Spis treści

CZĘŚĆ I PROGRAMOWANIE STRUKTURALNE — UZUPEŁNIENIA I DEFINICJE 7

Rozdział 1. Instalacja Turbo Pascala 9

Rozdział 2. Kopiowanie przykładów w C++. 13

Rozdział 3. System dwójkowy, ósemkowy, szesnastkowy 16

3.1. Operacje bitowe 18

3.1.1. Przesunięcia. 18

3.1.2. Funkcje logiczne: AND &, OR |, XOR ^, negacja NOT ~, dopełnienie NEG. . 19

3.2. Prawa logiki i prawa de Morgana 20

Rozdział 4. Języki programowania — pojęcia podstawowe 22

Rozdział 5. Schematy blokowe algorytmów. 23

5.1. Definicje — rodzaje bloków. 23

5.2. Opracowanie programów — kolejność czynności 24

5.3. Kompilacja programów. 24

Rozdział 6. Elementy Pascala 26

6.1. Konstrukcja programu w Pascalu. 26

6.2. Typy danych 26

6.2.1. Typy proste i porządkowe 27

6.2.2. Typy strukturalne 28

6.2.3. Definiowanie typów danych użytkownika/programisty. 29

6.3. Stałe i zmienne — deklarowanie i inicjowanie 29
6.3.1. Deklaracje zmiennych 29
6.3.2. Deklarowanie i inicjowanie stałych i zmiennych 29

6.4. Deklarowanie i inicjowanie stałych i zmiennych strukturalnych (operator kropki) 30

6.5. Operatory Pascala 31

6.6. Wyrażenia 31

6.7. Operacje na zbiorach (teoria mnogości). .32

6.8. Instrukcje Pascala 32
6.8.1. Instrukcja warunkowa If-Then-Else (Else jest opcjonalne) 32
6.8.2. Instrukcja wyboru wielokrotnego Case 33
6.8.3. Instrukcja For 33
6.8.4. Instrukcje While i Repeat 34

Rozdział 7. Funkcje i procedury 35

7.1. Obsługa standardowego wejścia-wyjścia 35

7.2. Definicja procedury. 35

7.3. Definicja funkcji 35

Rozdział 8. Obsługa wejścia-wyjścia. Operacje plikowe. 36

8.1. Opis zmiennej plikowej 36

8.2. Skojarzenie zmiennej z fizycznym plikiem dyskowym, otwarcie pliku 36

8.3. Operacje na danych pliku dyskowego (odczyt, zapis) 37

Rozdział 9. C — przydatne definicje 39

9.1. Struktury i unie 39
9.1.1. Zmienne typu strukturalnego i wskaźniki do struktur. 39
9.1.2. Inicjowanie tablicy (wektora) struktur, autoreferencja 39
9.1.3. Unie 40

9.2. Obsługa plików dyskowych 40

9.3. Dynamiczne struktury danych — kolejka, lista 41

9.4. Formatowanie — funkcje printf() i scanf() 43

programowanie_strukturalne_DODATEK.indb 2 2010-07-05 11:54:40

CZĘŚĆ II PASCAL I C/C++ — ROZDZIAŁY DODATKOWE 45

Rozdział 1. Jak w Windows XP uruchamiać aplikacje tekstowe? 47

Rozdział 2. Pascal — trzy sposoby komputerowej animacji 50

2.1. Animacja — sposób 1 50

2.2. Animacja — sposób 2 53

2.3. Animacja — sposób 3 56

Rozdział 3. Pascal — komputerowe modelowanie. 62

3.1. Kiedy działa i jak działa statystyka? 62

3.2. Paradoks kwadransa akademickiego 65

3.2.1. Fragmentaryczne testowanie hipotezy 66

3.2.2. Algorytm wizualizacji 66

3.3. Problem spotkania — uogólnienie. 69

3.3.1. Sposób rozwiązania: analityczne zliczanie bez wizualizacji 70

3.3.2. Kod komputerowej symulacji w Pascalu 71

3.3.3. Komputery nie mylą się w liczeniu. 72

3.4. Sformułowanie zadania — wersja druga 73

3.4.1. Metoda wizualizacji prawdopodobieństwa 73

3.4.2. Oznaczenia — wyjściowe parametry iteracji 74

3.4.3. Algorytm pętli iteracyjnych. 75

3.5. Czy cierpliwość popłaca? 77

Rozdział 4. Język C — kody powstające w wyniku asemblacji 81

4.1. Kod asemblera generowany przez Borland C++ 83

Rozdział 5. Maszynowa reprezentacja liczb całkowitych 85

programowanie_strukturalne_DODATEK.indb 3 2010-07-05 11:54:40

CZĘŚĆ III ĆWICZENIA I ZADANIA. 89

Rozdział 1. Elementy Pascala 91

1.1. Ćwiczenia początkowe, środowisko tekstowe 91

1.2. Ćwiczenia początkowe, środowisko graficzne 92

1.3. Pytania powtórzeniowe 93

Rozdział 2. Elementy C i C++ 95

2.1. Podstawy programowania w C i C++, pytania powtórzeniowe . .95

2.2. Instrukcje sterujące — pytania 97

2.3. Programowanie strukturalne — zadania. 105

CZĘŚĆ IV DELPHI 113

Rozdział 1. Rozszerzenia i specyfika języka Object Pascal w Delphi 115

1.1. Tablice otwarte 115

1.2. Zmienna Result używana do zwrotu wartości przez funkcje . . .117

1.3. Typ wartości zwracanej przez funkcję .117

1.4. Przekazanie argumentów przez wartość lub poprzez referencję 118

Rozdział 2. Przenoszenie programów Turbo Pascala do Delphi 120

2.1. Aplikacje pracujące w trybie tekstowym. 125

2.2. Aplikacje konsoli w Delphi. 128

2.3. Aplikacje konsoli tworzone za pomocą kreatora 131

Rozdział 3. O konwersji typów danych 133

3.1. Własne funkcje konwersji typów. 133

3.2. Funkcje konwersji typów w Delphi 135

programowanie_strukturalne_DODATEK.indb 4 2010-07-05 11:54:40

CZĘŚĆ V C++ — ZAAWANSOWANE PROGRAMOWANIE OBIEKTOWE 139

Rozdział 1. O referencjach w C++. 141

1.1. Referencje w roli argumentów funkcji .144

1.2. Funkcje mogą zwracać i wskaźnik, i referencję 149

1.3. Wskaźnik specjalny this 152

Rozdział 2. Funkcje wirtualne i klasy abstrakcyjne 153

2.1. Funkcje wirtualne 153

2.2. Klasy abstrakcyjne 157

Rozdział 3. Polimorfizm — przykład zastosowania FIFO i LIFO 161

3.1. Klasy abstrakcyjne raz jeszcze 161

3.2. Funkcje w pełni i nie w pełni wirtualne .162

3.3. Sekwencje FIFO i LIFO, czyli o specyfikacji algorytmu 163

3.4. Obsługa stosu. 165

- **3.4.1.** Dziedziczenie klasy pochodnej Stack 166
- **3.4.2.** Zdejmowanie bajtów ze stosu 166
- **3.4.3.** Podglądanie bajtów na szczycie stosu 167
- **3.4.4.** Konstruktor klasy Stack a konstruktor klasy bazowej 168

3.5. Kolejka 171

- **3.5.1.** Przeglądanie zawartości kolejki 172

Rozdział 4. Przykład dziedziczenia od klasy bazowej ios 174

4.1. Funkcje-metody formatujące 174

4.2. Manipulowanie danymi numerycznymi .176

4.3. Manipulatory 180

Rozdział 5. Operacje dyskowe obiektowo 183

5.1. Obiektowe operacje na plikach 183

5.2. Obiektowe bazy danych 185

- **5.2.1.** Projekt bazy danych. 185

5.3. Przykład przeciążenia operatora << 197

programowanie_strukturalne_DODATEK.indb 5 2010-07-05 11:54:41

CZĘŚĆ VI O METODOLOGII METODOLOGII PROGRAMOWANIA 201

Rozdział 1. O algorytmach i stylach programowania . . . 204

1.1. Zdefiniowanie problemu 205

1.2. Rozwiązanie 1 — prosty program sekwencyjny 205

1.3. Rozwiązanie 2 — kod strukturalny 206

1.4. Rozwiązanie 3 — kod obiektowy. 208

1.5. Rozwiązanie 4 — kod zdarzeniowy 210

Rozdział 2. Visual C++, szablony i obsługa wyjątków . . 214

2.1. Konstruowanie i stosowanie szablonów 218

Rozdział 3. Można zastosować JavaScript 222

programowanie_strukturalne_DODATEK.indb 6 2010-07-05 11:54:41

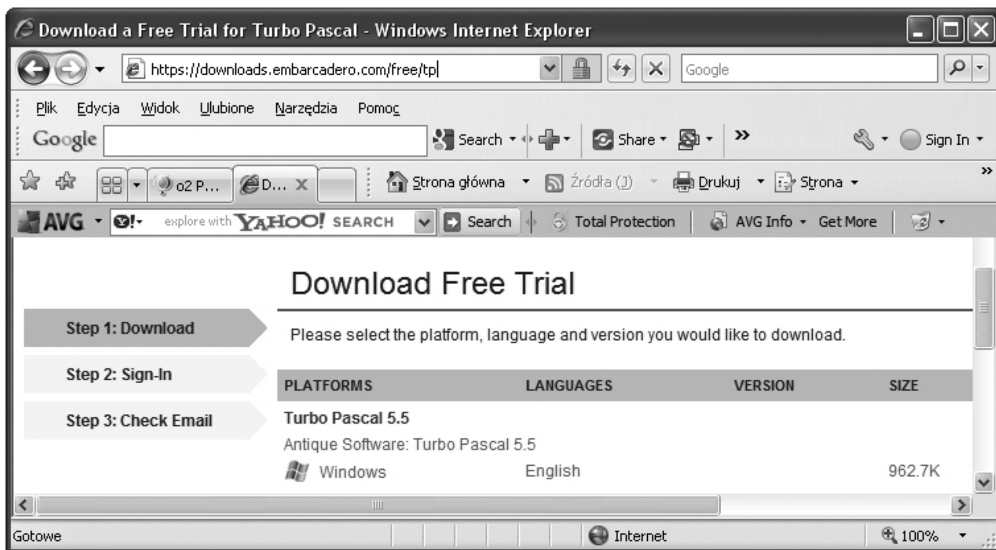
Część I

Programowanie strukturalne — uzupełnienia i definicje

1

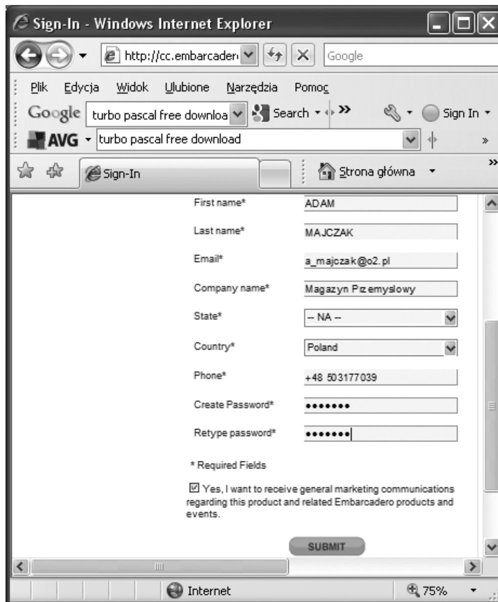
Instalacja Turbo Pascala

Najpierw wyszukujemy stronę umożliwiającą bezpłatne pobranie kompilatora, na przykład *https://downloads.embarcadero.com/free/tp* (rysunek D1). Skompresowany plik instalacyjny *TP55.ZIP* zostanie skopiowany do wybranego foldera na lokalnym dysku (tu: *D:\TP55INSTALL*).



Rysunek D1. Strona, z której można bezpłatnie pobrać kompilator. Aby go zapisać na dysku, należy wybrać przycisk Zapisz

Teraz należy wypełnić formularz rejestracji (rysunek D2) i kliknąć przycisk *Submit* (z ang. wyślij, przedłóż).

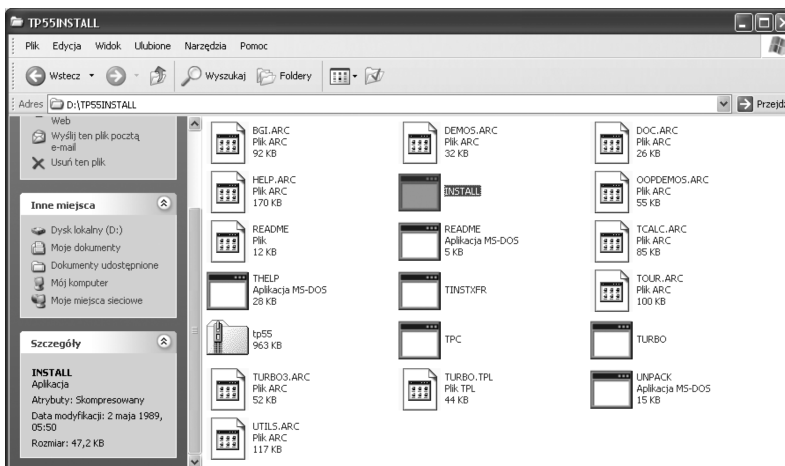


Rysunek D2.
Trzeba wypełnić wszystkie pola oznaczone gwiazdką

Na wskazany adres poczty elektronicznej zostanie wysłany plik tekstowy, który należy zapisać we własnym katalogu systemowym, na przykład:

- Windows 2000/2003/XP: *C:\Documents and Settings\<nazwa_uzytkownika>*
- Windows Vista: *C:\Users\<nazwa_uzytkownika>*
- Linux: */user/< nazwa_uzytkownika > lub /home/< nazwa_uzytkownika>*

W wiadomości e-mail, oprócz załączonego pliku o nazwie nadanej według szablonu *reg740.txt* (numery są różne dla różnych narzędzi), otrzymasz dwa klucze, które trzeba podać podczas instalacji. Kolejne etapy instalacji pokazane zostały na rysunkach D3-D5.



