do wielu innych zastosowań, np. gier komputerowych, nie nadaje się on zupełnie; w tych przypadkach zastępowany jest innymi, szybszymi, ale też znacznie bardziej skomplikowanymi bibliotekami, np. DirectX.

Zasadniczo interfejs GDI składa się z czterech podstawowych części:

1. Funkcje do rysowania i wypełniania różnych linii i figur geometrycznych.
2. Funkcje obsługujące mapy bitowe.
3. Funkcje do wyboru i wyświetlania czcionek oraz drukowania tekstu.
4. Funkcje zarządzające regionami i tzw. przycinaniem.

Najważniejsze elementy tego systemu omówię pokrótce w kolejnych paragrafach.

Kontekst urządzenia

Jak już wspomniałem, kontekst urządzenia to nasz łącznik z urządzeniami fizycznie wyświetlającymi bądź drukującymi takie obiekty graficzne jak linie, łuki czy napisy. W pewnym sensie można go porównać do deskryptora plików. Gdy chcemy coś odczytać z pliku, wywołujemy odpowiednią funkcję systemową, która zwraca deskryptor pliku (bądź inny rodzaj uchwytu) za pośrednictwem którego możemy odczytać interesujące nas informacje, ani przez chwilę nie przejmując się tym, czy pobierane dane pochodzą z dyskietki, dysku twardego, dysku sieciowego, innego programu czy też nawet klawiatury. Analogicznie konteksty urządzeń są swoistymi „uchwytami" do urządzeń wyświetlających.

Bodaj wszystkie funkcje odpowiedzialne za rysowanie czegoś na ekranie wymagają podania kontekstu urządzenia (tak, jak funkcje zapisujące dane na dysku wymagają dostarczenia im uchwytu pliku). Od tej pory będzie więc to nasz bardzo bliski znajomy.

Każdy kontekst urządzenia posiada określony stan zapamiętywany pomiędzy kolejnymi operacjami graficznymi. Dzięki temu zapamiętuje on, jakim piórem ostatnio rysowaliśmy linie, jakiego pędzla używaliśmy do wypełniania obszarów zamkniętych, jaką czcionką wypisywaliśmy tekst, w jakim obszarze ekranu wolno nam rysować, gdzie ostatnio skończyliśmy rysować linię itp. W każdej chwili możemy użyć w danym kontekście urządzenia inne pióro, inny pędzel czy inną czcionkę. Operację tę nazywamy ***wybieraniem*** (*selection*).

Pobieranie kontekstu urządzenia

W systemie Windows nie istnieje „standardowe urządzenie graficzne", które mogłoby pełnić rolę analogiczną do standardowego strumienia wyjścia dla plików. Za każdym razem, gdy chcemy coś narysować, musimy wpierw pobrać z systemu kontekst urządzenia. Jest w tym głęboki sens: z jednej strony w tej samej chwili na ekranie może być wyświetlanych kilka zachodzących na siebie okienek, wszystkie operacje na ekranie muszą wiec przechodzić przez system operacyjny, który wie, jaki obszar ekranu jest zarządzany przez które okno; z drugiej zaś strony jeden program może wyświetlać jednocześnie kilka, a nawet kilkanaście okienek i nie ma żadnego sposobu, by w jakiś sposób wyróżnić jedno z nich jako „okno domyślne" .

Istnieją dwie podstawowe metody otrzymywania kontekstu urządzenia związanego z istniejącym urządzeniem fizycznym. Pierwszą z nich stosujemy tylko i wyłącznie podczas przetwarzania znanego już nam komunikatu **`WM_PAINT`**. W tym przypadku wywołujemy funkcję **`BeginPaint`** i odczytujemy uchwyt do kontekstu urządzenia z odpowiedniej składowej struktury **`PAINTSTRUCT`**. W pozostałych sytuacjach, czyli gdy chcemy coś narysować natychmiast, bez pośrednictwa kolejki komunikatów, stosujemy drugą metodę: wywołujemy funkcję **`GetDC`**.

Zwalnianie kontekstu urządzenia

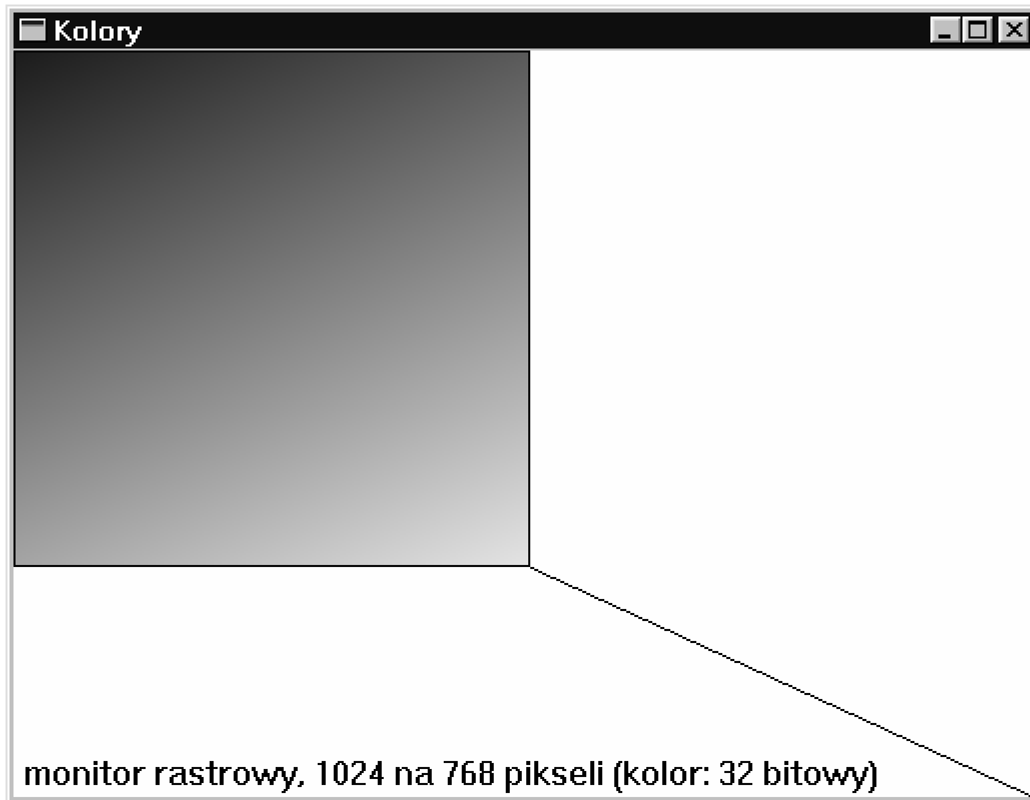
Jak już wiemy, konteksty urządzenia posiadają wewnętrzny stan. To oznacza, że pobierają z systemu pewne zasoby – na pewno należy do nich pamięć. Dlatego, gdy tylko kontekst urządzenia przestaje nam być potrzebny, powinniśmy go zwolnić, zwalniając w ten sposób wszystkie związane z nim zasoby. Robimy to na jeden z dwóch sposobów. Jeżeli kontekst uzyskaliśmy poprzez wywołanie funkcji **`BeginPaint`**, zwalniamy go poprzez wywołanie funkcji **`EndPaint`**. Konteksty urządzenia uzyskane w dowolny inny sposób, np. poprzez funkcję **`GetDC`**, zwalniamy funkcja **`ReleaseDC`**.

Jeżeli zapomnimy zwolnić niepotrzebny kontekst urządzenia, znaczyć to będzie, że w dziedzinie programowania jesteśmy amatorami. Jeżeli dopuścimy do wycieknięcia jakiegokolwiek innego zasobu, oznaczać to będzie to samo – musimy się jeszcze dużo, dużo uczyć!

Dużą pomocą w zarządzaniu zasobami Windows może być mechanizm konstruktor/destruktor języka C++ oraz inteligentne wykorzystanie wyjątków. Wymaga to opakowywania klasami języka C++ wszystkich funkcji API pobierających zasoby systemowe; klasy takie będą automatycznie zwalniać przydzielone im zasoby w destruktorach. Ale to zagadnienie na osobną książkę (którą już napisał Bartosz Milewski; http://www.relisoft.com/book).

Pierwszy rysunek

Możemy już przystąpić do wypełnienia obszaru roboczego okna nietrywialną treścią. Na początek wystarczy, że w programie omawianym w poprzednim rozdziale zmienimy kod obsługi komunikatu `WM_PAINT`. Efekt działania tak zmodyfikowanej aplikacji przedstawia rysunek 4.



Rysunek 4. Okienko programu *kolory*.

A oto kod obsługi komunikatu **WM_PAINT**:

```
case WM_PAINT:
{
  PAINTSTRUCT ps;
  RECT        rect;

  HDC hdc = BeginPaint (hwnd, &ps); // ← Zdobywamy kontekst urządzenia związanego z oknem hwnd
  Rectangle( hdc, 0, 0, 258, 258 ); // ← Rysujemy biały kwadrat z prostokątną obwódką

  GetClientRect (hwnd, &rect);        // ← Sprawdzamy, jaki jest rozmiar obszaru roboczego okna
  MoveToEx(hdc, 258, 258, 0);              // ← przesuwamy „kursor" w położenie (x=258, y=258)
  LineTo (hdc, rect.right, rect.bottom);  // ← i rysujemy linię do dolnego lewego narożnika okna

  // teraz pobieramy z kontekstu urządzenia kilka ciekawych informacji
  int technologia = GetDeviceCaps (hdc, TECHNOLOGY); // ← rodzaj urządzenia
  int r_x = GetDeviceCaps (hdc, HORZRES);             // ← rozdzielczość pozioma
  int r_y = GetDeviceCaps (hdc, VERTRES);             // ← rozdzielczość pozioma
  int b_c = GetDeviceCaps (hdc, BITSPIXEL);           // ← ilość bitów koloru na piksel

  const char* typ_urzadzenia [] =                     // ← tablica pomocnicza
        {
          „ploter", „monitor rastrowy",
          „drukarka rastrowa", „kamera rastrowa",
          „strumień znaków (PLP)", „meta-plik (VDM)", „display-file"
        };

  char bufor[512];                        // ← rezerwujemy miejsce na napis
  wsprintf(bufor,                         // ← tu funkcja wsprintf utworzy napis
             „%s, %d na %d pikseli (kolor: %d bitowy)", // ← definicja napisu
             typ_urzadzenia[technologia],  // ← to jest %s (typ: napis, czyli char*)
             r_x,                          // ← to jest pierwsze %d (typ: liczba całkowita)
             r_y,                          // ← to jest drugie %d (typ: liczba całkowita)
             b_c);                         // ← to jest trzecie %d (lyp: liczba całkowita)

  // wypisujemy ciekawą(?) wiadomość; strlen zwraca ilość znaków w napisie
  TextOut (hdc, 5, rect.bottom-22, bufor, strlen(bufor));
  //wypełniamy prostokąt pikselami o różnych barwach; x = czerwony, y = zielony
  for (int x = 0; x < 256; x++)
  {
     for (int y = 0; y < 256; y++)
     {
        int niebieski =  abs((x+y)/2-255); // ← abs zwraca wartość bezwzgledną
        // makrodefinicja RGB pobiera wartości 3 barw odstawowych: Red, Green i Blue, a zwraca COLORREF
        SetPixel( hdc, 1+x, 1+y, RGB(x, y, niebieski) );
     }
  }

  EndPaint (hwnd, &ps); // ← na koniec zawsze zwalniamy kontekst urządzednia !!!
  return 0;      // ← wartość 0 oznacza „hej Windows, nie martw się, ten komunikat przetworzyłem ja sam!"
}
```

Po uzyskaniu kontekstu urządzenia nasz program w lewej górnej części obszaru roboczego okna wyświetla prostokąt o rozmiarze 258 na 258 pikseli. W tym celu posługujemy się funkcją **Rectangle**:

```
Rectangle( hdc, 0, 0, 258, 258 );
```

Jako pierwszy argument pobiera ona kontekst urządzenia, w którym chcemy narysować prostokąt. Drugi i trzeci argument to, odpowiednio, współrzędne *x* i *y* lewego górnego wierzchołka rysowanego prostokąta. Kolejne dwa argumenty określają zaś współrzędne *x*, *y* jego prawego dolnego wierzchołka. Rozmiar prostokąta uwzględnia szerokość jego konturu (258 = 256 + 2*1 piksel).