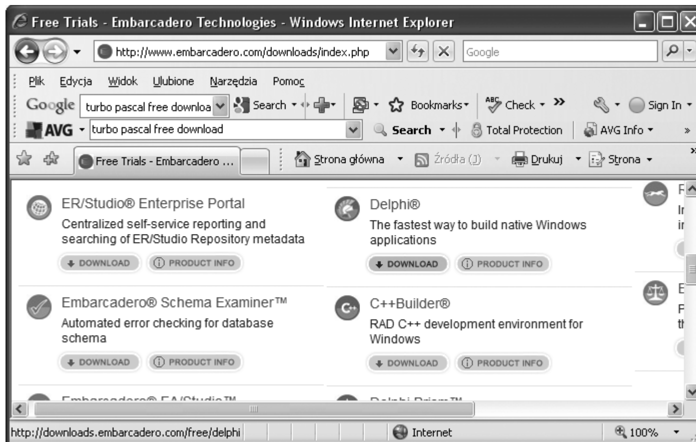
Rysunek D3. Po rozpakowaniu folderu należy uruchomić program INSTALL.EXE

programowanie_strukturalne_DODATEK.indb 10 2010-07-05 11:54:41

Rysunek D4. Po ustawieniu — najczęściej domyślnych — parametrów instalacji należy wybrać polecenie Start Installation

Rysunek D5. Turbo Pascal 5.5 po uruchomieniu (plik D:\TP\TURBO.EXE)

W podobny sposób znajdujemy Delphi i C++ Builder. W tym przypadku jednak trzeba się zalogować na stronie producenta *http://www.embarcadero.com/downloads/index.php* (rysunek D6). Wersje testowe są zazwyczaj ograniczone czasowo. Można je wykorzystywać zazwyczaj przez 30 lub 120 dni.



Rysunek D6. Po zalogowaniu możemy pobrać także Delphi i C++ Builder

Szczegółowy opis (w razie potrzeby) jest dostępny w języku polskim na stronie: *http://www.borland.pl/delphi/*.

Wersja 7.0 Turbo Pascala jest dostępna w sieci za darmo bez konieczności rejestracji (np. *http://www.brothersoft.com/downloads/turbo-pascal-7.html*).

2

Kopiowanie przykładów w C++

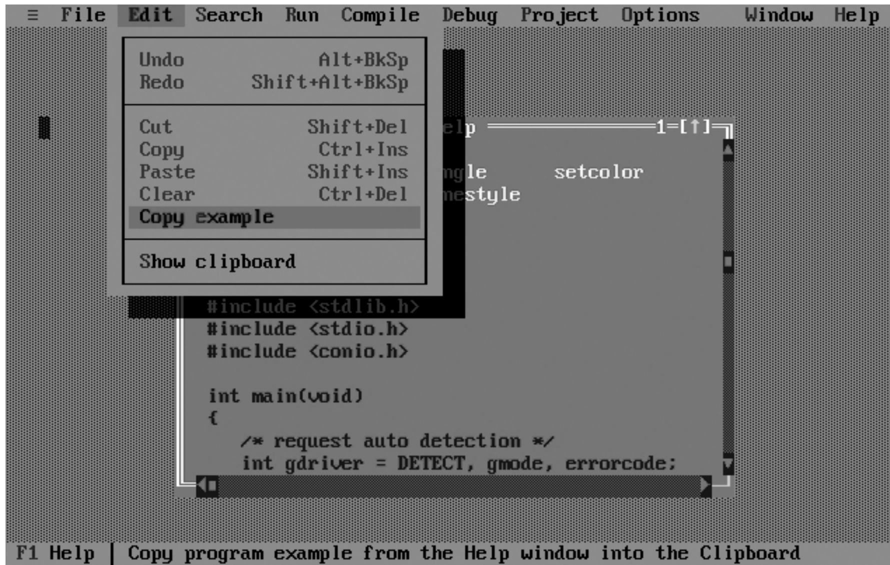
Aby uruchomić program przykładowy z systemu pomocy elektronicznej w Turbo C++, należy wykonać następujące czynności:

1. Uruchamiamy kompilator: C:\TC\BIN\TC.EXE.
2. Otwieramy system pomocy poleceniem z menu *Help* (Alt+H).
3. W menu *Help* wybieramy *Index* (spis).
4. Wpisujemy nazwę poszukiwanej funkcji, np. bar... i naciskamy [Enter].

System pomocy otwiera odpowiednie okno. Przechodzimy do jego dolnej części i odnajdujemy kod przykładu (patrz rysunek poniżej).

Rys. D7. Kod przykładu w systemie pomocy Turbo C++ (wszystkie funkcje są w Turbo C++ zilustrowane przykładami)

5. Wybieramy polecenie *Copy example* (kopiuj przykład — patrz: rysunek poniżej).

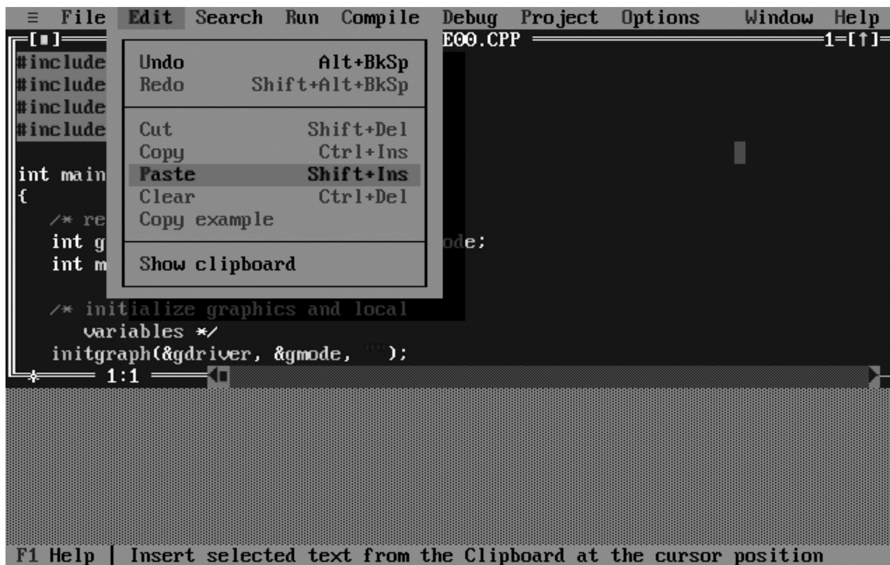


Rys. D8. Przykłady w Turbo C++ można skopiować automatycznie

6. Klawiszem [Esc] zamykamy okno systemu pomocy.

7. Otwieramy poleceniem *File/New* nowe okno edycji programu w Turbo C++.

8. Poleceniem *Edit/Paste* wstawiamy skopiowany przykład.



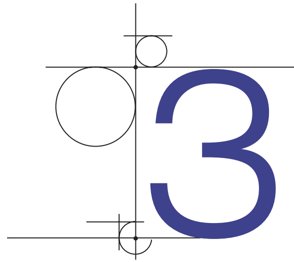
Rys. D9. Przykład został skopiowany

programowanie_strukturalne_DODATEK.indb 14 2010-07-05 11:54:42

9. Przeglądamy kod przykładowy i — jeśli trzeba — wprowadzamy modyfikacje.
10. Uruchamiamy program (Ctrl+F9) i obserwujemy jego działanie.
11. Zapisujemy program na dysku lub kończymy pracę z kompilatorem C++.

Materiał zawarty w niniejszym „Dodatku" stanowi uzupełnienie podręcznika. Do ilustracji przykładów można posłużyć się programami dokonującymi konwersji liczb dziesiętnych na dwójkowe (ich listingi zamieszczone są w podręczniku). Poprawność konwersji można sprawdzać za pomocą kalkulatora (*Start/Wszystkie programy/Akcesoria/Kalkulator*). Tabela i przykłady mogą zostać wykorzystane do ćwiczeń i zadań kontrolnych.

programowanie_strukturalne_DODATEK.indb 15 2010-07-05 11:54:42



System dwójkowy, ósemkowy, szesnastkowy

W tabeli D.1 poniżej przedstawiono zapis dwójkowy, ósemkowy, szesnastkowy i dziesiętny początkowych liczb naturalnych.

Tabela D.1. Liczby przedstawione w różnych formatach

Liczba szesnastkowa HEX(adecimal)	Liczba dwójkowa BIN(ary)	Liczba dziesiętna DEC(imal)	Liczba ósemkowa OCT(al)
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
A	1010	10	12
B	1011	11	13
C	1100	12	14
D	1101	13	15

E	1110	14	16
F	1111	15	17

System dwójkowy jest systemem pozycyjnym opartym na kolejnych potęgach dwójki. Każda cyfra dwójkowa jest nazywana bitem i może mieć wartość 0 albo 1.

Jeśli cyfra dwójkowa zajmuje w liczbie pozycję numer n (licząc od prawej), to ma wartość równą $0 * 2^n$ albo $1 * 2^n$. Najbardziej prawy bit w liczbie dwójkowej ma pozycję numer 0, a zatem ma wartość $0 * 2^0 = 0$ albo $1 * 2^0 = 1$ (zależy, czy jest jedynką, czy zerem). Zatem liczba dwójkowa, na przykład 1000_2, może być przeliczona na liczbę dziesiętną tak:

$1000_2 \rightarrow 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0 = 8_{10}$ (8 dziesiętnie)

UWAGA

Każde 4 cyfry dwójkowe odpowiadają 1 cyfrze szesnastkowej, na przykład:

1000 1010 1111 0111 = 8AF7

1010 1011 1111 1110 1000 1011 1110 1111 = ABFE 8BEF

Przykład 1.1. Jak zapisać w formacie dwójkowym liczbę –12 345?

1. Znajdujemy kod dwójkowy liczby 12 345, nie uwzględniając znaku:

0011 0000 0011 1001 = $1 * 2^{13} + 1 * 2^{12} + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^0 =$
8192 + 4096 + 32 + 16 + 8 + 1 = 12 345

2. Znajdujemy kod odwrotny:

1100 1111 1100 0110

3. Dodajemy 1, by uzyskać kod dopełniający (komplementarny):

1100 1111 1100 0111

To jest format dwójkowy liczby –12345. Pierwszy bit, czyli bit znaku, wskazuje, że jest to liczba ujemna.

Przykład 1.2. Zamiana „trójek” dwójkowych i „czwórek” na zapis ósemkowy i szesnastkowy.

Liczba dziesiętna 73 to 64 + 8 + 1, zatem jej zapis dwójkowy wygląda następująco:

1* 64 + 0*32 + 0*16 + 1*8 + 0*4 + 0*2 + 1*1

1 0 0 1 0 0 1

Dzielimy ten zapis na trójki:

001 001 001 → zapis ósemkowy: 111

Sprawdzamy: 1*1 + 1*8 + 1*64 = 73

Dzielimy ten zapis na czwórki:

0100 1001 → zapis szesnastkowy: 49

Sprawdzamy: 4 * 16 + 9 * 1 = 73

programowanie_strukturalne_DODATEK.indb 17 2010-07-05 11:54:42

Innymi słowy, dziesiętny odpowiednik dwójkowej liczby 1000 to liczba 8.

Jeśli chcemy zamienić liczbę dziesiętną, na przykład 10_{10}, na liczbę dwójkową, możemy to zrobić tak:

$10_{10} \rightarrow 8 + 2 \rightarrow 2^3 + 2^1 \rightarrow 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 1010_2$ (1010 dwójkowo)

Zapis dwójkowy:

$1101 = 1 * 2^3 + 1 * 2^2 + 1 * 2^0 = 8 + 4 + 1 = 13_{10}$

Zapis ósemkowy:

$370 = 3 * 8^2 + 7 * 8^1 + 0 * 8^0 = 3 * 64 + 7 * 8 + 0 = 248_{10}$

Zapis szesnastkowy (używane cyfry: 0 1 2 3 4 5 6 7 8 9 A B C D E F):

$F5 = 15 * 16^1 + 5 * 16^0 = 240 + 5 = 245_{10}$

3.1. Operacje bitowe

3.1.1. Przesunięcia

`00000001 << 3 = 00001000 = 8`	`SHL = Shift Left`
`00001101 >> 2 = 00000011 = 3`	`SHR = Shift Right`

Zapis: `8>>2` spowoduje przesunięcie bitów liczby 8 o dwie pozycje w prawo, co da w wyniku wartość 2 (dziesiętnie).

$8>>2 \rightarrow 1000>>2 \rightarrow (1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0)>>2$

— daje w rezultacie:

$(0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0) \rightarrow 0010$ (dwójkowo) $\rightarrow 2$ (dziesiętnie)

UWAGA

Przesunięcie może być CYKLICZNE (obcinany bit zostaje przepisany na początek).

Ciekawostka: Przesunięcie cykliczne może być w C/C++ zrealizowane np. tak:

(a<<shift) | (a>>(8-shift))

gdzie shift jest liczbą całkowitą z zakresu [0..7].

Zadziała to jednakże poprawnie tylko dla bajtów bez znaku, ponieważ interpretacja najstarszego bitu zmienia wynik:

(16<<3) = 128 (jeśli to unsigned char)

(16<<3) = -128 (jeśli to char)

Natomiast (-128>>3) = -16, podczas gdy (128>>3) = 16

Ale to tylko ciekawostka.

programowanie_strukturalne_DODATEK.indb 18 2010-07-05 11:54:43

Podobnie zapis `5<<1` oznacza, że należy przesunąć bity 0101 o jedną pozycję w lewo, co da wynik 1010, czyli 10 (dziesiętnie).

3.1.2. Funkcje logiczne: AND &, OR |, XOR ^, negacja NOT ~, dopełnienie NEG

```
01010101 & 11001100 = 01000100

01010101 | 11001100 = 11011101

00001101 ^ 11001100 = 11000001

NEG 01010101 = 10101011

NOT 11001100 = 00110011
```

Symbolicznie:

```
x & y

x | y

x ^ y

~x
```

— gdzie `x` i `y` oznaczają operandy.

Operator `&` porównuje każdy bit operandu `x` z odpowiednim bitem operandu `y`. Jeśli oba odpowiednie bity są równe 1, operator ustawia 1 na tej samej pozycji bitu wyniku. Jeśli choćby jeden spośród porównywanych bitów (lub oba) jest równy 0 — odpowiedni bit wyniku jest ustawiany na 0. Na przykład wyrażenie iloczynu bit po bicie operandów: `01 & 11` da wynik 01.

Operator bitowy sumy logicznej (alternatywy zwykłej) ustawia bit wyniku na 1, jeśli choćby jeden spośród bitów operandów (lub dwa) jest równy 1. Na przykład wyrażenie `01 | 11` zwróci wynik 11.

Operator `^` alternatywy wyłącznej (eXclusive OR = ALBO-ALBO) ustawia bit wyniku na 1, jeśli dokładnie jeden spośród odpowiednich bitów dwóch operandów jest równy 1 (ale nie dwa jednocześnie). Dlatego wyrażenie `01 | 11` zwróci wynik 10.

Wreszcie operator negacji (inwersji bitowej) `~` dokonuje operacji na pojedynczym operandzie, odwracając w nim wartość każdego bitu. Na przykład wyrażenie `~01` zwróci wynik 10. W tabeli D.2 poniżej podano przykładowe wyniki wyrażeń z zastosowaniem operatorów bitowych.

Tabela D.2. Przykłady zastosowania operatorów bitowych

WYRAŻENIA			**WYNIKI**		
Dziesiętnie [10]	**Hex [16]**	**Dwójkowo [2]**	**Dziesiętnie [10]**	**Hex [16]**	**Dwójkowo [2]**
12 & 10	0x0C & 0x0A	1100 & 1010	8	0x08	1000

programowanie_strukturalne_DODATEK.indb 19 2010-07-05 11:54:43

12 | 10	0x0C | 0x0A	1100 | 1010	14	0x0E	1110
12 ^ 10	0x0C ^ 0x0A	1100 ^ 1010	6	0x06	0110
~12	~0x0C	~0000000000001100	65523	FFF3	1111111111110011

UWAGA

- 1 bajt = 8 bitów — pozwala zapamiętać 2^8 (256) wartości, na przykład liczby 0 – 255, lub –128 – +127 (jeśli pierwszy bit jest bitem znaku), oznaczenie — 1B
- 1 kB (kilobajt) = 2^{10} B = 1024 B
- 1 MB (megabajt) = 2^{20} B = 1 048 576 B
- 1 GB (gigabajt) = 2^{30} B = 1 073 741 824 B
- 1 TB (terabajt) = 2^{40} B = 1024 GB

Słowo to ilość informacji przetwarzanej przez procesor w jednym cyklu (procesor 32-bitowy = procesor o słowie 32-bitowym). Popularne są mikroprocesory 8-bitowe, 16-bitowe, 32-bitowe i 64-bitowe. W komputerach klasy PC stosuje się 32- lub 64-bitowe. Procesory / rejestry 128-bitowe stosowane są w dużych komputerach (np. Cray). Komputer może być jednoprocesorowy (jak PC) lub wieloprocesorowy (serwery). Procesor może być jednordzeniowy (ang. *single-core*) lub wielordzeniowy (ang. *multicore*, *doublecore*, *quadcore*).

3.2. Prawa logiki i prawa de Morgana

Jeśli `X` i `Y` są zmiennymi logicznymi równymi `{0/1}` lub `{FALSE/TRUE}`, a znaki operatorów oznaczają odpowiednio:

```
| - Suma OR
& - Iloczyn AND
~ - Negacja NOT
```

— to:

```
1. X | ~X = 1
2. X & ~X = 0
3. X | 1 = 1
4. X & 1 = X
5. X | 0 = X
6. X & 0 = 0
7. X | X = X
8. X & X = X
```


programowanie_strukturalne_DODATEK.indb 20 2010-07-05 11:54:43

Powyższe podstawowe zależności są dość intuicyjne. Oprócz nich istnieją jednakże zasady mające kluczowe znaczenie w praktyce programowania, a już nieco mniej intuicyjne, na przykład:

9. X&Y + Y = Y ponieważ **(X | 1)&Y = 1 & Y = Y**

Jest to zasada pochłaniania jednej zmiennej, która, jak się okazuje, nie ma tu żadnego wpływu na wynik.

10. X&Y | ~X&Y = Y ponieważ **(X | ~X) &Y = 1 & Y = Y**

11. Prawa de Morgana:

11.1. ~(X&Y) ⇔ ~X | ~Y

Prawo zaprzeczenia koniunkcji: negacja koniunkcji jest równoważna alternatywie negacji.

Wyjaśnienie: jeśli nie mam psa (`~X`) lub nie mam kota (`~Y`), to znaczy, że nie jest prawdą, iż mam i psa, i kota jednocześnie (`X&Y`).

11.2. ~(X | Y) ⇔ ~X&~Y

Prawo zaprzeczenia alternatywy: negacja alternatywy jest równoważna koniunkcji negacji.

Wyjaśnienie: jeśli nie mam psa (`~X`) i jednocześnie (`&`) nie mam kota (`~Y`), to znaczy, że nie jest prawdą, iż mam psa lub kota (`X | Y`).

programowanie_strukturalne_DODATEK.indb 21 2010-07-05 11:54:43

4

Języki programowania — pojęcia podstawowe

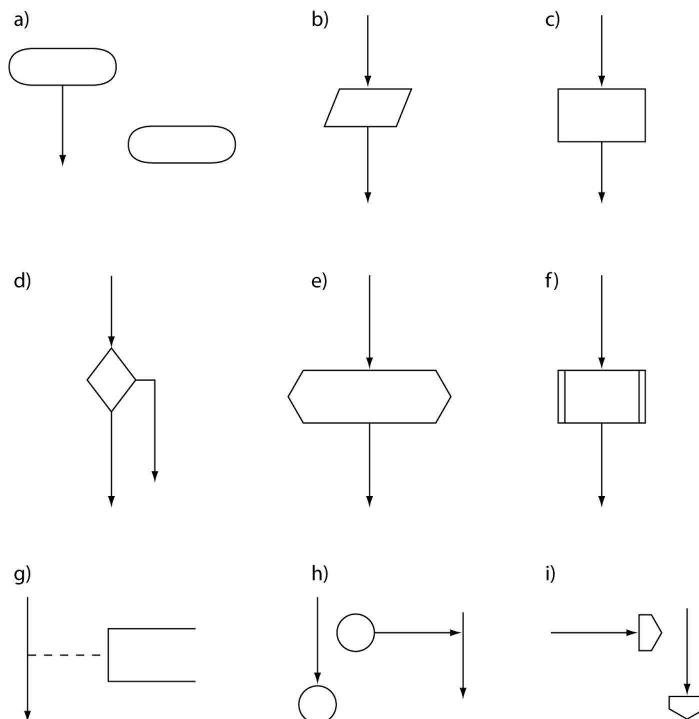
- Typy języków:
  - ✓ liniowe (np. QBASIC),
  - ✓ strukturalne (np. Visual Basic, Pascal, C),
  - ✓ obiektowe (np. Object Pascal, Delphi, C++, SmallTalk, Java, C#).
- Kategorie języków:
  - ✓ niskiego poziomu (asembler, C),
  - ✓ wysokiego poziomu (C/C++),
  - ✓ makropoleceń/programowania wsadowego (ang. *Batch Programming Language* — BPL),
  - ✓ specjalizowane, na przykład SQL (bazy danych).
- Koncepcje programowania i odpowiednie narzędzia programistyczne — przykłady:
  - ✓ programowanie strukturalne: Turbo Pascal, Turbo C,
  - ✓ programowanie obiektowe: Object Pascal, C++,
  - ✓ programowanie zdarzeniowe i wizualne: Delphi, C++ Builder, Kylix,
  - ✓ programowanie sieciowe i internetowe: Java, Visual Studio .NET, C++ Builder, JBuilder.

5

Schematy blokowe algorytmów

5.1. Definicje — rodzaje bloków

Na rysunku D10 poniżej przedstawiono symbole graficzne stosowane w schematach blokowych algorytmów i programów.



Rysunek D10. Symbole graficzne stosowane w grafach algorytmów (schematach blokowych)

a. **Blok graniczny** — początek, koniec, przerwanie lub wstrzymanie wykonywania działania, na przykład *Start*.

b. **Blok wejścia-wyjścia** — wprowadzanie danych do programu i wyprowadzanie wyników.

c. **Blok obliczeniowy** — wykonanie operacji, w rezultacie której zmienią się wartości lub postać danych.

d. **Blok decyzyjny** — wybór jednego z dwóch wariantów wykonywania programu na podstawie sprawdzenia warunku.

e. **Blok wywołania podprogramu** — wywołanie funkcji, metody lub większej części wydzielonego kodu, który jest logicznie samodzielną całością wykonującą określoną funkcję, np. wyznaczanie wartości minimalnej — MIN(x, y, z).

f. **Blok fragmentu** — większa część programu zdefiniowana odrębnie, funkcja biblioteczna producenta firmy trzeciej itp.

g. **Blok komentarza** — pozwala wprowadzać komentarze.

h. **Łącznik wewnętrzny** — służy do łączenia części schematu znajdujących się na tej samej stronie.

i. **Łącznik zewnętrzny** — służy do łączenia części schematu znajdujących się na odrębnych stronach.

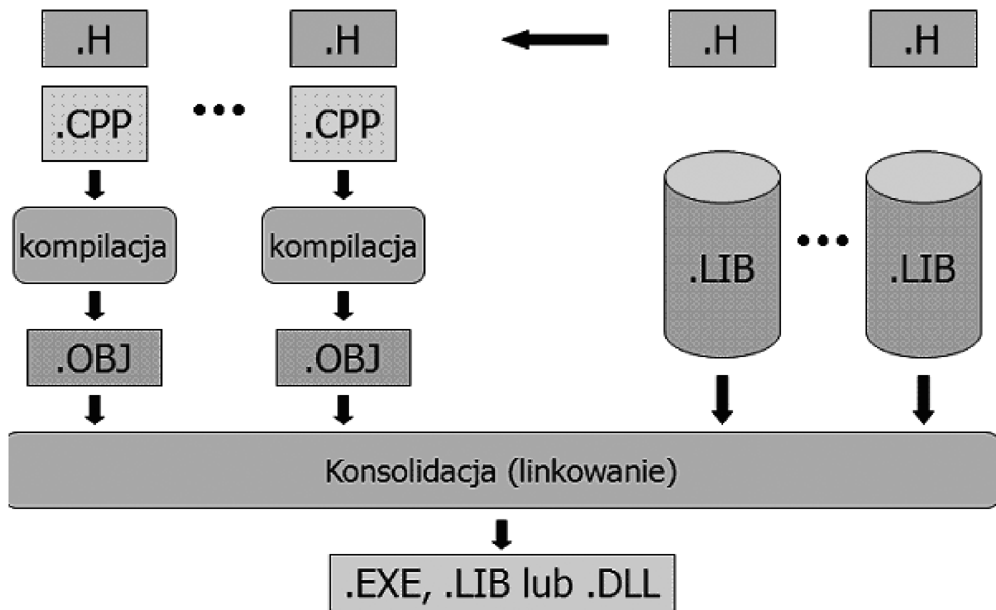
5.2. Opracowanie programów — kolejność czynności

1. Sformułowanie problemu (definicja zadania).
2. Analiza problemu.
3. Wybór metody (metod) rozwiązania.
4. Opracowanie algorytmu.
5. Kodowanie (implementacja programu).
6. Testowanie programu.
7. Sporządzenie dokumentacji.

5.3. Kompilacja programów

Uproszczony schemat graficzny kompilacji i konsolidacji programu (bez uwzględnienia kompilacji i dołączania zasobów ani roli kompilatora zasobów /ang. *Resource Compiler*/ oraz bez wyodrębnienia działania preprocesora) pokazano na rysunku D11 poniżej.

programowanie_strukturalne_DODATEK.indb 24 2010-07-05 11:54:43



Rysunek D11. Schemat kompilacji i konsolidacji programów

Powyższy rysunek przedstawia uproszczony schemat kompilacji i konsolidacji. Schemat nie uwzględnia działania preprocesora ani możliwego pośredniego etapu asemblacji.

UWAGA

W Pascalu zamiast plików nagłówkowych **.H* dołączane są moduły (**.TPU*).

6

Elementy Pascala

6.1. Konstrukcja programu w Pascalu

Poniższy szablon stanowi uogólniony model. Jak nietrudno zorientować się po przejrzeniu przykładowych listingów zamieszczonych w książce, nie wszystkie elementy tego szablonu muszą występować w każdym przykładowym programie w języku Pascal. Szablon nie uwzględnia również zaawansowanych stylów programowania strukturalnego, obiektowego czy zdarzeniowego. To jedynie prosty szablon elementarny dla programu sekwencyjnego, który nie został poddany żadnej formie modularyzacji.

```
Program etykieta; { nazwa programu (opcjonalna) }
Uses ... { lista używanych modułów }
Const ... { deklaracje stałych }
Type ... { deklaracje typów }
Var ... { deklaracje zmiennych }

BEGIN
... { główny blok programu }
END.
```

DEFINICJA

Identyfikator — każdy ciąg znaków złożony z symboli będących literą, cyfrą lub znakiem podkreślenia, rozpoczynający się literą lub podkreśleniem.

6.2. Typy danych

Podstawowe typy danych mają decydujący wpływ na tworzenie deklaracji zmiennych prostych i agregatów danych. Znajomość typów danych jest niezbędna również do poprawnego tworzenia deklaracji i definicji funkcji oraz procedur. O zakresie stosowalności decydują zazwyczaj dwa podstawowe parametry — zakres dostępnych war-

tości i wielkość zajmowanego pola pamięci (w bajtach). Warto zwrócić uwagę, że od wyboru typów danych zależy w sposób bezpośredni wielkość pliku wykonywalnego (zmienia się ilość zarezerwowanej pamięci), a w sposób pośredni — prędkość działania kodu. Do zademonstrowania różnic w czasach wykonania operacji w zależności od typu danych można wykorzystać przykładowe programy ilustrujące pomiar czasu rzeczywistego. W Pascalu do pomiaru czasu rzeczywistego z dokładnością do 1/100 sekundy można wykorzystać procedurę biblioteczną `GetTime()`.

6.2.1. Typy proste i porządkowe

Typ wyliczeniowy

```
Type identyfikator = (lista_identyfikatorów);

{ Przykład: }

Type DniTygodnia = (poniedzialek, wtorek, sroda, czwartek, piatek, sobota,
niedziela);
```

Analogią w C/C++ jest typ wyliczeniowy (patrz: enum).

Typy numeryczne całkowite

Nazwa	Zakres
Shortint	[–128, 127]
Byte	[0, 255]
Integer	[–32 768, 32 767]
Word	[0, 65 535]
Longint	[–2 147 483 648, 2 147 483 647]

Typ logiczny

```
Boolean = (False, True)
```

Typ znakowy char — 1 bajt

```
Var Znak : Char;
```

Typ okrojony

```
Type dwucyfrowe = 10 .. 99;

Type litera = 'A' .. 'Z';

Type dni_robocze = poniedzialek .. piatek;
```


programowanie_strukturalne_DODATEK.indb 27 2010-07-05 11:54:44

Typy numeryczne rzeczywiste

Nazwa	Wielkość (patrz: SizeOf(...))
Real	6 bajtów
Single	4 bajty
Double	8 bajtów
Extended	10 bajtów

Dane tekstowe: typ łańcuchowy

```
Type imie = String[25];
```

6.2.2. Typy strukturalne

Tablice: typ tablicowy

```
Type wektor = Array[0..20] Of Integer;
Type macierz1 = Array[1..10] Of Array[1..20] Of Real;
Type macierz2 = Array[1..20,1..20] Of Real;
```

Struktury danych: typ rekordowy

```
Type
 Data = Record
  dzien : 1 .. 31;
  miesiac : 1 .. 12;
  rok : Integer;
 End;

Type
 DanePersonalne = Record
  nazwisko : string[25];
  imie : string[18];
  data_urodzenia : Data;
 End;
```

Zbiory: Set Of

```
Type dni_pracy = Set Of dni_robocze;
```

6.2.3. Definiowanie typów danych użytkownika/programisty

```
Type liczba = Integer;
```

6.3. Stałe i zmienne — deklarowanie i inicjowanie

W językach strukturalnych deklaracja zmiennej, funkcji lub procedury jest konieczna przed jej pierwszym użyciem w programie. O ile BASIC (język interpretowany) nie wymaga deklaracji zmiennej, o tyle Pascal czy C wymagają, by deklaracje zmiennych nastąpiły przed początkiem części operacyjnej, czyli instrukcji. C++ jest pod tym względem bardziej elastyczny, ponieważ pozwala deklarować zmienną bezpośrednio przed jej użyciem. Miejsce deklaracji zmiennej decyduje o zakresie jej dostępności (widoczności). Zmienne deklarowane na początku, przed częścią operacyjną programu, to zmienne globalne. Zmienne deklarowane w obrębie funkcji/procedur to zmienne lokalne. Zmienne globalne są zazwyczaj automatycznie zerowane, ale wartość zmiennych lokalnych przed ich zainicjowaniem w programie może pozostawać przypadkowa. Deklaracja zmiennej powoduje zarezerwowanie dla niej pamięci. Zainicjowanie zmiennej powoduje przypisanie jej początkowej wartości.