Zwróćmy uwagę, że punktowi (0, 0) odpowiada lewy górny wierzchołek obszaru roboczego, a oś „y" skierowana jest *z góry do dołu*. Im niżej na ekranie położony jest piksel, tym większa jest wartość jego składowej *y*. Jest to ogólna cecha wszystkich funkcji interfejsu GDI – wszystkie one standardowo używają tego samego układu współrzędnych (istnieje sposób, by te ustawienia zmienić, jednak zagadnienia tego nie będę tu poruszał).

Następnie poprzez znaną już nam funkcję **GetClientRect** sprawdzamy, jaki jest bieżący rozmiar obszaru roboczego okna. Chodzi o to, że chcemy narysować linię prostą od prawego dolnego wierzchołka kwadratu do prawego dolnego narożnika obszaru roboczego okna. A położenie tego drugiego punktu może w każdej chwili ulec zmianie wskutek działań użytkownika.

Przystępujemy do kreślenia linii. Najpierw przesuwamy pióro do punktu (258, 258):

```
MoveToEx(hdc, 258, 258, 0);
```

Funkcja ta tradycyjnie w pierwszym argumencie wymaga podania kontekstu urządzenia. W dwóch kolejnych podajemy współrzędne punktu, nad który chcemy przesunąć pióro. Natomiast poprzez ostatni argument możemy uzyskać informację, gdzie pióro było umieszczone przed wykonaniem tej funkcji – nas jednak to nie obchodzi, co sygnalizujemy wpisaniem tu zera. Teraz przy pomocy funkcji **LineTo** możemy narysować linię:

```
LineTo(hdc, rect.right, rect.bottom);
```

Znaczenie argumentów tej funkcji jest już chyba oczywiste. Dodam tylko, że oprócz narysowania linii powoduje ona przesunięcie pióra do punktu końcowego linii.

Zauważmy, że funkcja **LineTo** nie wymaga podania punktu początkowego rysowanego odcinka. Współrzędne tego punktu są bowiem przechowywane w danym kontekście urządzenia jako „bieżące położenie pióra"; wartość tego parametru jest ustalana bądź modyfikowana pewnymi funkcjami GDI, np. **MoveToEx**, **LineTo**. Widzimy więc, że kontekst urządzenia rzeczywiście posiada pewien wewnętrzny stan. Dzięki temu nie musimy do każdej funkcji interfejsu GDI przekazywać za każdym razem wszystkich niezbędnych im informacji.

Funkcji rysujących linie jest znacznie więcej. Wymieńmy tu najważniejsze z nich:

- **Arc**, **ArcTo** (łuki elipsy);
- **LineTo** (linie);
- **PolyBezier**, **PolyBezierTo** (linie Béziera);
- **PolyDraw**, **Polyline**, **PolylineTo**, **PolyPolyline** (linie łamane)

Ale wróćmy do naszego programu. W kolejnych instrukcjach przy pomocy funkcji **GetDeviceCaps** usiłujemy dowiedzieć się, jakie są możliwości urządzenia związanego z bieżącym kontekstem urządzenia. Ilość informacji, które możemy w ten sposób uzyskać, jest ogromna. Ja ograniczyłem się do uzyskania danych na temat typu urządzenia (czyli czy jest to monitor, czy też może drukarka lub ploter), aktualnej rozdzielczości ekranu i na ilu bitach przechowywane są informacje o kolorze każdego piksela. Gorąco zachęcam Czytelnika do przejrzenia dokumentacji funkcji **GetDeviceCaps**.

Następnie rezerwuję pamięć na napis, który za chwilę wyświetlę w swoim okienku:

```
char bufor[512];
```

Teraz przy pomocy funkcji **wsprintf** wypełniam ten bufor znakami tworzącymi napis zawierający informacje uzyskane przed chwilą z funkcji **GetDeviceCaps**. Problem polega jednak na tym, że w

chwili pisania programu nie mogę przewidzieć, w jakim trybie karty graficznej będzie uruchomiony mój program, nie mogę więc z góry przewidzieć, jaką postać będzie miał mój napis (gdybym mógł, funkcja **GetDeviceCaps** nie byłaby mi do niczego potrzebna). Do takich zadań doskonale nadaje się funkcja **wsprintf** (będąca uproszczoną wersją standardowej funkcji języka C – `sprintf`).

```
wsprintf(bufor,                                // ← tu funkcja wsprintf utworzy napis
„%s, %d na %d pikseli (kolor: %d bitowy)”,    // ← definicja napisu
typ_urzadzenia[technologia],                   // ← to jest %s (typ: napis, czyli char*)
         r_x,                                  // ← to jest pierwsze %d (typ: liczba całkowita)
         r_y,                                  // ← to jest drugie %d (typ: liczba całkowita)
         b_c);                                 // ← to jest trzecie %d (lyp: liczba całkowita)
```

W pierwszym argumencie podajemy, gdzie funkcja **wsprintf** ma zapisać pożądany napis. W drugim argumencie podajemy tzw. format napisu. Wpisujemy tu po prostu cały nasz napis, tak jak chcielibyśmy go widzieć na ekranie, zastępując jednak w nim parametry nieznane podczas pisania programu specjalnymi dwuznakami rozpoczynającymi się od znaku `%` (procent). Druga litera każdego dwuznaku definiuje format danych, którymi ma on być zastąpiony. I tak `%s` oznacza „łańcuch znaków”, czyli napis, a `%d` oznacza „liczba naturalna”, czyli zmienną typu `int`. Następnie funkcji tej przekazujemy parametry, które mają zastąpić dwuznaki – ma być ich, oczywiście, dokładnie tyle, ile odpowiednich dwuznaków w drugim argumencie funkcji. Dostępnych jest *bardzo* dużo dwuznaków; zachęcam Czytelnika do zapoznania się z dokumentacją funkcji **wsprintf** (lub funkcji pokrewnych: `printf`, `fprintf` lub `sprintf`).

Teraz nadszedł czas, by wyświetlić na ekranie zawartość bufora. Zamiast poznanej już funkcji **DrawText**, tym razem używamy bardziej elastycznej funkcji **TextOut**. Oprócz kontekstu urządzenia pobiera ona współrzędne x i y początkowego punktu wyświetlanego napisu, napis oraz ilość znaków, jakie mają być wyświetlone:

```
TextOut (hdc, 5, rect.bottom - 22, bufor, strlen(bufor));
```

Ponieważ chciałem wyświetlić cały napis, do wyznaczenia liczby jego znaków użyłem standardową funkcję **strlen**.

Na koniec wnętrze kwadratu wypełniam pikselami o różnych, płynnie zmieniających się kolorach. W tym celu posługuję się funkcją **SetPixel**:

```
SetPixel (hdc, 1+x, 1+y, RGB(x, y, niebieski));
```

Oczywiście jej pierwszym argumentem jest kontekst urządzenia. Kolejne dwa określają współrzędne piksela, a ostatni definiuje jego nowy kolor. Kolor komponujemy z trzech barw podstawowych: czerwonej, zielonej i niebieskiej. Odpowiadają one kolorom trzech plamek luminoforu, z których składa się każdy piksel (na ekranie komputera można je dostrzec przez lupę, a gołym okiem – na ekranie telewizora). Wartością nasycenia każdej z tych barw może być dowolna liczba całkowita z przedziału 0..255. Zero odpowiada brakowi danej składowej koloru, a 255 – jej pełnemu nasyceniu. Okazuje się, że dzięki specyficznej budowie ludzkiego oka każdy inny kolor można traktować jako mieszaninę tych trzech barw podstawowych. Jako mieszalnik służy makrodefinicja **RGB**, do której przekazujemy kolejno nasycenie barwy czerwonej, zielonej i niebieskiej. Na przykład kolorowi czarnemu odpowiada `RGB(0,0,0)`, białemu – `RGB(255,255,255)`, a żółtemu – `RGB(255,255,0)`. Należy pamiętać, że jeżeli do makra **RGB** przekażemy argumenty spoza przedziału 0..255, tak naprawdę do definicji koloru zostaną użyte ich reszty z dzielenia przez 256. Innymi słowy `RGB(256,257,-1)` jest równoważne wyrażeniu `RGB(0,1,255)`.

Kod obsługi komunikatu **WM_PAINT** kończymy dwiema standardowymi instrukcjami. Pierwsza zwalnia kontekst urządzenia, druga informuje system o pomyślnym przetworzeniu komunikatu:

```
EndPaint (hwnd, &ps);
return 0;
```

Pióra, pędzle, czcionki…

W poprzednim programie nauczyliśmy się rysować punkty, linie i proste figury geometryczne, brakuje nam jednak wielu dodatkowych informacji. Czytelnik chciałby zapewne wiedzieć, w jaki sposób dostosować do własnych potrzeb kolor obwódki prostokąta. Ucieszy się też zapewne, gdy dowie się, że Windows potrafi rysować linie różnego rodzaju (np. kropkowane) i szerokości, a figury potrafi wypełnić nie tylko dowolną farbą, ale też wieloma standardowymi wzorkami (np. liniami poziomymi).

Pióra

Do rysowania linii Windows używa pióra (ang. *pen*). Każdy kontekst urządzenia przechowuje („jest właścicielem" dokładnie jednego pióra. Aby zmienić jakiś atrybut linii, np. jej kolor, rodzaj lub szerokość, należy:

A. Utworzyć pióro o pożądanych właściwościach.

B. Wstawić to pióro do kontekstu urządzenia (czyli *wybierać* pióro).

C. Używać go do woli.

D. Usunąć niepotrzebne już pióro z kontekstu urządzenia, zastępując go piórem uprzednio usuniętym z kontekstu urządzenia (w ten sposób przywrócimy pierwotny stan kontekstu urządzenia).