6.3.1. Deklaracje zmiennych

```
Var
 x : Integer;
 y : Real;
 b1, b2 : Boolean;
```

6.3.2. Deklarowanie i inicjowanie stałych i zmiennych

```
Const Pi : Real = 3.14;
znak : Char = 'A';
Const C : Real = 123;
Var X : Real;
....
X := 12345;
...

Var
 a : wektor;
```


programowanie_strukturalne_DODATEK.indb 29 2010-07-05 11:54:44

```
 b : macierz;
 c : Array[1..10] Of wektor;
 d : Array[1..10] Of macierz;

a[2] := 13.456;
b[3][4] := 12.34;
b[2,3] := 11.12345;


{Konkatenacja, czyli laczenie ciagow znakow}
{Rezultat:                                  }
'To ' + 'Oddzielne'+' '+'teksty'  → 'To Oddzielne teksty'
```

6.4. Deklarowanie i inicjowanie stałych i zmiennych strukturalnych (operator kropki)

Zmienne strukturalne (podobnie jak zwykłe zmienne) mogą być zmiennymi globalnymi lub lokalnymi. Jeśli rekord (tablica rekordów) jest zmienną globalną, zazwyczaj zostanie automatycznie wyzerowany. Jeśli rekord jest zmienną lokalną, to do momentu zainicjowania zmienna taka może zawierać zupełnie przypadkową zawartość. Próba użycia niezainicjowanej zmiennej może powodować trudny do przewidzenia i zdiagnozowania błąd w działaniu programu. W zmiennych typu `String` w Pascalu pierwszy bajt określa liczbę znaków, dlatego automatyczne wyzerowanie powoduje uznanie tekstu za „łańcuch o zerowej długości".

```
Type
 Data = Record
  dzien : 1..31;
  miesiac : 1..12;
  rok : Integer;
 End;


Var
 data_ur : Data;
 facet : Record
  nazwisko : string[25];
  imie : string[20];
  data_ur : data
 End;
```

```
Begin
 data_ur.dzien := 15;
 data_ur.miesiac := 5;
 data_ur.rok := 1988;
 facet.nazwisko := 'Burak';
 facet.imie := 'Jan-Maryja';
 facet.data_ur.rok := 1988;
 facet.data_ur := data_ur
 {...}
```

6.5. Operatory Pascala

`+`,`-` — zmiana znaku (operatory jednoargumentowe)

`@` — operator adresowy, pozwala zainicjować wskaźnik

`Not` — negacja logiczna

`*`, `/`, `Div`, `Mod`, — mnożenie, różne formy dzielenia

`And` — iloczyn logiczny

`Shl`, `Shr` — przesunięcie bitów w lewo/w prawo

`+`, `-` — dodawanie/odejmowanie arytmetyczne

`Or`, `Xor` — alternatywa zwykła i wyłączna

`=`, `<>`, `<`, `>`, `<=`, `>=` — operatory relacji

`In` — operator przynależności do zbioru

6.6. Wyrażenia

Wyrażenia w rozumieniu Pascala to:

- stała,
- zmienna,
- wywołanie funkcji,
- wyrażenia połączone operatorem dwuargumentowym.

UWAGA

Wobec wyrażeń można stosować operatory relacji (np. porównywać wyrażenia). Relacje dają w wyniku wartości typu `Boolean`, a działają także dla łańcuchów tekstowych (wg kodów ASCII znaków).

programowanie_strukturalne_DODATEK.indb 31 2010-07-05 11:54:44

6.7. Operacje na zbiorach (teoria mnogości)

Operacje na zbiorach (*Set*) pozwalają na łatwe wyznaczenie części wspólnej i różnicy zbiorów. W najprostszym przypadku (jak w przykładzie poniżej) elementami zbioru są liczby całkowite.

+, -, * — suma, różnica i iloczyn zbiorów o zgodnych typach elementów. Przykład:

	rezultat:
[2,3,4]+[4,5]	[2,3,4,5]
[2,3,4]-[4,5,6]	[2,3]
[3,4]*[4,5,6]	[4]

6.8. Instrukcje Pascala

Instrukcje elementarne zakończone są średnikiem. Instrukcja złożona (blok instrukcji) ujmowana jest w tzw. nawiasy programowe, czyli parę słów kluczowych `Begin-End`. Jest jeden wyjątek: średnika nie stawia się przed słowem kluczowym `Else` w instrukcji warunkowej.

```
{blok instrukcji sklada sie z instrukcji elementarnych}

Begin
 Instrukcja_1;
 ...
 Instrukcja_n;
End;
```

6.8.1. Instrukcja warunkowa If-Then-Else (Else jest opcjonalne)

```
If wyrazenie Then instrukcja;

{ lub }

If wyrazenie Then instrukcja1
Else instrukcja2;

{ Maksimum sposrod 2 liczb x i y }
If x>y Then maksimum := x Else maksimum := y;
```


programowanie_strukturalne_DODATEK.indb 32 2010-07-05 11:54:44

```
{ Zagniezdzone if-then-else-if ... }

If wyrażenie_1 Then instrukcja_1
Else If wyrażenie_2 Then instrukcja_2
Else If wyrażenie_3 Then instrukcja_3
Else { ... };
```

6.8.2. Instrukcja wyboru wielokrotnego Case

```
Case wyrazenie Of
 sekwencja_instrukcji_wyboru
End;

{ lub }

Case wyrazenie Of
 sekwencja_instrukcji_wyboru
Else instrukcja;
End;

Case miesiac Of
 1,3,5,7,8,10,12 : dni := 31;
 2 : dni := 28;
 4,6,9,11 : dni := 30;
End;

Case miesiac Of
 4,6,9,11 : dni := 30;
 2 : dni := 28;
 else dni := 31;
End;
```

6.8.3. Instrukcja For

```
For zmienna := wyr_1 To wyr_2 Do instrukcja;
{ lub }
For zmienna := wyr_1 DownTo wyr_2 Do instrukcja;
```

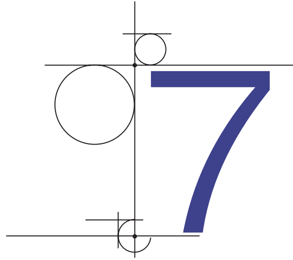
6.8.4. Instrukcje While i Repeat

```
While wyrażenie Do instrukcja;

Repeat instrukcja Until wyrażenie;
```

UWAGA

1. Aby skonstruować system funkcjonalnie pełny, niezbędne są dwie instrukcje pętli. Jedna musi sprawdzać warunek iteracji na wejściu (w Pascalu jest to `While`), a druga na wyjściu (w Pascalu — `Until`). Inne instrukcje pętli (np. `For-To`, `For-Down-To`) nie są niezbędne, ale są wygodne w stosowaniu.
2. `Break/Continue` w Pascalu są procedurami bibliotecznymi, a w `C/C++` są słowami kluczowymi języka, ale ich rola i działanie pozostają takie same.

programowanie_strukturalne_DODATEK.indb 34 2010-07-05 11:54:44



Funkcje i procedury

7.1. Obsługa standardowego wejścia-wyjścia

```
Write(lista_argumentow);
WriteLn(lista_argumentow);
Read(lista_argumentow);
ReadLn(lista_argumentow);
```

Wersje z końcówką `Ln()` dodatkowo zapisują/czytają znaki końca wiersza (`CRLF`).

Formatowanie: każdy argument powinien mieć postać:

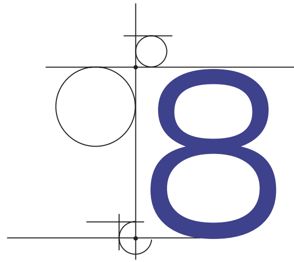
- *wyrażenie*
- *wyrażenie : długość*
- *wyrażenie : długość : liczba_pozycji_dziesiętnych*

7.2. Definicja procedury

```
Procedure nazwa_p ( lista_parametrow );
deklaracje_zmiennych, stalych, itp.
Begin
 instrukcje
End;
```

7.3. Definicja funkcji

```
Function nazwa_f ( lista_parametrow ) : typ_rezultatu;
deklaracje_zmiennych, stalych, itp.
Begin
 instrukcje
End;
```



8 Obsługa wejścia-wyjścia. Operacje plikowe

8.1. Opis zmiennej plikowej

```
Type Osoba = Record
 nazwisko : string[25];
 imie : string[20];
End;

TypPlikowy = File Of Osoba;  { plik o zdefiniowanym typie elementow}
TypPlikowy2 = File;     { plik binarny, niezdefiniowany typ }
TypPlikowy3 = Text;     { plik tekstowy }
```

8.2. Skojarzenie zmiennej z fizycznym plikiem dyskowym, otwarcie pliku

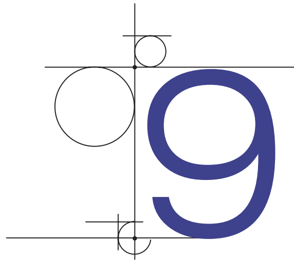
```
Var
 plik : TypPlikowy;
 ...
 Assign(plik, 'C:\OSOBY.PLK');

{ Otwieranie i zamykanie pliku }
{ Reset - aby otworzyc istniejacy plik }
Reset(zmienna_plikowa);
```

```
Reset(zmienna_plikowa, rozmiar_zapisu);

{ Rewrite - aby utworzyc nowy plik }
Rewrite(zmienna_plikowa);
Rewrite(zmienna_plikowa, rozmiar_zapisu);

{ Append - aby dopisywac do pliku tekstowego }
Append(zmienna_plikowa);

{ Close - aby zamknac plik }
Close(zmienna_plikowa);
```

8.3. Operacje na danych pliku dyskowego (odczyt, zapis)

```
{ Dla plikow tekstowych }
Write(zmienna_plikowa, lista_argumentow);
WriteLn(zmienna_plikowa, lista_argumentow);
Read(zmienna_plikowa, lista_zmiennych);
ReadLn(zmienna_plikowa, lista_zmiennych);

{ Dla plikow zdefiniowanych, zdefiniowany typ elementu }
Write(zmienna_plikowa, lista_zmiennych_wyjsciowych);
Read(zmienna_plikowa, lista_zmiennych_wejsciowych);

{ Dla plikow niezdefiniowanych, binarnych }
BlockWrite(zmienna_plikowa, bufor, licznik);
BlockWrite(zmienna_plikowa, bufor, licznik, wynik);
BlockRead(zmienna_plikowa, bufor, licznik);
BlockRead(zmienna_plikowa, bufor, licznik, wynik);

{ Czy to koniec pliku? End Of File}
Eof(zmienna_plikowa);

{ Czy to koniec wiersza? – End Of Line}
Eoln(zmienna_plikowa);
```

```
{ Aktualna pozycja w pliku }
FilePos(zmienna_plikowa);

{ Rozmiar pliku }
FileSize(zmienna_plikowa);

{ Przejdz do wskazanej pozycji w pliku }
Seek(zmienna_plikowa, pozycja);

{ Pobierz biezacy katalog dysku okreslonego przez
Zmienna liczba (0 - dysk biezacy, 1 - A:, 2 - B:, 3 - C: ...) }
GetDir(liczba, zmienna_lancuchowa);

{ Zmien biezacy katalog }
ChDir(wyrazenie_lancuchowe);

{ Utworz katalog }
MkDir(wyrazenie_lancuchowe);

{ Usun pusty katalog }
RmDir(wyrazenie_lancuchowe);

{ Usun plik (zamkniety) }
Erase(zmienna_plikowa);
```



C — przydatne definicje

9.1. Struktury i unie

```
struct data
{
  int dzien;
  int miesiac;
  int rok;
};
```

9.1.1. Zmienne typu strukturalnego i wskaźniki do struktur

```
struct data dzis = {30, 10, 2009}, *pd = &dzis;

printf("%d-%02d-%d", dzis.dzien, dzis.miesiac, dzis.rok);
printf("%d-%02d-%d", pd->dzien, pd->miesiac, pd->rok);

++pd->dzien; /* inkrementacja pola dzien poprzez wskaznik */
```

9.1.2. Inicjowanie tablicy (wektora) struktur, autoreferencja

```
struct data MojeDaty[] = {{1, 1, 2007}, {2, 11, 2009}, {13, 5, 2008}};

/* Struktury ze wskaznikami do samych siebie – autoreferencja struktur. */
struct drzewo
```

```
{
  float wartosc_wezla;
  struct drzewo *prawe_poddrzewo, *lewe_poddrzewo;
};
```

9.1.3. Unie

```
union Typ
{
  int i[2];     // 4 bajty, 2 liczby calkowite
  float f;      // 4 bajty, 1 liczba zmiennoprzecinkowa
  void *p;      // 4 bajty – wskaznik
} zmienna;
zmienna.f = 1.12345;
```

9.2. Obsługa plików dyskowych

Elementy potrzebne do obsługi plików:

- struktura `FILE` (predefiniowany typ struktur),
- wskaźnik do pliku (tu oznaczony `*fp` — *file pointer*).

```
FILE *fp = fopen(NazwaPliku, tryb_otwarcia);
...
fclose(fp);    // zamykanie pliku
```

Możliwe tryby otwarcia:

`r` — otwórz, do odczytu

`w` — otwórz, do zapisu (zamazuje starą treść)

`a` — otwórz, by pisać (stań na końcu — `append`, jeśli plik nie istnieje, to jest on tworzony)

`r+` — otwórz, żeby czytać i pisać

`w+` — otwórz, żeby czytać i pisać

`a+` — otwórz, żeby czytać i pisać (stań na końcu — `append`, jeśli plik nie istnieje, to jest on tworzony)

`b` — otwórz jako plik binarny

`t` — otwórz jako plik tekstowy

Flagi dostępu:

`O_RDONLY` — otwórz, żeby czytać

programowanie_strukturalne_DODATEK.indb 40 2010-07-05 11:54:45

`O_WRONLY` — otwórz, żeby pisać

`O_RDWR` — otwórz, żeby pisać i czytać

`O_APPEND` — po otwarciu stań na końcu pliku

`O_CREAT` — stwórz plik, jeśli nie istnieje

`O_TRUNC` — jeśli plik istnieje, wyczyść zawartość

`O_BINARY` — otwórz jako plik binarny

`O_TEXT` — otwórz jako plik tekstowy

Funkcje stosowane do obsługi plików:

```
int putc(int c, FILE *stream);
int getc(FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int fputs(const char *s, FILE *stream);
char *fgets(char *s, int n, FILE *stream);
```

Standardowe urządzenia: `stdin` (domyślne wejście — klawiatura), `stdout` (domyślne wyjście — ekran), `stderr` (domyślnie komunikaty o błędach kierowane są nie do pliku, lecz także na ekran).