E. Jeżeli pióro jest nam niepotrzebne, ***powinniśmy je w sposób jawny usunąć z systemu***.

Prześledźmy szczegółowo każdy z tych kroków.

A. Stworzenie pióra

Nowe pióro tworzymy funkcją **CreatePen**:

```
HPEN moje_pioro = CreatePen(styl_piora, szerokosc_piora, kolor_piora);
```

Prototyp tej funkcji jest dość prosty:

```
HPEN CreatePen(
    int fnPenStyle,      // styl pióra
    int nWidth,          // szerokość pióra
    COLORREF crColor     // kolor pióra
);
```

Wartością tej funkcji jest uchwyt do nowego pióra (**HPEN**). Jako styl pióra można podać jeden z siedmiu parametrów: **PS_SOLID** (——), **PS_DASH** (- - -), **PS_DOT** (·····), **PS_DASHDOT** (- · - · - · - ·), **PS_DASHDOTDOT** (- · · - · · - · ·), **PS_NULL** (pióro „bezbarwne") i **PS_INSIDEFRAME**. Interpretacja szerokości pióra zależy od przyjętego układu współrzędnych; domyślnie podawana jest w pikselach. Natomiast kolor pióra definiowany jest poprzez znaną już nam strukturę **COLORREF**, którą zazwyczaj wypełnimy przy pomocy makrodefinicji **RGB**.

B. Wstawienie pióra do kontekstu urządzenia

Do umieszczania w kontekście urządzenia piór, pędzli, czcionek, regionów i map bitowych służy uniwersalna funkcja **SelectObject**. Oto typowy przykład jej użycia:

```
HPEN stare_pioro = (HPEN)SelectObject(hdc, moje_pioro);
```

Funkcja ta przyjmuje dwa parametry: uchwyt do kontekstu urządzenia (**HDC**) i uchwyt do wstawianego obiektu (w tym przypadku: do pióra, **HPEN**). Przekazywany w drugim argumencie obiekt zastępuje odpowiedni obiekt znajdujący się dotychczas w kontekście urządzania (np. nowe pióro zastępuje stare pióro, ale nie stary pędzel). Ten zastępowany obiekt zwracany jest na zewnątrz (w formie uchwytu) jako wartość funkcji. Uchwyt ten musimy przechwycić i przypisać zmiennej odpowiedniego typu. Ponieważ jednak funkcja **SelectObject** może równie dobrze zwrócić uchwyt do czcionki (**HFONT**) jak i do pióra (**HPEN**), programista musi jawnie podać typ wartości tej funkcji. W naszym przypadku funkcja podmienia pióra, zwraca więc obiekt typu **HPEN**, więc jej wartość modyfikujemy operatorem rzutowania na typ **HPEN**, który zapisujemy jako `(HPEN)`. Uchwyt do starego pióra jest nam niezbędny,

gdyż prawdopodobnie wiążą się z nim pewne zasoby systemowe, które prędzej czy później będziemy musieli zwolnić, a nie sposób tego zrobić bez dostępu do tego uchwytu.

C. Używanie pióra

To już potrafimy. Wystarczy posłużyć się dowolnymi funkcjami rysującymi linie, np. **Rectangle**, **Ellipse** lub **LineTo**.

D. Usuwanie pióra z kontekstu urządzenia

Ten etap też już znamy – pióro usuwamy dokładnie tak samo, jak je wstawiamy, tyle że teraz usuwane pióro pełni rolę „pióra starego".

```
SelectObject(hdc, stare_pioro);
```

Najczęściej usuwane pióro zastępujemy tym, którym je jakiś czas temu w kontekście urządzenia zastąpiliśmy. Taka strategia stanowi solidny fundament umożliwiający konstrukcję programów, w których zasoby nie wyciekają programistom między palcami.

E. Niszczenie pióra

Pióro jest zasobem. Za każdym razem, gdy tworzymy nowy zasób, uszczuplamy możliwości korzystania z tych zasobów przez inne programy działające równolegle z naszym. Dlatego gdy jakiegoś zasobu nie potrzebujemy, powinniśmy natychmiast go zwolnić, czyli oddać systemowi operacyjnemu (można to porównać do recyklingu surowców wtórnych). Zasoby wykorzystywane przez kontekst urządzenia niszczymy funkcją **DeleteObject**:

```
DeleteObject (moje_pioro);
```

Ta sama funkcja służy też do niszczenia naszych pędzli, czcionek, regionów i map bitowych.

Inne obiekty interfejsu GDI używane w kontekstach urządzenia

Pozostałymi obiektami – pędzlami, czcionkami, regionami i mapami bitowymi – posługujemy się dokładnie tak, jak piórami: tworzymy je, umieszczamy w kontekście urządzenia, posługujemy się nimi, wyjmujemy je z kontekstu urządzenia i na koniec je niszczymy. Jedyne różnice pojawiają się podczas tworzenia obiektów – każdemu z nich odpowiada inna funkcja tworząca.

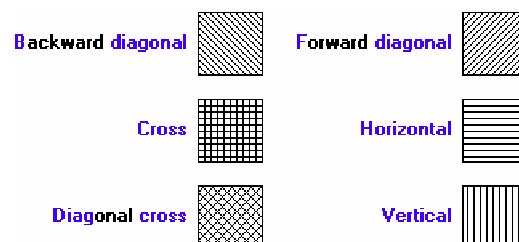
Nowy pędzel możemy utworzyć przy pomocy funkcji **CreateSolidBrush**:

```
HBRUSH moj_pedzel = CreateSolidBrush (kolor_pedzla);
```

Jest to bardzo prosta w użyciu funkcja, której jedynym argumentem jest kolor nowego pędzla. Utworzony za jej pomocą pędzel będzie zamalowywał obszary w sposób jednolity. Pewną alternatywą stanowi tu funkcja **CreateHatchBrush**,

```
HBRUSH moj_pedzel = CreateHatchBrush (styl_pedzla, kolor_pedzla);
```

która wypełnia powierzony jej obszar pewnym wzorkiem. Oprócz koloru pędzla przyjmuje ona jeszcze jeden argument, styl wzorku, który może być równy jednemu z sześciu parametrów: **HS_BDIAGONAL** (linie ukośne w dół), **HS_CROSS** (linie poziome i pionowe), **HS_DIAGCROSS** (przecinające się linie ukośne), **HS_FDIAGONAL** (linie ukośne w górę) **HS_HORIZONTAL** (linie poziome) i **HS_VERTICAL** (linie pionowe). Wzorki te ilustruje rysunek 5:



Rysunek 5. Wzorce wypełnień pędzla dostępne w funkcji **CreateSolidBrush**

Z kolei nowy krój czcionki można utworzyć przy pomocy funkcji `CreateFont`. Oto jej prototyp:

```
HFONT CreateFont(
  int nHeight,                  // logiczna wysokość czcionki
  int nWidth,                   // logiczna średnia szerokość znaku
  int nEscapement,              // kąt "ucieczki" wiersza tekstu (??)
  int nOrientation,             // kąt nachylenia linii bazowej
  int fnWeight,                 // stopień wytłuszczenia czcionki
  DWORD fdwItalic,              // flaga pochylenia czcionki (kursywy)
  DWORD fdwUnderline,           // flaga podkreślenia czcionki
  DWORD fdwStrikeOut,           // flaga przekreślenia czcionki
  DWORD fdwCharSet,             // identyfikator systemu kodowania znaków
  DWORD fdwOutputPrecision,     // dokładność na wyjściu
  DWORD fdwClipPrecision,       // dokładność przycinania
  DWORD fdwQuality,             // jakość na wyjściu
  DWORD fdwPitchAndFamily,      // skok i rodzina czcionki
  LPCTSTR lpszFace              // wskaźnik do nazwy kroju czcionki
);
```

A oto dość realistyczny przykład użycia tej funkcji w programie:

```
HFONT nowa_czcionka = CreateFont(-20, 0, 450, 0, FW_NORMAL, 0, 0,
      0, EASTEUROPE_CHARSET, OUT_DEFAULT_PRECIS,
      CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, FF_SCRIPT, 0 );
```

Instrukcja powyższa tworzy dowolną czcionkę (tę dowolność sygnalizuje ostatnie 0) spełniającą następujące warunki: jej wysokość równa jest 20 jednostek (pikseli), nachylona jest do osi „x" o 450/10 = 45 stopni, jest zgodna z systemem kodowania znaków obowiązującym w Europie Wschodniej (`EASTEUROPE_CHARSET`), kształt liter przypomina pismo odręczne (`FF_SCRIPT`), a poza tym wszystkie inne jej parametry mają wartości standardowe. Użycie liczby ujemnej jako pierwszego parametru tej funkcji nie jest błędem – wartość dodatnia też byłaby dobra, ale definiowałaby nieco inną czcionkę (ta subtelna różnica jest opisana w instrukcji *online* kompilatora).

Funkcja `CreateFont` posiada ogromną ilość parametrów, jednak nie powinno nas to do niej zniechęcać. Większość z nich ma zastosowanie przede wszystkim w profesjonalnych wydrukach na papierze.

I to już wszystko. Zostało nam co prawda do omówienia tworzenie regionów i – co szczególnie ważne – map bitowych, ale to temat na oddzielny wykład. Kto wie, może kiedyś wrócę do tego zagadnienia…

Programy przykładowe

1. Program `kolory` to pierwszy z omawianych w tym rozdziale programów. Demonstruje sposoby wyświetlania pojedynczych pikseli, odcinków i całych figur geometrycznych, np. prostokątów lub kół.
2. Program `GDI` ilustruje omawiane tu sposoby posługiwania się podstawowymi obiektami interfejsu GDI: piórami, pędzlami i czcionkami.

Podsumowanie

- Naszym łącznikiem ze światem monitorów, drukarek, ploterów i innych „wyświetlaczy" jest interfejs GDI. Nadzoruje on wszystkie operacje graficzne i uniezależnia nas od fizycznej charakterystyki wykorzystywanych urządzeń.
- Naszym kluczem do interfejsu GDI są tzw. konteksty urządzeń (DC, *Device Contexts*). Każda funkcja interfejsu GDI odpowiedzialna za narysowanie czegokolwiek na ekranie czy drukarce wymaga podania jej uchwytu do kontekstu urządzenia (`HDC`).
- Kontekst urządzenia związany z danym oknem pobieramy z systemu albo poprzez funkcję `BeginPaint`, albo poprzez funkcję `GetDC`. Pierwszą z nich stosujemy tylko i wyłącznie w kodzie obsługi komunikatu `WM_PAINT`.