`*stream` — wskaźnik do strumienia

9.3. Dynamiczne struktury danych — kolejka, lista

Poniższe kody zawierają uogólnione schematy umożliwiające budowę programów proceduralno-strukturalnych w Pascalu i w C/C++.

```
typedef struct skladnik_kolejki
{
  typ_obiektu obiekt;
  struct skladnik_kolejki *wskaznik;
} skladnik_kolejki;

void DoKolejki(skladnik_kolejki **poczatek,
  skladnik_kolejki **koniec, typ_obiektu o)
{
  skladnik_kolejki *tmp = (skladnik_kolejki *) malloc(sizeof(skladnik_ko-
lejki));
```


programowanie_strukturalne_DODATEK.indb 41 2010-07-05 11:54:45

```
  tmp->obiekt = o;
  tmp->wskaznik = 0;
  if (*koniec)
  {
    (*koniec)->wskaznik = tmp;
    *koniec = tmp;
  }
  else
    *koniec = *poczatek = tmp;
}

typ_obiektu ZKolejki(skladnik_kolejki **poczatek,
  skladnik_kolejki **koniec)
{
  skladnik_kolejki *tmp = *poczatek;
  typ_obiektu skladnik = tmp->obiekt;
  if (*koniec == *poczatek) *koniec = 0;
  *poczatek = tmp->wskaznik;
  free(tmp);
  return skladnik;
}

// lista dwukierunkowa
typedef struct element_listy
{
  typ_obiektu obiekt;
  struct element_listy *poprzedni;
  struct element_listy *nastepny;
} element_listy;

void DoListyZaElement(element_listy *zaKtory, typ_obiektu o)
{
  element_listy *nowy = (element_listy *) malloc(sizeof(element_listy));
  nowy->obiekt = o;
  nowy->poprzedni = zaKtory;
  nowy->nastepny = zaKtory->nastepny;
```

```
    nowy->nastepny->poprzedni = nowy;
    zaKtory->nastepny = nowy;
  }

  void ZListy(element_listy *ktory)
  {
    ktory->poprzedni->nastepny = ktory->nastepny;
    ktory->nastepny->poprzedni = ktory->poprzedni;
    free(ktory);
  }
```

9.4. Formatowanie — funkcje printf() i scanf()

Funkcje `printf()`, `scanf()` posługują się tymi samymi specyfikatorami typu. Specyfikator typu to inaczej polecenie konwersji danych do zadanej postaci docelowej. W tabeli D.3 podano listę specyfikatorów.

Tabela D.3. Pełna lista specyfikatorów, które można stosować w wywołaniach funkcji printf()

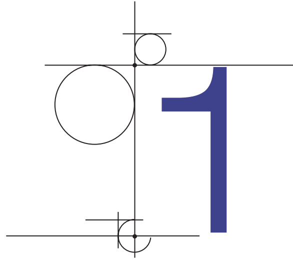
%c	Znak — typ `char`
%d	Liczba — typ `int`
%i	Liczba — typ `int` (to samo co `%d`)
%e	Format wykładniczy z użyciem małej litery `e`
%E	Format wykładniczy z użyciem wielkiej litery `E`
%f	Liczba zmiennoprzecinkowa — typ `float`
%g	Zastosuj `%f` lub `%e` — wybierz format, w którym wynik będzie krótszy
%G	Zastosuj `%f` lub `%E` — wybierz format, w którym wynik będzie krótszy
%o	Liczba ósemkowa bez znaku
%s	Łańcuch znaków (`string`)
%u	Liczba całkowita, typ `unsigned` — jak `int`, ale bez znaku
%x	Liczba szesnastkowa bez znaku (z zastosowaniem przyrostka `x`, małe litery)
%X	Liczba szesnastkowa bez znaku (z zastosowaniem przyrostka `X`, wielkie litery)
%p	Argument odpowiadający wskaźnikowi (`pointer`)
%n	Rejestruj liczbę znaków wyprowadzonych do tego momentu
%%	Wyprowadź znak „`%`” („we własnej osobie”)

programowanie_strukturalne_DODATEK.indb 44 2010-07-05 11:54:45

Część II

Pascal i C/C++ — rozdziały dodatkowe

programowanie_strukturalne_DODATEK.indb 46 2010-07-05 11:54:45



Jak w Windows XP uruchamiać aplikacje tekstowe?

Przejścia do środowiska Windows XP najprościej można dokonać poprzez zastosowanie kompilatora BPW (ang. *Borland Pascal for Windows*) lub Delphi. Programy wykonujące proste operacje „przetłumaczyć" można bardzo łatwo. Zamiast:

```
Uses Crt;     { dla DOS }
```

— należy zastosować klauzulę:

```
Uses WinCrt;     { dla Windows }
```

Oto przykład prostego programu przystosowanego dla środowiska Windows. Pewne elementy spotkamy w dziale poświęconym Delphi, ale w szkołach, w których jest dostępna wersja BPW, można wszystkie dotychczasowe przykłady uruchamiać w bardziej eleganckich okienkach.

Listing D01.PAS

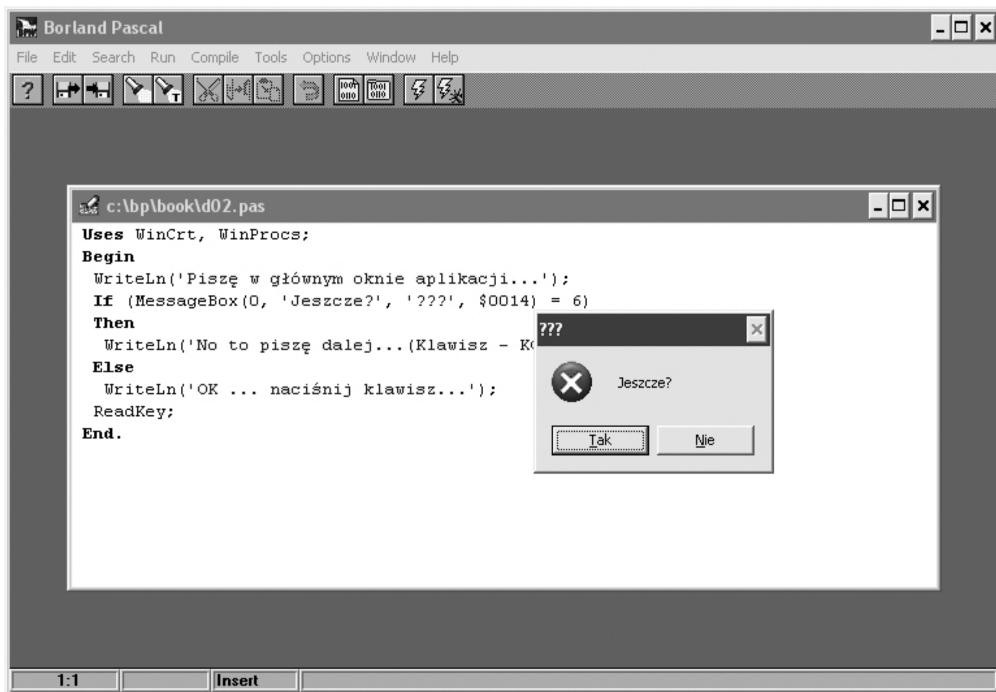
```
Uses WinCrt;
Begin
 WriteLn('Piszę w głównym oknie!');
 WriteLn('Potrafię liczyć: ', 2*2+1);
 ReadLn;
End.
```

Jeśli chcemy skorzystać z funkcji i procedur typowych dla środowiska graficznego Windows, należy zastosować dodatkowo stosowny moduł, na przykład *WINPROCS.TPW*. Warto zwrócić uwagę, że moduły dla DOS mają charakterystyczne rozszerzenie TPU (ang. *Turbo Pascal Units*), a te dla Windows — TPW (ang. *Turbo Pascal for Windows Units*). Przydatną i zarazem najbardziej charakterystyczną dla środowiska Windows

funkcją z tego modułu jest `MessageBox(...)` — typowe okienko komunikatów. Poniżej prosty przykład zastosowania okienka do dialogu z użytkownikiem. Efekt pokazano na rysunku D12.

Listing D02.PAS

```
Uses WinCrt, WinProcs;
Begin
 WriteLn('Piszę w głównym oknie aplikacji...');
 If (MessageBox(0, 'Jeszcze?', '???', $0014) = 6)
 Then
  WriteLn('No to piszę dalej...(Klawisz - KONIEC)')
 Else
  WriteLn('OK ... naciśnij klawisz...');
 ReadKey;
End.
```



Rysunek D12. Prosta aplikacja Pascala w środowisku Windows

Okienko dialogowe typu `MessageBox(...)` pozwala w prosty sposób umieszczać wewnątrz typowe przyciski sterujące, predefiniowane w Windows ikonki (`MB_ICONSTOP`, `MB_ICONQUESTION` itp.) i dodatkowo sprawdzać w programie, co wybrał użytkownik.

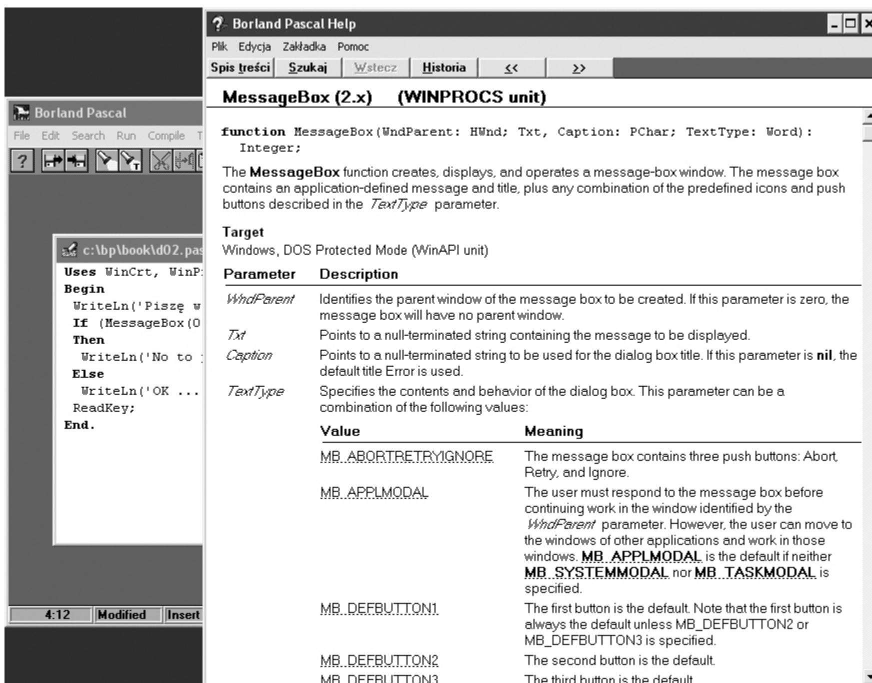
Identyfikatorami okienek, przycisków i innych stałych mogą być nazwy słowne, na przykład `MB_OK`, lub liczby. W przykładowym programie zastosowany został właśnie ten drugi sposób. Podana tam wartość `$0014` to szesnastkowy identyfikator określający, że w okienku powinny się znaleźć:

- przycisk [*Tak*] — `IDYES`
- przycisk [*Nie*] — `IDNO`
- ikonka „*X*" — `MB_ICONSTOP`

UWAGA

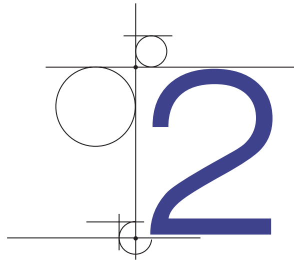
Identyfikatory przycisków i ikon można sumować.

Identyfikator *6* służy sprawdzeniu, czy użytkownik kliknął przycisk *Tak* w okienku. Można by tu oczywiście przytoczyć tabelę predefiniowanych stałych, ale lepiej po prostu wiedzieć, gdzie jej szukać. Na rysunku D13 zaprezentowano fragment systemu pomocy *Borland Pascal Help*.



Rysunek D13. Predefiniowanych parametrów jest sporo — najlepiej po prostu wiedzieć, gdzie ich szukać

Niestety, nie wszystkie funkcje i procedury znane z modułu `Crt` mają swoje dokładne odpowiedniki w `WinCrt`.



Pascal — trzy sposoby komputerowej animacji

Najprostsza animacja komputerowa polega na umiejętnym wykorzystaniu prostej zasady: czarny rysunek na czarnym tle jest słabo widoczny.

Posługując się tą zasadą, napiszemy pierwszy program dokonujący animacji. Poprzez rysowanie dwóch elementów graficznych — linii prostej i okręgu — najpierw na biało, a następnie w tym samym miejscu na czarno, uzyskamy prosty efekt poruszania się po ekranie.

2.1. Animacja — sposób 1

Jeśli tło rysunku jest czarne (domyślne), narysowanie linii w kolorze czarnym powoduje jej „zniknięcie" z ekranu. Jeśli teraz narysujemy ją powtórnie o kilka pikseli dalej, po czym znów „pokryjemy" linią czarną, uzyskamy efekt przesuwania się linii po ekranie. Sekwencja takich rozkazów mogłaby wyglądać na przykład tak:

```
SetBkColor(BLUE);              { kolor tla, np. niebieskie }
SetColor(YELLOW);              { kolor linii, np. zolta }
Line(100, 100, 200, 200);      { rysujemy linie zolta (linia widoczna) }
Delay(100);                    { chwile czekamy }
SetColor(BLUE);                { przestawiamy kolor rysunku na niebieski }
Line(100, 100, 200, 200);      { rysujemy linie na niebiesko w tym samym miejscu (linia
„znika") }
```

```
SetColor(YELLOW);              { wracamy do koloru zoltego }
Line(100, 101, 200, 201);      { rysujemy linie na zolto, ale juz przesunieta }
Delay(100);                    { chwile czekamy }
SetColor(BLUE);                { ustawiamy kolor rysunku na niebieski }
Line(100, 101, 200, 201);      { linia znika }
```

Zamiast pisać ręcznie setki instrukcji, sparametryzujemy program, posługując się zmiennymi, i każemy komputerowi samodzielnie wyliczać następne współrzędne poruszanych po ekranie elementów graficznych. Prosta pętla programowa wykonująca takie zadanie w Turbo Pascalu mogłaby wyglądać na przykład tak:

```
Var I: Integer;
 ...
BEGIN
 ...
For I:= 1 To 100 Do
Begin
 SetColor(Yellow);
 Line(100, I, 200, I);
 SetColor(Blue);
 Line(100, I, 200, I);
End;
 ...
```

Jeśli po wyjaśnieniu tych podstaw napiszemy prosty program doświadczalny wykonujący animację komputerową dwóch elementów graficznych, jego tekst źródłowy może wyglądać na przykład tak jak na listingu poniżej.

Listing D03.PAS

```
Program ANIMACJA1;
Uses Crt, Graph;
Var Karta, Tryb, I, Koniec: Integer;

BEGIN
 Karta:=Detect;
 InitGraph(Karta, Tryb, 'C:\TP\BGI');
 SetBkColor(Blue);
 Koniec:=GetMaxY;
 For I:=1 To Koniec Do
```


programowanie_strukturalne_DODATEK.indb 51 2010-07-05 11:54:46

```
 Begin
  SetColor(White);
  Line(50, I, 300, I);
  Circle(I, 200, 20);
  Delay(10);
  SetColor(Blue);
  Line(50, I, 300, I);
  Circle(I, 200, 20);
 End;
 CloseGraph;
END.
```

Linia prosta będzie poruszać się z góry w dół, a okrąg z lewa na prawo. Zwróćmy uwagę, że górną granicę pętli `For` podaliśmy tym razem nie bezpośrednio za pomocą liczby:

```
For I:= 1 To 480 Do
```

— lecz w postaci zmiennej (parametr `Koniec`), której wcześniej nadaliśmy odpowiednią wartość:

```
Koniec := GetMaxY;
 ...
For I := 1 To Koniec Do
```

Chodzi o to, by linia doszła aż do końca (`MaxY`) ekranu. Taki sposób parametryzowania programów jest bardzo wygodny. Gdybyśmy chcieli, by linia dochodziła na przykład tylko do połowy ekranu, wystarczy zmienić parametr w jednym z dwóch miejsc — albo:

```
Koniec := GetMaxY Div 2;
```

— albo:

```
For I := 1 To (Koniec Div 2) Do
```

Jeśli chcemy przyspieszyć poruszanie się po ekranie, nie musimy rysować co jeden piksel. Można skok ustawić na 2, 3, 5 pikseli. Jeśli natomiast chcemy zwolnić animację, wystarczy dodać dłuższy czas zwłoki, na przykład:

```
Delay(100);
```

— po każdej pętli. To samo rozwiązanie zapisane w C przedstawia kolejny listing.

Listing D04.C

```
#include <conio.h>
#include <graphics.h>
#include <dos.h>              // prototyp delay()
```

```
main()
{
  int Karta = DETECT, Tryb, I, Koniec;
  initgraph(&Karta, &Tryb, "C:\\TC\\BGI");    // UWAGA: podwojne ukosniki
  setbkcolor(BLUE);
  Koniec=getmaxy();
  for (I=1; I<=Koniec; I++)
  {
    setcolor(WHITE);
    line(50, I, 300, I);
    circle(I, 200, 20);
    delay(10);
    setcolor(BLUE);
    line(50, I, 300, I);
    circle(I, 200, 20);
  }
  closegraph();
  getch();
  return 0;
}
```

Mechanizm animacji czarne-białe-czarne-białe (bo tak chyba najprościej można by nazwać tę metodę) jest na tyle prosty, że nie wymaga zapewne dodatkowych wyjaśnień.

2.2. Animacja — sposób 2

Drugi prosty sposób animacji polega na zmianie aktywnej i widocznej strony pamięci karty obrazu. Karty sterownika graficznego mogą w swojej pamięci odwzorowywać więcej niż jedną stronę ekranu. Wszystkie operacje graficzne polegają w istocie na odpowiednim zapisie danych dotyczących obrazu w pamięci karty. Pamięć karty jest następnie odwzorowywana na ekranie monitora.

Ponieważ nie ma nic za darmo, zawsze mamy do wyboru albo większą rozdzielczość odwzorowania obrazu, albo większą liczbę stron i tym samym większą szybkość odwzorowania obrazu. Liczba dostępnych stron zależy zatem od rodzaju karty i od wybranego trybu graficznego. Sekwencja animacji w Pascalu powinna wyglądać tak:

```
SetActivePage(0);
{Rysujemy pierwsze stadium}
```


programowanie_strukturalne_DODATEK.indb 53 2010-07-05 11:54:46

```
SetActivePage(1);
{Rysujemy drugie stadium}

SetActivePage(2);
{Rysujemy trzecie stadium}

SetActivePage(3);
{Rysujemy czwarte stadium ruchu}

For i:= 0 To 3 Do               { wyświetlamy cztery kolejne fazy ruchu }
Begin
 SetVisualPage(i);
 Delay(50);                     { określenie tempa (tu: zwłoka 50 milisekund) }
End;
```

Jeśli chcemy użytkownikowi dać więcej czasu na obejrzenie poszczególnych rysunków, można w pętli umieścić oczekiwanie na klawisz:

```
For i := 0 To 3 Do
Begin
 SetVisualPage(i);
 ReadKey;
End;
```

Jeśli użytkownik pracował (np. czytał) ze stroną domyślną nr 0, podczas gdy na pozostałych przygotowywaliśmy rysunki, można wyświetlić tylko strony 1 – 3:

```
For i := 1 To 3 Do
Begin
 SetVisualPage(i);
 ReadKey;
End;
```

Aby nie komplikować przykładu, zamiast rysować coś bardzo ambitnego, w programie przykładowym poniżej po prostu został wyprowadzony tekst. Aby było widać dokonującą się prostą animację, na każdej stronie tekst znajduje się w innej pozycji. Animacja jest w pierwszej pętli powolna, w drugiej — szybsza. Tryb niskiej rozdzielczości `VGALo` wybrany został dlatego, że znaki są większe, a dostępnych stron pamięci ekranu jest 4 (o numerach 0 – 3). Strona nr 0 jest domyślną stroną ekranu, widoczną i aktywną.

Listing D05.PAS

```
Program ZMIANA_STRON;
Uses Crt, Graph;
Var Karta, Tryb, i: Integer;

BEGIN
 Karta:=VGA;
 Tryb:=VGALo;
 InitGraph(Karta, Tryb, 'C:\TP\BGI');
 For i:=1 To 4 Do
 Begin
  SetActivePage(i);
  MoveTo(i, i*10);
  OutText('XXXX');
  SetVisualPage(i);
  Delay(500);
 End;
 For I:=0 To 60 Do
 Begin
  SetVisualPage(I);
  Delay(100);
 End;
 CloseGraph;
END.
```

Poniżej został zaprezentowany ten sam program przygotowany w C.

Listing D06.C

```
#include <graphics.h>
#include <conio.h>
#include <dos.h>
main()
{
  int Karta = VGA, Tryb = VGALO, i;
  initgraph(&Karta, &Tryb, "C:\\TC\\BGI");
  for (i=1; i<5; i++)
  {
    setactivepage(i);
```


programowanie_strukturalne_DODATEK.indb 55 2010-07-05 11:54:46

```
      moveto(i, i*10);
      outtext("XXXX");
      setvisualpage(i);
      delay(1000);
    }
    for (i=0; i<=60; i++)
    {
      setvisualpage(i);
      delay(100);
    }
    closegraph();
    return 0;
  }
```

Taką technikę pracy można zastosować również wtedy, gdy mamy do wykonania jeden lub kilka dość skomplikowanych rysunków, a nie chcemy drażnić użytkownika oczekiwaniem. Dopóki użytkownik czyta tekst na widocznej właśnie stronie nr 0, na stronach 1, 2 i 3 mogą wykonywać się nawet bardzo skomplikowane rysunki czy wykresy, które zostaną momentalnie przywołane na ekran (jako całkowicie gotowe) na życzenie użytkownika, na przykład po naciśnięciu odpowiedniego klawisza. I nie ma to nic wspólnego z oszukiwaniem! To po prostu sztuka programowania.

2.3. Animacja — sposób 3

Trzeci sposób polega na zapamiętaniu wycinka obrazu z ekranu (dokładniej rzecz biorąc, polega to na przepisaniu zawartości pamięci obrazu — z pamięci karty do pamięci operacyjnej komputera), a następnie odtwarzaniu tego wycinka w wybranym miejscu ekranu. Będą nam do tego potrzebne następujące narzędzia:

```
GetImage();        { pobierz obrazek (wycinek ekranu) }
PutImage();        { wstaw wycinek we wskazane miejsce na ekranie }
GetMem();          { przydziel pamiec operacyjna dla wycinka obrazu }
```

Zarówno Turbo Pascal, jak i Turbo C++ pozwalają wskazać określone miejsce w pamięci za pomocą wskaźników. Najpierw, posługując się niewidoczną stroną (by nie zaśmiecać sobie ekranu), narysujemy okrąg i wypełnimy go krateczką, by wycinek rzucał się w oczy. Można to zrobić na przykład tak:

```
SetActivePage(1);
SetFillStyle(XHATCH_FILL, WHITE);      { gruba biala kratka }
Circle(100, 100, 10);                  { okrag bialy (domyslnie) }
FloodFill(100, 100, WHITE);            { wypelniamy kratka az do bialego obrzeza }
```


programowanie_strukturalne_DODATEK.indb 56 2010-07-05 11:54:46

Prostokąt o współrzędnych górnego lewego narożnika:

```
X1 = 90
Y1 = 90
```

— oraz dolnego narożnika:

```
X2 = 110
Y2 = 110
```

— pokryje zakratkowane kółko. Taki też fragment wytnie z ekranu funkcja:

```
GetImage(90, 90, 110, 110, ...);
```

Zanim jednakże przystąpimy do wycinania, powinniśmy mieć przygotowaną pamięć na obrazek. Zrobi to dla nas funkcja `GetMem()` (ang. *Get Memory* — dosłownie pobierz pamięć). Aby wszystko odbyło się zgodnie z zasadami, powinniśmy na początku programu zadeklarować zmienne. Nie musimy się przejmować szczegółami. W Pascalu i C/C++ mamy do dyspozycji gotową funkcję, która obliczy dla nas ilość pamięci potrzebną do przechowywania obrazka:

```
Wielkosc = ImageSize(90, 90, 110, 110);
```

Nie można się bez tego obejść, ponieważ w zależności od rozdzielczości i stosowanej liczby kolorów ilość pamięci potrzebna dla zapamiętania wycinka może być różna. Adres pamięci na obraz też gdzieś trzeba zapamiętać. Posłużymy się więc wskaźnikiem `P`:

```
Var P: Pointer;       {w Pascalu}
int *Wskaznik;         // w C
```

Teraz, nie wnikając już w szczegóły techniczne, możemy resztę pracy zlecić kompilatorowi.

Najpierw rozpatrzmy konstrukcję w Turbo Pascalu:

```
Var
 Karta, Tryb: Integer;
 P: Pointer;
 Wielkosc: Word;
BEGIN
 Karta := VGA;
 Tryb := VGALo;
 InitGraph(Karta, Tryb, 'C:\TP\BGI');
 ...
 Wielkosc := ImageSize(90, 90, 110, 110);
 GetMem(P,  Wielkosc);
 GetImage(90, 90, 110, 110,  P^);
```

Klawisz szóstki na klawiaturach starszego typu zamiast „daszka” ^ zawierał strzałkę. Informatycy są konserwatywni. Taka strzałka po literze P oznaczała „to coś, na co pokazuje wskaźnik P”. Strzałka zniknęła, ale obyczaj używania tego klawisza pozostał. By zgodnie z zasadą czarnego rysunku na czarnym tle wykonać animację, będziemy potrzebować jeszcze drugiego kwadracika — czarnego — do „zamazywania”. Możemy go również przygotować na niewidocznej stronie pamięci, wycinając dowolny kwadrat z czarnego tła. Obowiązują nas oczywiście wszystkie opisane wcześniej ograniczenia. Musimy przygotować dla niego miejsce w pamięci i wskaźnik, który będzie go wskazywał. Przecież będziemy go potrzebować wiele razy.

```
Var
 Karta, Tryb: Integer;
 P1, P2: Pointer;
 Wielkosc1, Wielkosc2: Word;
BEGIN
 Karta := VGA;
 Tryb := VGALo;
 InitGraph(Karta, Tryb, 'C:\TP\BGI');
  ...
 Wielkosc1 := ImageSize(90, 90, 110, 110);
 GetMem(P1,  Wielkosc1);
 GetImage(90, 90, 110, 110, P1^);
 Wielkosc2 := ImageSize(200, 200, 220, 220);
 GetMem(P2,  Wielkosc2);
 GetImage(200, 200, 220, 220, P2^);
```

Teraz wystarczy już tylko przełączyć się na aktywną stronę nr 0 (domyślnie strona widoczna) i posługując się pętlą, wstawiać obrazki przy zmieniających się współrzędnych:

```
SetActivePage(0);
For i := 1 To GetMaxY Do
Begin
 PutImage(i, i, P1^, NormalPut);
 PutImage(i, i, P2^, NormalPut);
End;
```

Sposób wstawiania obrazka (tu: NormalPut) oznacza jego nadpisywanie na poprzednią zawartość tego fragmentu ekranu, a współrzędne i, i to bieżące współrzędne lewego górnego narożnika prostokąta z wycinkiem. Teraz możemy już złożyć program w całość.

programowanie_strukturalne_DODATEK.indb 58 2010-07-05 11:54:46

Listing D07.PAS

```
Program ANIMACJA3;
Uses Crt, Graph;
Var
 Karta, Tryb, i: Integer;
 P1, P2: Pointer;
 Wielkosc1, Wielkosc2: Word;
BEGIN
 Karta := VGA;
 Tryb := VGALo;
 InitGraph(Karta, Tryb, 'C:\TP\BGI');
 SetActivePage(1);
 SetFillStyle(XHatchFill, White);
 Circle(100, 100, 10);
 FloodFill(100, 100, White);
 Wielkosc1 := ImageSize(90, 90, 110, 110);
 Wielkosc2 := ImageSize(200, 200, 220, 220);
 GetMem(P1, Wielkosc1);
 GetMem(P2, Wielkosc2);
 GetImage(90, 90, 110, 110, P1^);
 GetImage(200, 200, 220, 220, P2^);
 SetActivePage(0);
 For i := 1 To GetMaxY Do
 Begin
  PutImage(i, i, P1^, NormalPut);
  Delay(50);
  PutImage(i, i, P2^, NormalPut);
 End;
 CloseGraph;
END.
```

Kółeczko z kratkami będzie wędrować ukośnie w prawo w dół.