- Niepotrzebny kontekst urządzenia **<u>MUSIMY</u>** natychmiast zwolnić. Konteksty utworzone funkcją `BeginPaint` zwalniamy funkcją `EndPaint`, a utworzone funkcją `GetDC` – przez wywołanie funkcji `ReleaseDC`.
- Kontekst urządzenia przechowuje swój stan. Zapisane są w nim m.in. informacje na temat rodzaju, koloru i położenia pióra rysującego linie, rodzaju i koloru pędzla, koloru tekstu, koloru tła tekstu, parametrów bieżącej czcionki, mapy bitowej i tzw. regionu przycinania.
- Aby w kontekście urządzenia zmienić pióro, pędzel, czcionkę, mapę bitową lub region należy:
  - Utworzyć nowe pióro, pędzel, czcionkę… (`CreatePen`, `CreateSolidBrush`, `CreateFont`…).
  - Wstawić je do kontekstu urządzenia funkcją `SelectObject`, przechwytując jednocześnie uchwyt do wyjmowanego obiektu.
  - Używać nowy obiekt w danym kontekście urządzenia.
  - Wyjąć ten obiekt z kontekstu urządzenia, a na jego miejsce wstawić obiekt „stary" (funkcją `SelectObject`).
  - Niepotrzebne pióro czy pędzel natychmiast zniszczyć funkcją `DeleteObject`.

  Pamiętajmy, że do zmiany koloru czcionki służy osobna funkcja `SetTextColor`, a do zmiany koloru tła – funkcja `SetBkColor`.
- Uwaga: uchwyty do kilku standardowych piór i pędzli można także uzyskać poprzez funkcję `GetStockObject`; takich obiektów nie musimy usuwać funkcją `DeleteObject`.
- Oto kilka popularnych funkcji interfejsu GDI: `Rectangle`, `Ellipse`, `MoveToEx`, `LineTo`, `SetPixel`, `GetPixel`.
- Kolor przechowywany jest w strukturze o nazwie `COLORREF`. Obiekty tego typu najłatwiej jest tworzyć za pośrednictwem makrodefinicji `RGB`, która pobiera trzy parametry, określające kolejno nasycenia trzech barw podstawowych ekranu: czerwonej, zielonej i niebieskiej.
- Wartości parametrów używanych w makrodefinicji `RGB` powinny mieścić się w zakresie 0,..,255. W przeciwnym wypadku system użyje resztę ich dzielenia przez 256 (czyli np. 256 jest równoważne 0).
- Standardowo punkt o współrzędnej (0,0) mieści się w lewym górnym narożniku obszaru roboczego okna.
- Podstawowych informacji o danym kontekście urządzenia może nam dostarczyć funkcja `GetDeviveCaps`.
- Zdecydowanie warto zrozumieć i polubić funkcję `wsprintf`.
- Tytuł okienka można łatwo zmodyfikować funkcją `SetWindowText` (przykład jej użycia znajduje się drugim programie przykładowym).

Zadania

1. Uruchom drugi program przykładowy (`GDI`) i zaobserwuj, jak się on zachowuje w sytuacjach typowych dla okienek Windows: gdy zasłaniamy i odsłaniamy go innym okienkiem, gdy je minimalizujemy i maksymalizujemy, gdy najeżdżamy kursorem myszki nad znajdujące się na pasku tytułowym kwadraciki służące do zamykania lub minimalizacji okna (co powoduje wyświetlenie okienka podpowiedzi), gdy otwieramy menu systemowe. Jakie wnioski można stąd wyciągnąć na temat generowania przez system komunikatu **`WM_PAINT`**?
2. Po wykonaniu poprzedniego zadnia Czytelnik powinien słusznie skonstatować, że kontekst urządzenia otrzymany z funkcji **`BeginPaint`** w cudowny sposób ogranicza nam możliwość rysowania w oknie do minimalnego obszaru, który uległ modyfikacji. Ma to znaczenie dla zwiększenia prędkości obsługi okienek. Napisz program, który w pasku tytułowym będzie wyświetlał bieżące położenie wierzchołków „prostokąta przycinania", w którym mamy prawo cokolwiek narysować. Jego współrzędne odczytasz ze składowej `rcPaint` struktury typu **`PAINTSTRUCT`**, której adres przekazywany jest w drugim argumencie funkcji **`BeginPaint`**.
3. Napisz program, który wyświetli pięć zachodzących na siebie kół olimpijskich.
4. „Pobaw się" funkcją **`SetTextAlign`**, która określa sposób wyrównywania tekstu (np. zmodyfikuj program przykładowy, wywołując funkcję **`SetTextAlign`** z opcją **`TA_LEFT`**, **`TA_RIGHT`**, **`TA_CENTER`**, **`TA_BOOTOM`**).
5. Napisz program, który narysuje planszę do gry w warcaby.
6. Zmodyfikuj program z punktu 3) tak, by wyświetlić bierki w położeniu początkowym gry. (Bierki możesz reprezentować w postaci kół).

4. Obsługa komunikatów użytkownika

Przedstawione w poprzednim rozdziale programy powinny budzić u Czytelnika zarówno uczucie niedosytu jak i niepokoju. Źródłem niedosytu jest brak możliwości realnego wpływu na zachowanie się tych programów – jak dotąd nie potrafimy bowiem obsługiwać klawiatury ani myszki. Z kolei uczucie niepokoju powinno towarzyszyć konstatacji, że zastosowana przez nas technika – wykonywanie pewnych nietrywialnych operacji w ramach obsługi komunikatu **WM_PAINT** (np. generowanie nowych figur geometrycznych lub zmiana kroju czcionki) – nie jest najszczęśliwszym pomysłem. Sęk w tym, że otrzymywany poprzez funkcję **BeginPaint** kontekst urządzenia ma wbudowane informacje o dopuszczalnym obszarze odświeżania okna, przy czym obszar ten rzadko kiedy obejmuje cały obszar roboczy okna – dlatego ten kontekst urządzenia nie nadaje się do odświeżania *całego* okna!

Nasz niepokój jest w pełni uzasadniony – projektanci komunikatu **WM_PAINT** mieli na myśli bardzo specyficzny zakres jego użyteczności: komunikat ten wysyłany jest do okna w chwili, gdy *system operacyjny* dochodzi do wniosku, że w wyniku normalnego posługiwania się różnymi okienkami Windows należy odświeżyć konkretny fragment konkretnego okienka. Fragment i tylko fragment – ograniczenie obszaru odświeżania okienka do dobrze zdefiniowanego obszaru znacznie przyspiesza bowiem wyświetlanie zawartości *wszystkich* innych okien, dając użytkownikowi złudzenie, że proces obsługi nawet kilkunastu okienek naraz nie wiąże się praktycznie z żadnym narzutem czasowym. To dlatego właśnie używana w kodzie obsługi komunikatu **WM_PAINT** funkcja **BeginPaint** zwraca nie tylko uchwyt do kontekstu urządzenia, ale też i pewne dodatkowe informacje dotyczące m.in. obszaru, do którego powinny ograniczyć się bieżące operacje graficzne. Programista nie musi tych informacji uwzględniać – w razie czego system i tak obetnie wszystkie kreślone przez niego elipsy, prostokąty czy linie do „obszaru dozwolonego”. Ale jeśli się te informacje uwzględni, będzie można znacznie przyspieszyć obsługę komunikatu **WM_PAINT**, co jest ze wszech miar pożądane!

Tak więc kod obsługi komunikatu **WM_PAINT** nie powinien w programie niczego zmieniać, lecz ograniczać się do wyświetlania aktualnego *stanu programu*. Jedno proste zdanie: *każdy program okienkowy posiada swój dobrze określony stan*, jest kluczem do zrozumienia filozofii tworzenia aplikacji okienkowych. Jak dobrze wiemy, takie program są niemal z definicji programami interaktywnymi: oczekują, że użytkownik będzie im przekazywał pewne polecenia zmieniające ich wewnętrzny stan. Zmiana ta może (ale nie musi) wiązać się z koniecznością zmodyfikowania wyglądu okienka aplikacji. Jednak najważniejsze jest to, że po przetworzeniu danego polecenia program przechodzi w stan uśpienia. Jeśli nie liczyć kodu obsługi pętli komunikatów w funkcji **WinMain**, program nie robi dosłownie nic. Nie obciąża procesora wykonywaniem niepotrzebnych czynności. Program *pasywnie*[3] czeka na kolejne polecenie, które w języku Windows zwie się *komunikatem*. Dzięki temu inne aplikacje mogą pracować pełną parą.

Prosta aplikacja interaktywna – gra „snake”

Przedstawione tu idee spróbuję zilustrować na prostym przykładzie praktycznym. Opiszę kod imitujący popularną grę *snake* (znaną z systemu MS-DOS i telefonów komórkowych).

Specyfikacja gry

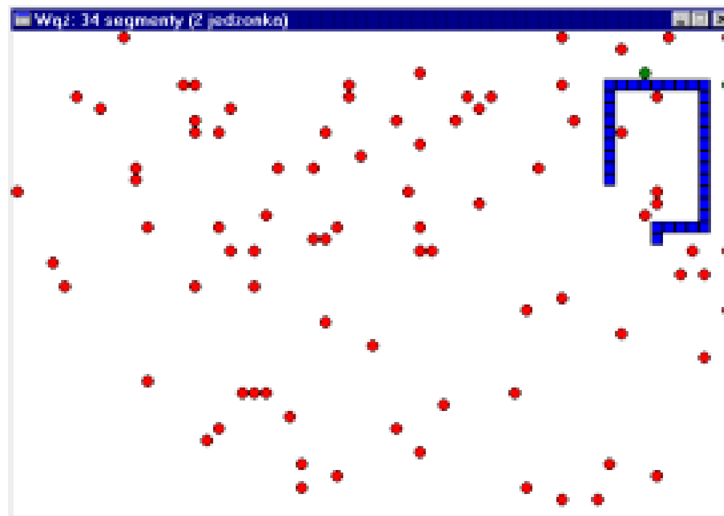
W naszej wersji gra toczy się na prostokątnej planszy składającej się z 61×41 identycznych, kwadratowych pól. Na planszy tej w chwili początkowej umieszczamy na różnych, losowo wybranych polach 61 min (zwanych też bombami) i 8 pudełek z jedzeniem. Po planszy porusza się wąż. Jego głowa może poruszać się w jednym z czterech kierunków równoległych do boków planszy. Na planszy obowiązują periodyczne warunki brzegowe: np. po przekroczeniu górnego rzędu planszy głowa węża pojawia się w rzędzie dolnym (innymi słowy plansza topologicznie równoważna jest torusowi). Gdy wąż wejdzie na pole, na którym znajduje się pożywienie, długość jego ciała wzrasta o jeden segment (każdy

[3] Ta pasywność wiąże się bezpośrednio z tym, że funkcje obsługi okien są deklarowane z atrybutem **CALLBACK**: to nie nasz program odpytuje Windows, czy ma dla niego nowy komunikat – to Windows uruchamia tę procedurę w chwili, gdy jakiś komunikat ma być przesłany do obsługiwanego przez nią okienka.

segment zajmuje dokładnie jedno pole planszy). W trakcie gry na planszy w losowo wybranych, wolnych miejscach systematycznie pojawiają się zarówno nowe pudełka z pożywieniem jak i kolejne bomby. Gracz wygrywa, jeżeli wąż zdąży zjeść całe pożywienia; gracz przegrywa, jeżeli wąż wpełźnie na minę lub gdy jego głowa wejdzie na pole zajmowane przez inny segment węża.

Specyfikacja interfejsu użytkownika

Typowy wygląd ekranu tej gry przedstawia rysunek 6.



Rysunek 6. Zrzut ekranu programu *snake*

Wąż wyświetlany jest jako układ niebieskich kwadracików, miny – jako czerwone kółka, a pożywienie – w postaci kółek zielonych. Użytkownik może sterować ruchem głowy węża przy pomocy czterech klawiszy ↑,→,↓ i ←. Dodatkowo w każdej chwili może zatrzymać grę klawiszem Esc; jego ponowne wciśnięcie spowoduje kontynuowanie gry. Podczas gry na pasku tytułowym wyświetlana jest aktualna długość węża i ilość pól z jedzeniem. Każdą rozgrywkę rozpoczyna wyświetlenie „strony informacyjnej"; informuje ona o położeniu węża, bomb i pożywienia (dzięki temu gracz może nadać wężowi pożądany kierunek początkowy), a na pasku tytułowym – informację o autorze. Grę rozpoczyna przyciśnięcie klawisza Esc.

Podczas gry ukrywa się kursor myszy, żeby nie zasłaniał planszy.

Specyfikacja stanu programu, stałych globalnych i nowych typów zmiennych

Z punktu widzenia programisty stan układu składa się z kilku zmiennych i tablic. Dodatkowo, aby uprościć kod i przygotować go na ewentualne modyfikacje w przyszłości, w programie używa się kilku stałych oraz własnych typów danych.

Stałe globalne

W programie korzystam z 10 stałych całkowitych:

```
const int x_size = 61;                            // poziomy rozmiar planszy (nieparzysty!)
const int y_size = 41;                            // pionowy rozmiar planszy (nieparzysty!)
const int x_center = x_tab_size/2;                // skladowa x środka planszy
const int y_center = y_tab_size/2;                // skladowa y środka planszy
const int pix_bok = 10;                           // szerokość pola planszy w pikselach
const int client_x = x_tab_size * pix_bok;        // szerokość obszaru roboczego okna (piksele)
const int client_y = y_tab_size * pix_bok;        // wysokość obszaru roboczego okna (piksele)
const int init_jedzonka = 8;                      // ile jedzonek rozkładamy na początku
const int czas_odswiezania = 80;                  // czas trwania jednego ruchu (w milisekundach)
const int bomb_frequency = 20;                    // częstotliwość dostawiania bomb/jedzonek
```

Parametry `x_size` i `y_size` określają szerokość i wysokość planszy. Powinny być liczbami nieparzystymi, gdyż tylko wtedy dwa parametry pomocnicze `x_center` i `y_center` określają położenie środka planszy. Parametr `pix_bok` definiuje liniowy rozmiar pola planszy wyrażony w pikselach.

Parametry `client_x` i `client_y` definiują szerokość i wysokość obszaru roboczego okienka (a więc i planszy) wyrażoną w pikselach. Parametr `init_jedzonka` określa, na ilu polach planszy mamy początkowo rozmieścić pożywienie dla węża; `czas_odswiezania` definiuje czas (w milisekundach), jaki zajmuje wężowi pokonanie jednego pola planszy, a parametr `bomb_frequency` określa, jak często w układzie mają pojawiać się nowe bomby lub pojemniki z pożywieniem.

Dodatkowo jako stałe parametry definiuję cztery pędzle o różnych kolorach:

```
const HBRUSH pedzel_zablokowany = CreateSolidBrush(RGB(255,0,0)); // bomba
const HBRUSH pedzel_zajety = CreateSolidBrush(RGB(0,0,255));       // wąż
const HBRUSH pedzel_smaczny = CreateSolidBrush(RGB(0,128,0));      // jedzenie
const HBRUSH pedzel_tla = (HBRUSH)GetStockObject(WHITE_BRUSH);     // tlo
```

Nowe typy danych

Plansza składa się z punktów, które przechowujemy w strukturze `Punkt` składającej się z dwóch współrzędnych całkowitych `x` i `y`:

```
struct Punkt
{
  int x;
  int y;
};
```

Dodatkowo wprowadzam dwa wyliczenia o oczywistej interpretacji:

```
enum EStan  {Pusty = 0, Zajety = 1, Zablokowany = 2, Smaczny = 3}; // stan pola
enum EKierunek {Polnoc, Wschod, Poludnie, Zachod};       // kierunek ruchu głowy
```

Zmienne globalne, czyli stan układu

Pełny stan programu opisany jest w trzech tablicach i siedmiu zmiennych globalnych:

```
EStan plansza[x_size][y_size];        // plansza do gry – opisuje stan każdego pola
Punkt snake[x_size * y_size];         // położenie wszystkich segmentów węża
Punkt przeszkody[x_size * y_size];    // połozenie wszystkich bomb i jedzonek („przeszkód")

int ile_segmentow;         // z ilu segmentów składa się wąż?
int ile_przeszkod;         // ile jest na planszy bomb lub pól z pożywieniem
int ile_jedzonka;          // na ilu polach znajduje się pożywienie?
EKierunek kierunek;        // kierunek ruchu głowy węża
bool pause;                // jeśli true, to wstrzymujemy ruch węża
bool ekran_powitalny;      // jeśli true, to wyświetlamy ekran powitalny, jeśli false, to gramy
```

Tablica `plansza` przechowuje informacje o stanie każdego pola: czy jest ono puste, zajęte przez bombę, jedzenie czy węża. Tablice `snake` i `przeszkody` mają charakter pomocniczy – zawierają położenia kolejnych pól planszy zajętych przez węża (`snake`) i bomby lub pożywienia (`przeszkody`). Oczywiście dane zawarte w tych tablicach powielają informacje już zawarte w tablicy `plansza`; tablice te służą do optymalizacji procesu aktualizacji planszy po wykonaniu ruchu.

Tablice `snake` i `przeszkody` są jednowymiarowe, przy czym ich rozmiar zadeklarowałem asekurancko jako równy ilości pól planszy. Oczywiście wystarczy to, by pomieścić dowolną ilość bomb, pól z pożywieniem i segmentów węża. Rzeczywista liczba elementów przechowywanych w tych tablicach zapisana jest w zmiennych `ile_segmentow` i `ile_przeszkod`.

Dodatkowo w zmiennej `ile_jedzonka` przechowywana jest informacja o bieżącej ilości pól z pożywieniem (liczbę tę systematycznie wyświetlamy na pasku tytułowym programu), zmienna `kierunek` zawiera zaś informację o aktualnym kierunku ruchu głowy węża. Używam jeszcze dwóch pomocniczych zmiennych logicznych, `pause` i `ekran_powitalny`, które służą wyłącznie do poprawnego wyświetlania okienka na ekranie. Pierwsza z nich informuje komputer, czy użytkownik zażyczył sobie wstrzymania gry (np. naciskając klawisz Esc), druga natomiast jest flagą sygnalizującą, że rozpoczyna się nowa gra i należy wyświetlić odpowiednie napisy powitalne.

Rozruch programu

Postać użytej w grze *snake* funkcji **WinMain** tylko nieznacznie różni się od tych, które widzieliśmy w poprzednich przykładach. Zasadnicza różnica polega na tym, że obecnie chcemy otworzyć okno o z góry zadanej wielkości obszaru roboczego, którego nie będzie można zmienić w trakcie gry. Pożądaną szerokością obszaru roboczego jest iloczyn liczby pól planszy w kierunku poziomym i szerokości każdego pola wyrażonej w pikselach, czyli `client_x = x_size * pix_bok`. Analogicznie pożądana wysokość obszaru roboczego wynosi `client_y = y_size * pix_bok`. Jednak funkcja **CreateWindowEx** wymaga podania jej szerokości i wysokości całego okna, a więc uwzględnienia miejsca zajmowanego przez jego brzeg, pasek tytułowy i być może inne elementy, np. menu. Na szczęście istnieje specjalna funkcja, **AdjustWindowRectEx**, która wyznacza rozmiar całego okna na podstawie pożądanej wielkości jego obszaru roboczego. Funkcja ta wymaga podania czterech argumentów:

```
BOOL AdjustWindowRectEx(
  LPRECT lpRect,      // adres prostokąta zawierającego rozmiary pożądanego obszaru roboczego okna
  DWORD dwStyle,      // styl okna (używany jako czwarty parametr funkcji CreateWindowEx)
  BOOL bMenu,         // czy okno będzie miało menu?
  DWORD dwExStyle     // tzw. rozszerzony styl okna (używany jako pierwszy parametr funkcji CreateWindowEx)
);
```

Po wywołaniu tej funkcji w prostokącie wskazywanym przez parametr `lpRect` umieszczone zostaną współrzędne całego okna, które można już przekazać jako argumenty do funkcji **CreateWindowEx**.

Ustalanie początkowego położenia okna przeprowadzam w osobnej funkcji `FindInitRect`[4]:

```
void FindInitRect(RECT & rect, int flagi, int flagiEx, BOOL jestMenu)
{
   int max_x = GetSystemMetrics(SM_CXSCREEN);
   int max_y = GetSystemMetrics(SM_CYSCREEN);

   int dx = (max_x - client_x)/2;
   int dy = (max_y - client_y)/2;

   rect.left = dx;
   rect.top = dy;
   rect.right = max_x - dx;
   rect.bottom = max_y - dy;
   AdjustWindowRectEx (&rect, flagi, jestMenu, flagiEx);
}
```

W funkcji tej ustalam nie tylko rozmiary okienka roboczego, ale i wymuszam jego pojawienie się dokładnie na środku ekranu. W tym celu przy pomocy funkcji **GetSystemMetrics** pobieram z systemu informacje o bieżącej rozdzielczości ekranu (poziomej i pionowej); reszta jest chyba jasna.