A teraz to samo zadanie wykonamy w C. Zwróćmy uwagę na sposób deklarowania i zastosowania wskaźnika. Funkcję `GetMem()` Pascala trzeba zastąpić przez `malloc()`, a stałą `NormalPut` — przez stałą `COPY_PUT`.

programowanie_strukturalne_DODATEK.indb 59 2010-07-05 11:54:47

Listing D08.C

```
#include <stdlib.h>
#include <dos.h>
#include <graphics.h>
main()
{
  int Karta = VGA, Tryb = VGALO, i;
  void *P1, *P2;
  unsigned Wielkosc1, Wielkosc2;
  initgraph(&Karta, &Tryb, "C:\\TC\\BGI");
  setactivepage(1);
  setfillstyle(XHATCH_FILL, WHITE);
  circle(100, 100, 10);
  floodfill(100, 100, WHITE);
  Wielkosc1 = imagesize(90, 90, 110, 110);
  Wielkosc2 = imagesize(200, 200, 220, 220);
  P1 = malloc(Wielkosc1);
  P2 = malloc(Wielkosc2);
  getimage(90, 90, 110, 110, P1);
  getimage(200, 200, 220, 220, P2);
  setactivepage(0);
  for (i=1; i<=getmaxy(); i++)
  {
    putimage(i, i, P1, COPY_PUT);
    delay(50);
    putimage(i, i, P2, COPY_PUT);
  }
  closegraph();
  return 0;
}
```

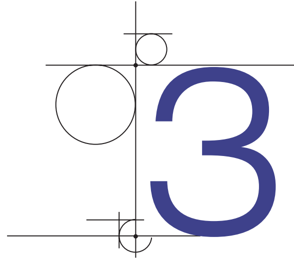
UWAGA

Turbo Pascal i Turbo C++ posługują się tymi samymi funkcjami graficznymi i tą samą biblioteką graficzną.

W środowisku Windows jest podobnie, choć tam, zamiast domyślnie całego ekranu, operuje się oknem aplikacji. Czyszczenie i rysowanie od nowa wycinka okna lub całego okna oparte jest na pojęciach:

- `InvalidateRect` — uznaj prostokątny obszar za nieaktualny,
- `InvalidateWindow` — uznaj obszar okna za nieaktualny,
- `Paint` — narysuj okno,
- `Refresh` — odśwież rysunek.

programowanie_strukturalne_DODATEK.indb 61 2010-07-05 11:54:47



Pascal — komputerowe modelowanie

3.1. Kiedy działa i jak działa statystyka?

Do różnorodnych średnich statystycznych trzeba podchodzić z pewną rezerwą i ze zrozumieniem ich istoty. Każdy wie, że prawdopodobieństwo uzyskania 1, 2, 3 czy 6 przy rzucie kostką do gry jest równe $^1/_6$. Przy symulacji procesów losowych programiści często jednakże posługują się dodatkowo pojęciem *średniej wartości oczekiwanej*, będącej analogią do matematycznego pojęcia względnej *częstotliwości występowania zjawisk losowych* oraz *nadziei matematycznej*. Kłopot polega jednakże na tym, że matematycy pomagają sobie pojęciami nieskończoności, granicy, ciągłości, nieskończenie małego przyrostu itp. W praktyce natomiast mamy do czynienia zazwyczaj ze skokowymi zmianami i skończoną liczbą możliwych do wykonania prób.

Problem w uproszczeniu można przedstawić tak: skoro z równym prawdopodobieństwem można uzyskać 1, 2, 3, 4, 5, 6, to jakiej średniej wartości wyrzucanych oczek można spodziewać się przy dużej liczbie rzutów? Obliczenie jest proste: (1 + 2 + 3 + 4 + 5 + 6) / 6 = 21 / 6 = 3,5

Nie znaczy to jednak, że w którymkolwiek pojedynczym rzucie możemy spodziewać się wyniku 3,5! Zwróćmy uwagę, że zmienna — wylosowana liczba oczek — ma krok dyskretyzacji = 1 (tylko wartości całkowite), a jej średnia wartość oczekiwana jest liczbą rzeczywistą. Często pomijane w tekście (a przyjmowane domyślnie słowo „średnio”) powoduje, że wynik pozornie koliduje z rzeczywistością.

Drugim przyjmowanym domyślnie założeniem („przy dużej liczbie rzutów”) zajmiemy się w kolejnej symulacji. Liczbę wylosowanych oczek będziemy dodawać do zmiennej `Suma`, a po każdym rzucie podzielimy `Sumę` przez liczbę rzutów i wyznaczymy średnią.

```
Suma := 0;
Randomize;
.....
For Nr_rzutu := 1 To 120 Do
Begin
 wynik := Random(6) + 1;               {losowa: 1 ... 6}
 Suma := Suma + wynik;
 Srednia := Suma / Nr_rzutu;
```

W porównaniu z poprzednimi wykresami zmniejszymy też odpowiednio skalę wykresu, by zmiany wartości średniej były wyraźnie widoczne (jedno oczko to 30 pikseli):

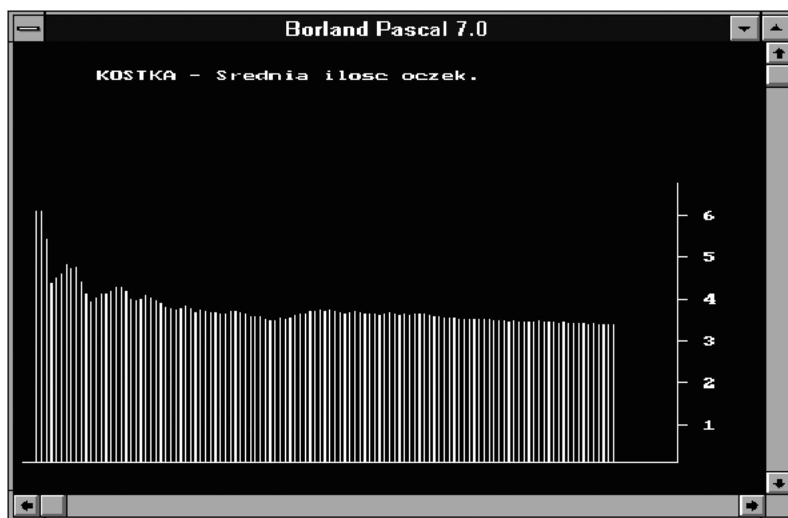
```
 OutTextXY(400, 270, '- 1');
 OutTextXY(400, 240, '- 2');
 OutTextXY(400, 210, '- 3');
 OutTextXY(400, 180, '- 4');
 OutTextXY(400, 150, '- 5');
 OutTextXY(400, 120, '- 6');
 .....
 MoveTo(10 + Nr_rzutu * 3, 300);
 LineTo(10 + Nr_rzutu * 3, 300 - Round( Srednia  * 30 ));
```

Po wprowadzeniu tych modyfikacji program w całości będzie wyglądał tak jak na poniższym listingu. Jego rezultat pokazano na rysunku D14.

Listing D09.PAS

```
Program Statystyka;
Uses Crt, Graph;
Var
 Suma, Nr_rzutu, wynik, Karta, Tryb : Integer;
 Srednia : Real;
BEGIN
 Karta := DETECT;
 Tryb := VGAHi;
 Initgraph( Karta, Tryb, 'C:\TP\BGI' );
 Suma := 0;
 Randomize;
 OutTextXY(50, 20, 'KOSTKA - Srednia liczba oczek.');
 MoveTo(5, 300);
```

```
 LineTo(400, 300);
 LineTo(400, 100);
 OutTextXY(400, 270, '- 1');
 OutTextXY(400, 240, '- 2');
 OutTextXY(400, 210, '- 3');
 OutTextXY(400, 180, '- 4');
 OutTextXY(400, 150, '- 5');
 OutTextXY(400, 120, '- 6');
 For Nr_rzutu := 1 To 120 Do
 Begin
  wynik := Random(6) + 1;            {losowa: 1 ... 6}
  Suma := Suma + wynik;
  Srednia := Suma / Nr_rzutu;
  MoveTo( 10 + Nr_rzutu * 3, 300 );
  LineTo( 10 + Nr_rzutu * 3, 300 - Round( Srednia  * 30 ) );
  ReadKey;
 End;
 CloseGraph;
END.
```



Rysunek D14. Jak widać z rysunku, w miarę wzrostu liczby rzutów średnia liczba oczek powoli i z pewnymi wahaniami zbliżała się jednak do oczekiwanej wartości średniej

Oprócz klasycznego szkolnego problemu kostki do gry rysunek ten ilustruje problem bardziej ogólny, pokazuje mianowicie, jak wolno następuje zbliżanie się do obszaru o największym prawdopodobieństwie podczas dziesiątek i setek eksperymentów. Wypływa stąd jeszcze jeden istotny wniosek praktyczny, że często dopiero bardzo duża liczba eksperymentów może w procesach stochastycznych dać wyniki o dużym prawdopodobieństwie poprawności i zgodności z rzeczywistością.

Zanim jeszcze zaczniemy opowieść o komputerowym modelowaniu, należy wyjaśnić pojęcia. Jeśli mianowicie najpierw mamy problem, a później (na podstawie programów komputerowych) tworzymy jakąś hipotezę, tezę, uogólnienie — nazywa się to **modelowaniem**. Jeśli hipoteza jest najpierw, a program komputerowy ma za zadanie sprawdzić ją i potwierdzić lub obalić — mówi się o **testowaniu hipotezy**.

3.2. Paradoks kwadransa akademickiego

Jako przykład prostego testowania omówimy **paradoks kwadransa akademickiego**. Nie ma tu żadnej rzeczywistej sprzeczności. Jest za to typowy błąd w rozumowaniu, popełniany przy pierwszej ocenie „na oko".

Wyobraźmy sobie, że osoba A umówiła się z osobą B na spotkanie w ciągu 1 godziny, na przykład pomiędzy 12:00 a 13:00. Każda z osób po przyjściu na miejsce spotkania będzie czekać równo 15 minut. Jeśli druga osoba w ciągu tych 15 minut nie pojawi się, pierwsza osoba odchodzi. Przyjmujemy, że prawdopodobieństwo przyjścia każdej z obu osób w dowolnym momencie pomiędzy 12:00 a 13:00 jest dokładnie takie samo (tj. chodzi tu o zdarzenia równo prawdopodobne).

Typowe uproszczone rozumowanie przebiega tak:

1. Skoro każda z osób może przyjść w dowolnym momencie między godziną 12 a 13 z takim samym prawdopodobieństwem, to prawdopodobieństwo dla każdego, dowolnie wybranego momentu dla osoby A (że właśnie jest na miejscu spotkania) wynosi: $15/60 = 1/4$.
2. Dla dowolnego momentu w ciągu tej godziny odpowiednie prawdopodobieństwo dla osoby B wynosi dokładnie tyle samo, czyli też $1/4$.
3. Nadejście (lub nie) obu osób to zdarzenia niezależne, więc prawdopodobieństwo, by znalazły się na miejscu spotkania jednocześnie, wynosi w dowolnym momencie $1/4 * 1/4 = 1/16$.

Rozumowanie jest słuszne tylko pozornie, a wynik stwarza zupełnie mylne wrażenie, jakoby sukces spotkania w takich warunkach był, delikatnie mówiąc, zdecydowanie mało prawdopodobny. Naprawdę jest zupełnie inaczej, co udowodnimy właśnie metodą testowania tej fałszywej hipotezy. Przy okazji w trakcie tych rozważań zademonstrowanych zostanie kilka pojęć i zagadnień dość charakterystycznych dla różnych metod komputerowego modelowania.

programowanie_strukturalne_DODATEK.indb 65 2010-07-05 11:54:47

3.2.1. Fragmentaryczne testowanie hipotezy

Zazwyczaj każde twierdzenie matematyczne lub hipoteza opierają się na jakichś założeniach. Fragmentaryczne testowanie polega na oczywistej zasadzie: „Jeśli uda się obalić choćby jedno założenie, hipoteza może być uznana za dalece wątpliwą i odrzucona". Co prawda, zgodnie z zasadami klasycznej logiki matematycznej, fałszywa implikacja (wnioskowanie) oparta na fałszywych założeniach może dać prawdziwy wynik końcowy, ale nas interesuje tu jedynie przypadek, gdy poprawne wnioskowanie, oparte na prawdziwych założeniach, daje prawdziwy wniosek końcowy.

Zatem do dzieła. Jeśli po sprawdzeniu okaże się, że na przykład prawdopodobieństwo obecności osoby A (lub B) w dowolnym momencie nie jest takie samo (i zawsze równe ¼), wystarczy to, by stwierdzić, że testowanie hipotezy wypadło negatywnie, a sama hipoteza może być słusznie odrzucona jako fałszywa (lub co najmniej ryzykowna).

Jak w praktyce sprawdzić taką uproszczoną hipotezę roboczą (tj. jeden z warunków)? To oczywiste — napisać program, w którym policzymy, ile razy osoba A była na miejscu spotkania osiągalna w kolejnych chwilach pomiędzy godziną 12 a 13. Jest tu pewien drobny kłopot: tych chwil jest nieskończenie wiele. Możemy uprościć sobie zadanie, dzieląc przedział czasu na przykład na jednominutowe odcinki.

3.2.2. Algorytm wizualizacji

Aby sprawdzić założenie, zastosujemy 61-elementową tablicę (jednowymiarową macierz). Każdy element tablicy będzie odpowiadał pojedynczej minucie. Zmienna `X` będzie określać moment pojawienia się osoby, tj. numer minuty po godzinie 12.00, a więc:

`X = 0` odpowiada `12:00`,

`X = 1` odpowiada `12:01`, itd.