Powyższą funkcję pomocniczą wywołuję po odpowiednim przygotowaniu prostokąta, zawierającego pożądane współrzędne obszaru roboczego okna:

```
RECT rect;
const DWORD flagi = WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX;
const DWORD flagi_ex = 0;
const bool jest_menu = false;
FindInitRect (rect, flagi, flagi_ex, jest_menu);
```

Otrzymane w ten sposób parametry okna wstawiamy następnie do znanej już nam funkcji **CreateWindowEx**:

```
CreateWindowEx (flagi_ex,  nazwa_klasy_okien, "Wąż", flagi,
  rect.left, rect.top, rect.right - rect.left, rect.bottom - rect.top,
  NULL, NULL, hInstance, NULL);
```

---

[4] Widać tu moją niekonsekwencję w doborze nazw zmiennych i funkcji: raz posługuję się językiem polskim, raz angielskim. Cóż, po prostu najczęściej wybieram wersję krótszą…

Zwróćmy jeszcze uwagę na zestaw użytych flag: **WS_CAPTION** | **WS_SYSMENU** | **WS_MINIMIZEBOX**. Gwarantuje on nam, że okno będzie zupełnie „normalnym” oknem Windows z menu systemowym i przyciskami minimalizacji i usuwania okna, ale nie dające użytkownikowi możliwości zmiany jego rozmiaru.

I jeszcze jedna, drobna nowość: pod koniec programu, po zakończeniu działania pętli komunikatów, usuwam wszystkie zasoby przydzielone zmiennym globalnym. W naszym przypadku są to trzy pędzle (`pedzel_zablokowany`, `pedzel_zajety` i `pedzel_smaczny`).

Specyfikacja obsługiwanych komunikatów i implementacja procedury okna

Jak już wiemy, każde okienko komunikuje się ze światem zewnętrznym poprzez system komunikatów. Spośród kilkuset dostępnych komunikatów zazwyczaj decydujemy się na obsługę kilkunastu najlepiej spełniających nasze potrzeby.

W przypadku programu *snake* zakładamy, że będzie on obsługiwać następujące komunikaty:

- **WM_CREATE**. Nasze okienko otrzymuje ten komunikat w momencie, gdy jest tworzone, a jeszcze nie jest wyświetlane na ekranie. „Wysyła go” funkcja **CreateWindowEx**.
- **WM_TIMER**. System operacyjny wysyła ten komunikat do okienka w z góry przez nas określonych odstępach czasu. Obsługa tego komunikatu to podstawowy mechanizm implementowania wszelkiego rodzaju „animacji” wykonywanych w czasie rzeczywistym.
- **WM_KEYDOWN**. Ten komunikat sygnalizuje, że użytkownik przycisnął pewien klawisz klawiatury.
- **WM_PAINT**. Ten komunikat już znamy: jego pojawienie się oznacza, że powinniśmy odświeżyć wygląd okienka na ekranie.
- **WM_DESTROY**. Komunikat otrzymujemy w trakcie niszczenia okna, gdy jego zagłada jest nieuchronna.
- **WM_CLOSE**. Użytkownik wyraził chęć zamknięcia okna.

Oto krótki opis implementacji obsługi tych sześciu komunikatów w programie *snake*.

Komunikat **WM_CREATE**

W kodzie obsługi tego komunikatu zazwyczaj umieszcza się instrukcje służące do zainicjowania stanu okienka. W naszym przypadku robimy tu dwie rzeczy: po pierwsze, inicjujemy stan planszy (w tym celu wywołujemy osobną funkcję `NowaGra`); po drugie, tworzymy tzw. zegar Windows.

Funkcja `NowaGra` nadaje rozsądne wartości wszystkim zmiennym globalnym, w których, jak wiemy, przechowywany jest aktualny stan gry. Funkcję tę wywołujemy za każdym razem, gdy zachodzi potrzeba rozpoczęcia gry od nowa. Jej kod znajduje się w programie przykładowym i nie wymaga komentarza. Wyjątkiem są dwie ostatnie instrukcje wykorzystujące funkcje interfejsu API:

```
InvalidateRect (hwnd, 0, true); // unieważniamy wygląd całego okienka
UpdateWindow (hwnd);            // żądamy, by okienko zostało natychmiast odmalowane
```

Pierwsza z nich unieważnia fragment okienka roboczego; parametry tego fragmentu przekazuje się w drugim argumencie. Jeżeli wpiszemy tu 0, oznaczać to będzie żądanie odświeżenia całego obszaru roboczego okna. Z kolei poprzez trzeci argument informujemy system, czy przed odświeżeniem okienka powinien zamalować okienko robocze domyślnym pędzlem aplikacji.

Funkcja **InvalidateRect** co prawda generuje komunikat **WM_PAINT**, ale wstawia go do kolejki komunikatów, gdzie będzie on obsłużony w ostatniej kolejności. Teoretycznie może to doprowadzić do niepożądanych efektów (program może „widzieć” co innego niż użytkownik). Dlatego wywołujemy funkcję **UpdateWindow**, która powoduje natychmiastowe przetworzenie komunikatu **WM_PAINT**, z pominięciem kolejki komunikatów.

Po ustaleniu początkowego stanu programu uruchamiamy *zegar Windows*. W tym celu wywołujemy funkcję **SetTimer**:

```
SetTimer (hwnd, numer_zegara, czas_odswiezania, 0);
```

Pierwszym argumentem tej funkcji jest uchwyt okna do którego mają być wysyłane komunikaty **WM_TIMER** generowane przez nowotworzony zegar. Takich zegarów można utworzyć kilka, a rozróż-

niamy je poprzez identyfikatory nadawane podczas ich tworzenia. Identyfikatory są liczbami całkowitymi przekazywanymi w drugim argumencie funkcji `SetTimer`. W jej trzecim argumencie podajemy, jak często system ma generować komunikat `WM_TIMER`. Czas ten określamy w milisekundach. W programie *snake* wstawiliśmy tu globalną stałą symboliczną `czas_odswiezania` o wartości 80; dlatego spodziewamy się, że nasze okno będzie otrzymywać komunikaty `WM_PAINT` co ok. 80/1000 = 0.08 sekundy (innymi słowy, ok. 12 razy na sekundę). Jeżeli w ostatnim parametrze funkcji `SetTimer` wstawimy 0, system operacyjny będzie kierował komunikaty `WM_PAINT` pochodzące z danego zegara do procedury okna wskazanego w pierwszym argumencie tej funkcji (`hwnd`); niezerowa wartość czwartego argumentu umożliwia zastosowanie innego mechanizmu obsługi zegara, którego tu omawiać jednak nie będę.

Komunikat `WM_DESTROY`

O ile komunikat `WM_CREATE` służy do inicjalizacji okienka, komunikat `WM_DESTROY` daje nam okazję do „posprzątania" po zamykanym okienku. Zasada jest prosta: w kodzie obsługi komunikatu `WM_DESTROY` zwalniamy wszystkie zasoby przydzielone wcześniej podczas przetwarzania komunikatu `WM_CREATE`. Dlatego kody obsługi obu tych komunikatów warto umieszczać obok siebie (można je też porównać do pary konstruktor/destruktor obiektu w języku C++). W naszym przykładzie mamy do czynienia z jednym takim zasobem: zegarem Windows. Usuwamy go z systemu specjalną funkcją `KillTimer`:

```
KillTimer (hwnd, numer_zegara);
```

Funkcji tej przekazujemy dwa oczywiste argumenty: uchwyt okna, z którym skojarzony był dany zegar, oraz identyfikator niszczonego zegara.

Komunikat `WM_PAINT`

Jak już wiemy, obsługę tego komunikatu należy sprowadzić do odświeżenia aktualnego wyglądu planszy. Ponieważ nie jest to już takie proste, oddelegowałem to zadanie do specjalnej funkcji `OdrysujPlansze (HDC hdc)`. Pobiera ona jeden, za to absolutnie niezbędny parametr: uchwyt kontekstu urządzenia, w którym chcemy rysować planszę. Z kolei funkcja ta przekazuje zadanie odświeżenia planszy do innych funkcji pomocniczych, `RysujSegment` i `RysujPrzeszkody`:

```
void OdrysujPlansze(HDC hdc)
{
  for (int i = 0; i < ile_segmentow; i++)
    RysujSegment(hdc, snake[i].x, snake[i].y);
  for (int j = 0; j < ile_przeszkod; j++)
    RysujPrzeszkody(hdc, przeszkody[j].x, przeszkody[j].y);
}
```

Pierwsza z nich rysuje na ekranie segment węża jako kwadracik wypełniony odpowiednim pędzlem i obrysowany domyślnym piórem (tu: piórem czarnym o szerokości 1 piksela).

```
void RysujSegment(HDC hdc, int x, int y)
{
  int x0 = x*pix_bok;     // współrzędna „x" lewego górnego wierzchołka pola
  int y0 = y*pix_bok;     // współrzędna „y" lewego górnego wierzchołka pola
  HBRUSH stary_pedzel = (HBRUSH) SelectObject(hdc, pedzel_zajety);
  Rectangle (hdc, x0, y0, x0 + pix_bok, y0 + pix_bok);
  SelectObject(hdc, stary_pedzel);
}
```

Druga funkcja wygląda bardzo podobnie; rysuje przeszkodę jako koło w odpowiednim kolorze zależnie od tego, czy „przeszkodą" jest pożywienie czy bomba.

Na koniec zwróćmy jeszcze uwagę na to, że implementacja funkcji `OdrysujPlansze` zakłada, że przed jej wywołaniem ktoś (prawdopodobnie system) wymazał całą zawartość okienka roboczego.

Komunikat `WM_TIMER`

To bodaj najważniejszy komunikat programu. Jego pojawienie się sygnalizuje bowiem, że prawdopodobnie powinniśmy przesunąć węża do następnego położenia i być może dostawić bombę lub pudełko

z pożywieniem. Celowo użyłem słowa „prawdopodobnie”, gdyż może się okazać, że gra jest na chwilę zatrzymana – np. wskutek naciśnięcia przez użytkownika klawisza Esc lub zakończenia bieżącej partii. Informacja o tym, czy gra jest zatrzymana, znajduje się w globalnej zmiennej logicznej `pause`. Dlatego obsługę komunikatu **`WM_TIMER`** rozpoczynamy prostym testem:

```
if (pause)
  return 0;    // Zwrócenie 0 oznacza całkowite, pomyślne zakończenie przetwarzania komunikatu
```

Dalsza część instrukcji wykonywana będzie tylko wtedy, gdy zmienna `pause` ma wartość `false`, tj. gdy wąż ma się poruszać.

Zanim przesuniemy węża, zapamiętujemy położenie jego ogona: za chwilę opuści on przecież swoje położenie, będziemy więc zaraz musieli na ekranie narysować w tym miejscu puste pole:

```
Punkt ogon = snake[0];
```

Następnie przesuwamy węża do następnego położenia. W tym celu wywołujemy funkcję `RuszWeza`

```
bool gramy_dalej = RuszWeza();
```

która nie tylko przesuwa węża, ale też zwraca informację, czy aby gra nie uległa zakończeniu (na skutek najechania głowy węża na bombę lub inny segment węża). Jeżeli gra uległa zakończeniu, zapisujemy w zmiennej `pause` wartość `true`, co efektywnie zatrzyma przetwarzanie komunikatów **`WM_TIMER`**. Następnie przy pomocy funkcji **`MessageBox`** pytamy użytkownika, czy chce rozpocząć nową rozgrywkę, czy też zakończyć działanie program. Zależnie od uzyskanej odpowiedzi wywołujemy funkcję `NowaGra` lub znaną już funkcję interfejsu Win32 API, **`PostQuitMessage`**:

```
if (!gramy_dalej)
{
  WstrzymajGre();     // ← tu ustawiamy pause = true;
  int odpowiedz =  MessageBox(hwnd, "Nowa gra?", "Przegrałeś" ,
                     MB_YESNO | MB_ICONEXCLAMATION);
  if (odpowiedz == IDYES)
    NowaGra(hwnd);
  else
    PostQuitMessage(0);
}
```

Teraz aktualizujemy zawartość paska tytułowego okna (na którym wyświetlamy długość węża i liczbę pól z pożywieniem). W tym celu wywołujemy funkcję `AktualizujTytul`, która ustala nowy tekst na pasku tytułowym przy pomocy funkcji systemowej **`SetWindowText`**.

Teraz aktualizujemy wygląd planszy:

```
Punkt glowa = snake[ile_segmentow - 1];
HDC hdc = GetDC(hwnd);
RysujSegment (hdc, glowa.x, glowa.y);
WyczyscPole (hdc, ogon.x, ogon.y);
```

Jednocześnie sprawdzamy, czy aby nie należy dostawić na planszy kolejnej przeszkody. Ponieważ chcemy, by proces ten sprawiał wrażenie zupełnie losowego zarówno w sensie przestrzennym, jak i czasowym, przeszkodę stawiamy nie w równych odstępach czasu, lecz tylko z prawdopodobieństwem `1.0/bomb_frequency`:

```
if ( Random(bomb_frequency) == 0 ) // Random(n) generuje liczbę pseudolosową z przedziału 0,..,n-1
  GenerujPrzeszkode(hdc);
```

Procedura `GenerujPrzeszkode` generuje z jednakowym prawdopodobieństwem albo bombę, albo pole z pożywieniem i wyświetla ją na urządzeniu `hdc`; uaktualnia też globalne zmienne `ile_przeszkod`, `ile_jedzonka` oraz tablice `plansza` i `przeszkody`.

Teraz możemy już zwolnić kontekst urządzenia:

```
ReleaseDC(hwnd, hdc);
```

i sprawdzić, czy aby gra nie zakończyła się zwycięstwem użytkownika:

```
if (ile_jedzonka == 0) // jeżeli gracz wygrał, to...
{
  WstrzymajGre();
  int wynik = MessageBox(hwnd,
    "Wygraleś! Kolejna gra?", "Wygrałeś!", MB_ICONEXCLAMATION | MB_YESNO);
  if (wynik == IDYES)
    NowaGra(hwnd);
  else
    PostQuitMessage(0);
}
```

Komunikat WM_KEYDOWN

Otrzymanie komunikatu WM_KEYDOWN oznacza, że użytkownik przycisnął jakiś klawisz klawiatury. Ale skąd mamy się dowiedzieć, który? Odpowiedź jest prosta: kod przyciśniętego klawisza przekazywany jest do procedury okna wraz z komunikatem WM_KEYDOWN w towarzyszącym mu parametrze wParam. Jest to bardzo ogólna zasada, towarzysząca przetwarzaniu zdecydowanej większości komunikatów: szczegółowe informacje na temat danego komunikatu przekazywane są w parametrach wParam i lParam.

Ponieważ chcemy, by nasz program reagował wyłącznie na przyciskanie klawiszy ↑,→,↓ i ← i Esc, najprostszy kod obsługi komunikatu WM_KEYDOWN może wyglądać następująco:

```
case WM_KEYDOWN:
{
  switch( int(wParam) )
  {
    case VK_LEFT:
       kierunek = Zachod;   break;
    case VK_RIGHT:
       kierunek = Wschod;   break;
    case VK_DOWN:
       kierunek = Poludnie; break;
    case VK_UP:
       kierunek = Polnoc;   break;
    case VK_ESCAPE:
      {
        pause = !pause;            // Esc jest przełącznikiem Graj/ZatrzymajGrę
        ekran_powitalny = false;
        ShowCursor(pause);         // chowamy lub pokazujemy kursor myszy
      }
  }
  return 0;
}
```

Jako kod klawisza najlepiej jest podać stałą symboliczną o nazwie zaczynającej się od przedrostka VK_, np. VK_Z to kod klawisza Z, VK_F11 to kod klawisza F11, a VK_SHIFT to kod klawisza Shift. Pełny wykaz kodów klawiszy wirtualnych znajduje się w pliku nagłówkowym winuser.h, a ich opis w systemie pomocy *online* kompilatora.

Warto jeszcze skomentować ostatnią instrukcję powyższego kodu:

```
ShowCursor(pause);
```

Użyta tu funkcja ShowCursor chowa (jeżeli uruchomiono ją z argumentem false) lub pokazuje (jeżeli wywołano ją z argumentem true) kursor myszy. Kursor myszy chowamy po to, by nie zakłócał obrazu planszy podczas gry, w której przecież i tak nie wykorzystujemy myszy. Kursor ten wyświetlamy jednak zawsze, gdy tylko gra z jakichkolwiek powodów ulegnie przerwaniu.

Programy przykładowe

1. Program snake.cpp zawiera kod źródłowy opisanej w tym rozdziale gry.

Podsumowanie

- Konstrukcja programów przeznaczonych dla systemu okienkowego (np. Windows) zasadniczo różni się od konstrukcji typowego programu uruchamianego z konsoli: o ile ten drugi wykonuje się w sposób nieprzerwany, cały czas angażując procesor, programy okienkowe z zasady większość czasu spędzają w uśpieniu.
- Z tego stanu hibernacji od czasu do czasu wyprowadza je system operacyjny, uruchamiając odpowiednią procedurę okna wraz ze stosownymi parametrami: uchwytem okna, numerem komunikatu i dwoma parametrami pomocniczymi.
- Program reaguje na otrzymanie komunikatu, przeprowadzając pewne obliczenia, które prowadzą do zmiany jego *stanu*.
- Stan układu to zestaw zmiennych niezbędnych do prawidłowego przetworzenia dowolnego komunikatu, co obejmuje informacje niezbędne do poprawnego wyświetlenia obszaru roboczego każdego okienka. W niewielkich programach najprościej stan układu zapisać w zmiennych globalnych.
- Cała sztuka polega na tym, by dobrze zdefiniować: *co to jest stan układu* i jak ma się ona zmieniać w odpowiedzi na *dowolny komunikat*.
- Należy wybrać stosunkowo niewielki zestaw komunikatów, które mogą zmienić stan układu, i zaimplementować ich obsługę; resztę obsługujemy standardową funkcją **DefWindowProc**. Trzeba uważać, by te nie obsługiwane przez nas komunikaty faktycznie nie zmieniały stanu układu (np. rezygnując z obsługi komunikatu **WM_SIZE** możemy przegapić zmianę rozmiaru okienka przez użytkownika).
- Zmienne reprezentujące stan okienka inicjujemy w kodzie obsługi komunikatu **WM_CREATE**; kod obsługi komunikatu **WM_DESTROY** traktujemy jak destruktor okna.
- Do obsługi klawiatury służą komunikaty **WM_KEYDOWN**, **WM_KEYUP** i – przede wszystkim – **WM_CHAR**. Kod klawisza i pewne dodatkowe informacje można odczytać z parametrów `lParam` i `wParam` procedury okna.
- Przy pomocy funkcji **SetTimer** program może tworzyć tzw. zegary systemowe, które następnie będą go budzić w regularnych odstępach czasu komunikatem **WM_TIMER**.
- Zegary są zasobami systemowymi: gdy przestają być potrzebne, usuwamy je funkcją **KillTimer**.
- Czytaj dokumentację! Ucz się angielskiego!

Zadania

1. Rozwiń grę wg swoich upodobań. Oto kilka pomysłów:
   - Wąż może początkowo składać się z większej ilości segmentów, a zjedzenie pożywienia może go wydłużać o więcej niż jeden segment.
   - Przydałoby się dodać „poważny” ekran powitalny, zastępujący jego aktualną namiastkę na pasku tytułowym.
   - Obecnie przeszkody losowane są w zupełnie przypadkowych miejscach, z wykluczeniem jedynie pól, na których już coś się znajduje. Wskutek tego możliwe jest wylosowanie bomby tuż przed głową węża, gracz nie ma wtedy żadnych szans. Może warto wyeliminować taką możliwość?
   - Zamiast kończyć grę, po wygraniu rozgrywki można przejść do nowej gry *na wyższym poziomie*. Trzeba jednak zdefiniować, czym charakteryzuje się ten „wyższy poziom”. Prędkością ruchu? Długością węża? Częstotliwością dostawiania przeszkód?
   - Byłoby miło, gdyby głowę wyświetlać inaczej, niż pozostałe segmenty węża.
2. Proszę wczytać się w kod i w nim poeksperymentować: np. wziąć pewne instrukcje w komentarz lub zmienić wartości stałych, po czym spróbować przewidzieć efekt tego zabiegu, skompilować tak zmodyfikowany program i skonfrontować jego działanie ze swoimi oczekiwaniami.

Dodatek A
Kompilacja programów napisanych w Windows API

Programy napisane dla systemu Windows muszą być kompilowane w specjalny sposób. Wynika to choćby z tego, że ich wykonywanie nie rozpoczyna się od funkcji `main`, lecz od `WinMain`. Jeżeli więc Twój kompilator (a właściwie jego moduł zwany konsolidatorem, ang. *linker*) wyświetla komunikat typu

```
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland
Error: Unresolved external '_main' referenced from
C:\BORLAND\BCC55\LIB\C0X32.OBJ
```

powodem jest to, że usiłujesz skompilować program napisany dla systemu Windows tak, jak „zwyczajny" program w języku C lub C++. Poprawny sposób kompilowania programów okienkowych zależy od tego, czy posługujemy się zintegrowanym systemem programistycznym, czy też programy kompilujemy, wydając polecenia z konsoli (lub umieszczając je w pliku `makefile`).

Zintegrowane środowiska programistyczne

W przypadku zintegrowanych środowisk programistycznych, czyli „kombajnów" łączących w sobie edytor programu, kompilator, debugger, profiler i inne cudeńka, z reguły kompiluje się nie programy, lecz *projekty*. Projekty (ang. *projects*) zawierają definicje wszystkich czynności, jakie musi wykonać zintegrowane środowisko programistyczne, aby doprowadzić do wygenerowania kodu wykonywalnego programu. Definicja projektu obejmuje więc m.in. informacje o tym, czego projekt dotyczy (programu wykonywalnego, biblioteki DLL itp.), z jakich plików się składa, czy jest programem dla systemu Windows czy DOS, czy ma być optymalizowany ze względu na prędkość wykonywania, minimalizację rozmiaru czy weryfikację jego poprawności, z jakich dodatkowych bibliotek korzysta itp., itd. W każdym systemie zintegrowanym tworzenie projektów przebiega inaczej, dlatego zawsze należy zapoznać się ze stosowną dokumentacją.

Środowisko MS Visual C++ 6.0

W środowisku MS Visual C++ 6.0 projekty tworzy się następująco:

1. Z menu wybieramy New (lub naciskamy Ctrl-N).
2. Pojawia się okno New. Wybieramy w nim zakładkę Projects, a w niej właściwą platformę. W przypadku programów przeznaczonych dla systemu Windows jest to Win32 Application (natomiast projekty napisane w „czystym" C/C++ deklarujemy jako Win32 Console Application).
3. W polu Project name wpisujemy nazwę projektu, a w polu Location – katalog, w którym standardowo będą umieszczane wszystkie składniki projektu. Przyciskamy OK.
4. Pojawia się okno Win32 Application – Step 1 of 1. Zaznaczamy w nim pole An empty project. Klikamy Finish, a w kolejnym oknie – OK.

Co prawda projekt jest już gotowy, ale nie ma w nim plików! Pliki dołączamy do projektów na jeden z 2 sposobów:

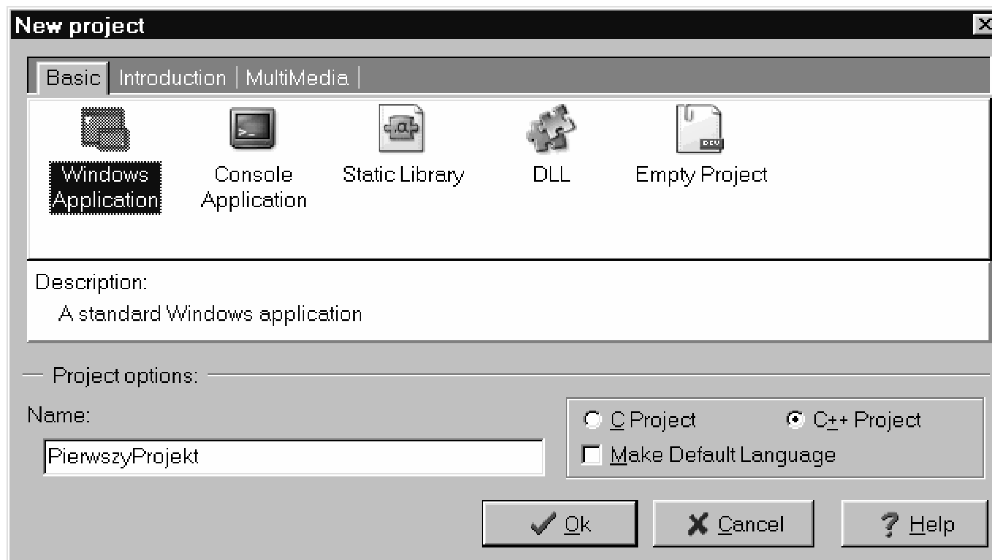
- Z menu wybieramy Project | Add to project | Files.
- Otwieramy w edytorze plik z kodem źródłowym i przyciskamy w jego obszarze prawy klawisz myszki. Pojawi się menu kontekstowe, w którym wybieramy opcję Insert File Into Project.

Gotowe projekty zapisywane są w plikach `*.dsp`. Jednak aby otworzyć gotowy projekt, należy w Eksploratorze Windows kliknąć dwukrotnie plik `*.dsw` przechowujący dodatkowo informacje o stanie tzw. przestrzeni roboczej programu (*workspace*). Dlatego do każdego programu przykładowego towarzyszącemu niniejszemu skryptowi dołączyłem odpowiednie pliki `dsw` i `dsp`.

Środowisko edytora Dev C++

W środowisku Dev C++ mamy do wyboru 4 predefiniowane typy projektów: Windows Application, Console Application, Static Library i DLL. Bez trudu można dołączyć tu nowe, własne rodzaje pro-

jektów. W przypadku programów okienkowych wybieramy oczywiście Windows Application – w tym celu z menu wybieramy New | Project i zakładkę Basic. Pojawi się okienko jak na rysunku A1. W polu Name wpisujemy nazwę naszego nowego projektu i klikamy ikonkę Windows Application. Upewniamy się jeszcze, że zaznaczono przycisk radiowy C++ Project i klikamy OK.



Rysunek A1. Okno *New Project* edytora Dev-C++.

W tym momencie pojawi się okienko Save File, w którym wybieramy nazwę i lokalizację pliku `*.dev` zawierającego definicję projektu. Po zamknięciu tego okienka wyświetli się główne okno edytora. Okazuje się, że Dev-C++ automatycznie wygenerował dla nas plik zawierający podstawowy kod programu okienkowego. Musimy ten plik jeszcze zapisać (być może pod zmienioną nazwą) i możemy przystąpić do tworzenia programu.

Raz utworzony projekt otwieramy, klikając dwukrotnie plik `*.dev`.

Kompilacja z poziomu wiersza poleceń

Każdy kompilator posiada specjalną opcję przełączającą go w tryb kompilacji programów dla Windows:

- Kompilator `bcc32` (*Borland Builder*): stosowna opcja to `-tW`. Tak więc program `WinMain.cpp` skompilujemy poleceniem `bcc32 -tW WinMain.cpp`, w wyniku czego otrzymamy plik wykonywalny `WinMain.exe`.
- Kompilator `MinGW` (dołączany do środowiska Dev-C++): stosowna opcja to `–mwindows`. Tak więc program `WinMain.cpp` skompilujemy poleceniem `g++ -mwindows WinMain.cpp -o WinMain.exe`. W wyniku tej operacji otrzymamy plik wykonywalny `WinMain.exe`.

Oczywiście kompilacja programów z poziomu wiersza poleceń wymaga ustawienia odpowiedniej wartości ścieżki programów wykonywalnych (`PATH`) i skonfigurowania kompilatora. W przypadku kompilatora MinGW warto zainstalować pakiet `MSYS`, który przemienia konsolę systemu DOS w konsolę linuksową (`rvterm + bash`). Opis sposobu konfigurowania kompilatora Borlanda znajduje się (w przypadku standardowej instalacji) w pliku `c:\Borland\Bcc55\readme.txt` w rozdziale *Installing and running the Command Line Tools*.

Programy okienkowe składają się zazwyczaj nie tylko z kodu napisanego w języku programowania, np. C++, lecz i z tzw. **skryptu zasobów** definiującego zasoby programu, np. menu, ikony, szablony okien dialogowych. Skrypt ten musi być kompilowany osobnym programem (w przypadku kompilatora MinGW jest to program `windres.exe`), a następnie łączony z innymi modułami przy pomocy

standardowego konsolidatora wywoływanego pośrednio przez program `g++.exe`. Tak, zasoby programu umieszczane są w pliku wykonywalnym!

Dlatego jeśli program składa się z pliku `prog.cpp` i skryptu zasobów `res.rc`, kompilujemy go w trzech krokach, które mogą wyglądać następująco:

```
g++ -c prog.cpp –o prog.o -I"C:/DEV-CPP/include"
windres -i res.rc -o res.o
g++ -o prog.exe prog.o res.o -L"C:/DEV-CPP/lib" -mwindows
```

Nie wygląda to szczególnie zachęcająco. Z tego powodu w przypadku programów składających się z wielu plików kompilację przeprowadza się najczęściej przy pomocy polecenia `make`, które odczytuje instrukcje dotyczące kompilacji z pliku `makefile`. Instrukcje te zawierają informacje o tym, jakie programy, w jakiej kolejności i z jakimi opcjami należy uruchomić, aby wygenerować plik wykonywalny. Ten właśnie sposób wykorzystywany jest w środowisku Dev-C++, które najpierw generuje plik `Makefile.win`, a następnie kompiluje program poleceniem `make –f Makefile.win all`.

Indeks

AdjustWindowRectEx ...37
API ...3
Arc ...27
ArcTo ...27

BeginPaint ...19
BUTTON ...20, 21

CALLBACK ...4, 12
COLORREF ...32
COMBOBOX ...20, 21
CreateProcess ...7
CreateWindowEx ...11, 16, 37
CS_DBLCLKS ...15
CS_HREDRAW ...15
CS_NOCLOSE ...15
CS_VREDRAW ...15
CW_USEDEFAULT ...16

DefWindowProc ...11, 18, 20
DispatchMessage ...12, 17
DrawText ...20
DT_CENTER ...20
DT_SINGLELINE ...20
DT_VCENTER ...20

EDIT ...20
EndPaint ...19
ExitWindowsEx ...9

GetClientRect ...27
GetDC ...24
GetDeviceCaps ...27
GetMessage ...17
GetStockObject ...15, 32
GetSystemMetrics ...37

HBRUSH ...15, 30
HFONT ...31
HINSTANCE ...3
HPEN ...29

IDABORT ...6
IDCANCEL ...6
IDIGNORE ...6
IDNO ...6
IDOK ...6
IDRETRY ...6
IDYES ...6
interfejs GDI ...23
InvalidateRect ...38

KillTimer ...39
klasa okien ...11
kolejka komunikatów ...12
kontekst urządzenia ...23

LineTo ...27
LISTBOX ...20, 21
LoadCursor ...15
LoadIcon ...15
lParam ...18
LPCSTR ...3
LPINT ...3
LPSTR ...3
LRESULT ...4, 17

MDICLIENT ...20
MessageBoxEx ...5
MoveToEx ...27

PAINTSTRUCT ...19
pętla komunikatów ...15, 17
PolyBezier ...27
PolyBezierTo ...27
PolyDraw ...27
Polyline ...27
PolylineTo ...27
PolyPolyline ...27
PostMessage ...12
PostQuitMessage ...12, 20
procedura okna ...12, 17

Rectangle ...27
RegisterClass ...11
RegisterClassEx ...15
ReleaseDC ...24
RGB ...28
RichEdit ...20
RICHEDIT_CLASS ...20

SCROLLBAR ...20
SelectObject ...30
SendMessage ...12
SetPixel ...28
SetTimer ...38, 42
SetWindowText ...32
ShowCursor ...41
ShowWindow ...11
STATIC ...20
STRICT ...3

TranslateMessage ...17

uchwyt ...3
UINT ...3
UpdateWindow ...11, 38

WINAPI ...4
windows.h ...3
WinMain ...3, 4
WM_CLOSE ...22, 38
WM_CREATE ...38
WM_DESTROY ...20
WM_KEYDOWN ...38
WM_PAINT ...18
WM_QUIT ...12
WM_TIMER ...38, 40
WNDCLASSEX ...15
wParam ...18
WS_CAPTION ...16
WS_CHILD ...16
WS_HSCROLL ...16
WS_MAXIMIZEBOX ...16
WS_MINIMIZEBOX ...16
WS_OVERLAPPED ...16
WS_OVERLAPPEDWINDOW 16
WS_SIZEBOX ...16
WS_SYSMENU ...16
WS_VSCROLL ...16
wsprintf ...7, 27
wystąpienie (*instance*) ...4

zegar Windows ...38