`----`

`X = 60` odpowiada `13:00`

Skoro pojawienie się na przykład o 12:21 (`X = 21`) odpowiada obecności (czekaniu) od 12.21 do 12.36, to dla `X = 21` należy dodać 1 do wszystkich minut-elementów tablicy `m[21]` – `m[36]`, zaznaczając w ten sposób „obecność" w trakcie tych minut.

```
For X := 0 To 60 Do
   For i := X To X+15 Do
      Begin
        m[i] := m[i] + 1;
      End;
```


programowanie_strukturalne_DODATEK.indb 66 2010-07-05 11:54:47

Zapis powyższy wygląda zupełnie niewinnie, ale jest przykładem jednego z typowych szkolnych błędów. Przy X równym na przykład 55 pętla wewnętrzna będzie wymagać wykonania inkrementacji dla elementów tablicy:

```
m[55] .... m[70]     (!!!)
```

— a takie elementy przecież nie istnieją! Taka próba grozi katastrofą. Niektóre systemy operacyjne, takie jak Windows XP, przerwą wykonanie programu i wyświetlą komunikat, że nastąpiła próba wykonania niedozwolonej operacji. Mniej stabilne systemy, takie jak MS-DOS, mogą po prostu kompletnie się zawiesić. Co robić w takich sytuacjach? To proste — nie pozwolić komputerowi na próby sięgania do nieodpowiednich bajtów:

```
For X := 0 To 60 Do
   Begin
    For i := X To X+15 Do
      Begin
        If i <= 60 Then m[i] := m[i] + 1;
      End;
   End;
```

Teraz operacja jest dopuszczalna tylko dla `i <= 60`.

Wizualizację wyników przeprowadzimy za pomocą wykresu słupkowego. Poniżej zaprezentowano kod symulacji w całości.

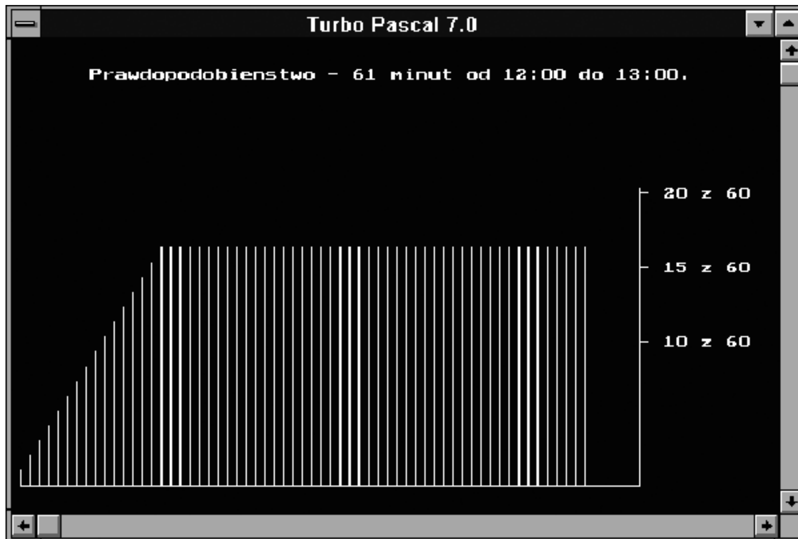
Listing D10.PAS

```
Program Symulacja1;
Uses Crt, Graph;
Var
 Karta, Tryb, X, i, WspX0, WspY0 : Integer;
 m : Array[0..60] of Integer;

BEGIN
 {Inicjujemy tablice minut.}
 For i := 0 To 60 Do m[i] := 0;
  {Zliczamy przypadki obecnosci w poszczegolnych minutach.}
  For X := 0 To 60 Do
  Begin
   For i := X To X+15 Do
   Begin
    If i <= 60 Then m[i] := m[i] + 1;     { UWAGA! }
```

```
     End;
    End;
    {Inicjujemy tryb graficzny.}
    Karta := DETECT;
    Tryb := VGAHi;
    Initgraph( Karta, Tryb, 'C:\TP\BGI' );
    If GraphResult <> 0 Then Halt
    Else
    Begin
     {Rysujemy uklad wspolrzednych dla wykresu.}
     OutTextXY(50, 20, 'Prawdopodobienstwo - 61 minut od 12:00 do 13:00.');
     MoveTo(5, 300);
     LineTo(400, 300);
     LineTo(400, 100);
     OutTextXY(400, 200, '- 10 z 60');
     OutTextXY(400, 150, '- 15 z 60');
     OutTextXY(400, 100, '- 20 z 60');
     {Rysujemy slupki dla kazdej minuty.}
     WspX0 := 5;
     WspY0 := 300;
     For i := 0 To 60 Do
     Begin
      MoveTo( WspX0 + 6*i, WspY0 );
      LineTo( WspX0 + 6*i, WspY0 - 10*m[i] );
     End;
     ReadKey;
     CloseGraph;
    End;
  END.
```

Przy zapisanym w taki sposób algorytmie w istocie nakazujemy rozpatrywanej osobie wykazać więcej cierpliwości, czekać nie 15, lecz 15:59,99 minuty (tj. 16 minut) i rozważamy w rzeczywistości nie 60, lecz 61 minut. Widać to na wykresie z rysunku D15.

programowanie_strukturalne_DODATEK.indb 68 2010-07-05 11:54:47



Rysunek D15. Model pokazujący, jak zmienia się prawdopodobieństwo

W tym momencie nie to jednak jest dla nas najważniejsze. Najprostszą hipotezę (¼) można obalić, gdyż gołym okiem widać, że choć prawdopodobieństwo pojawienia się obu osób w dowolnej chwili jest identyczne, to prawdopodobieństwo obecności na miejscu spotkania wcale nie jest takie samo dla każdej spośród rozpatrywanych 60 lub 61 minut.

Z przykładu i z wykresu widać wyraźnie, że nie zawsze jest wszystko jedno, na jakie kawałki podzielimy rozpatrywane ciągłe zakresy.

3.3. Problem spotkania — uogólnienie

Tak zwany problem spotkania jest nad wyraz typowy. Miewamy z nim do czynienia często w życiu codziennym. Warunkuje on powodzenie misji kosmicznych, działanie urządzeń wykorzystujących koniunkcję dwóch lub wielu sygnałów sterujących itp. Wykorzystamy teraz komputer do sprawdzenia prawie wszystkich potencjalnych sytuacji („prawie", dlatego że możliwości jest nieskończenie wiele, niestety). Oba poniższe przykłady ilustrują jednakże podstawową zaletę komputerów. Jeśli mianowicie zamienimy „nieskończenie wiele" na „bardzo, bardzo dużo", komputer będzie mógł wykazać się swoimi zaletami i prawdziwą potęgą. Przeanalizowanie tysięcy iteracji przerasta cierpliwość i wytrzymałość najbardziej zaciekłych matematyków, ale współczesnym komputerom zajmuje to zaledwie ułamki sekund, ba, one to podobno lubią i są w tym naprawdę dobre...

W pierwszej kolejności zajmiemy się opisaną wcześniej sytuacją. Przypomnijmy. Dwie osoby umówiły się na spotkanie między godziną 12:00 a 13:00. Ta, która przyjdzie pierwsza, czeka 15 minut (¼ h), po czym odchodzi. Należy określić prawdopodobieństwo, że spotkanie powiedzie się, przy założeniu, że każda spośród tych dwu osób po-

programowanie_strukturalne_DODATEK.indb 69 2010-07-05 11:54:47

jawi się z równym prawdopodobieństwem w dowolnym momencie pomiędzy godziną 12:00 a 13:00.

3.3.1. Sposób rozwiązania: analityczne zliczanie bez wizualizacji

Założenia — wyjściowe parametry iteracji

Podręczniki klasycznego rachunku prawdopodobieństwa podają tu jako poprawne rozwiązanie P = $^{7}/_{16}$. Przetestujmy tę hipotezę.

Co prawda rozpatrywane osoby mogą przybyć na miejsce spotkania z równym prawdopodobieństwem w każdym ułamku każdej sekundy (tylko ciągłość i nieskończenie wiele nieskończenie małych odcinków czasu przedstawia realny obraz z absolutną dokładnością), ale z opisanych poniżej przyczyn komputer by tego nie zrozumiał, więc będziemy symulować sytuację z krokiem iteracji równym 1 minucie.

```
Var
 T1,         {Czas przybycia 1. osoby: 12.00, 01, 02, ... 12.60}
 T2,         {Czas przybycia 2. osoby}
 Proby,       {Ogolna liczba wszystkich rozwazonych kombinacji}
 OK   :  Integer;  {Liczba prob zakonczonych powodzeniem — SPOTKANIE „OK”}

BEGIN
 Proby := 0;        {Inicjujemy zmienne}
 OK := 0;
 For T1 := 0 To 60 Do     {T1 = 12.00 ... 13.00}
  For T2 := 0 To 60 Do     {T2 = 12.00 ... 13.00}
  Begin ...
```

Przy takich założeniach i oznaczeniach zapis samego algorytmu testującego hipotezę będzie stosunkowo łatwy.

Algorytm we wnętrzu pętli iteracyjnych

Możliwe są cztery sytuacje.

1. Osoba 1. przyszła wcześniej lub równo z drugą (`T2 >= T1`):
   a) różnica czasów była nie większa niż 15 minut: (`T2 >= T1`) AND (`T2 <= T1 + 15`),
   b) w przeciwnym razie do spotkania nie doszło.
2. Osoba 2. przyszła wcześniej niż pierwsza (`T1 > T2`):
   a) różnica czasów była nie większa niż 15 minut: (`T1 > T2`) AND (`T1 <= T2 + 15`),
   b) w przeciwnym razie do spotkania nie doszło.

programowanie_strukturalne_DODATEK.indb 70 2010-07-05 11:54:47

Aby udane spotkanie zostało za każdym razem poprawnie zliczone, algorytm możemy zapisać następująco:

```
For T1 := 0 To 60 Do
 For T2 := 0 To 60 Do
 Begin
  If ( (T2 >= T1) And (T2 <= T1 + 15)) Then OK := OK + 1;
  If ( (T1 > T2) And (T1 <= T2 + 15)) Then OK := OK + 1;
  Proby := Proby + 1;
 End;
```

Użycie `And` tworzy koniunkcję (iloczyn logiczny) obu warunków, które muszą być spełnione jednocześnie, by można było „odnotować sukces".

Sposób reprezentacji wyników

Dobrą praktyką w programach tego typu jest drukowanie dla użytkownika dodatkowej informacji o tym, co zostało sprawdzone, tak aby nie musiał gubić się w domysłach, cóż to za dziwne szeregi cyfr, ani odchodzić od komputera w poszukiwaniu książki z instrukcjami. Szanując tę zasadę, możemy wydrukować rezultaty na przykład tak:

```
WriteLn('REZULTATY:');
WriteLn('Komputer wykonal: ', Proby, ' iteracji.');
WriteLn('Wynik = ', OK / Proby );
WriteLn('Dla porownania 7/16 to: ',  7/16);
```

Możemy oczywiście śledzić pracę programu na bieżąco, drukując wyniki cząstkowe z wnętrza każdej dowolnie wybranej pętli, w tym przypadku jednak byłyby to tysiące cyferek przewijane przez dziesiątki ekranów, więc program raczej straciłby, niż zyskał na komunikatywności.

3.3.2. Kod komputerowej symulacji w Pascalu

Z opisanych powyżej fragmentów możemy poskładać końcowy kod źródłowy całego programu. Nie zawiera on żadnych nowych elementów, poza kosmetycznymi uzupełnieniami, ułatwiającymi jego obsługę.

Listing D11.PAS

```
Program Spotkanie;
Uses Crt;
Var T1, T2, Proby, OK : Integer;

BEGIN
 ClrScr;
```


programowanie_strukturalne_DODATEK.indb 71 2010-07-05 11:54:47

```
  Proby := 0;
  OK := 0;

  For T1 := 0 To 60 Do
   For T2 := 0 To 60 Do
   Begin
    If ( (T2 >= T1) AND (T2 <= T1 + 15)) Then OK := OK + 1;
    If ( (T1 > T2) AND (T1 <= T2 + 15)) Then OK := OK + 1;
    Proby := Proby + 1;
   End;
  WriteLn('REZULTATY:');
  WriteLn('Komputer wykonal: ', Proby, ' iteracji.');
  WriteLn('Wynik = ', OK / Proby );
  WriteLn('Dla porownania 7/16 to: ',  7/16);
  ReadKey;
END.
```

Można łatwo przekonać się, że hipoteza została potwierdzona z dokładnością, która w większości przypadków okaże się zupełnie wystarczająca. Możesz samodzielnie upewnić się, czy zmiana kroku iteracji spowoduje zwiększenie dokładności odwzorowania, należy tu jednak zwrócić uwagę, że iteracja na przykład co sekundę wymaga zmiany typów deklarowanych zmiennych na `Longint`, czyli „długie liczby całkowite". Liczba iteracji wzrośnie wtedy dość znacznie, bo aż około 3600 razy.

3.3.3. Komputery nie mylą się w liczeniu

Na tym względnie prostym przykładzie można łatwo przekonać się, że w programach takiego typu istnieje pewna specyficzna „grupa ryzyka". Leży ona w ludzkiej naturze: to (dokładnie odwrotnie niż z komputerami) wielce poniekąd cenna skłonność do uogólnień, abstrahowania i szybkich szacunków. Bardzo często występującym, a trudnym do dostrzeżenia i wychwycenia błędem bywają nierówności w warunkach algorytmu. Czytelnicy mogą łatwo przekonać się, że zastąpienie na przykład:

```
If ( (T2 >= T1) And (T2 <= T1 + 15)) Then OK := OK + 1;
If ( (T1 >= T2) And (T1 <= T2 + 15)) Then OK := OK + 1;
```

— przez:

```
If ( (T2 > T1) And (T2 < T1 + 15)) Then OK := OK + 1;
If ( (T1 > T2) And (T1 < T2 + 15)) Then OK := OK + 1;
```

— spowoduje dość znaczne, a w każdym razie wyraźnie zauważalne różnice między uzyskiwanymi w programie końcowymi rezultatami. Należy tu pamiętać, że w przy-

programowanie_strukturalne_DODATEK.indb 72 2010-07-05 11:54:47

padku zastosowania nieostrej nierówności liczba kroków iteracji zmniejszy się prawie zawsze o 1, a w przypadku pętli wielokrotnie zagnieżdżonych (tj. tych najbardziej wewnętrznych) może to powodować znaczne różnice (nawet wielotysięczne) w ostatecznej liczbie pętli, które zostaną rzeczywiście wykonane.

3.4. Sformułowanie zadania — wersja druga

Urządzenie otrzymuje dwa sygnały wejściowe. Każdy moment nadejścia sygnału jest wewnątrz przedziału czasowego `<0...T>` jednakowo prawdopodobny. Urządzenie zadziała, jeśli odstęp czasowy pomiędzy nadejściem obu sygnałów będzie mniejszy niż `t (t < T)`. Należy określić prawdopodobieństwo zadziałania urządzenia w okresie `T`, jeśli wiadomo, że w okresie tym każdy spośród sygnałów wejściowych został wysłany dokładnie jeden raz.

Podręczniki klasycznego rachunku prawdopodobieństwa podają tu jako poprawne rozwiązanie:

$$P = \frac{T^2 - 2\frac{(T-t)^2}{2}}{T^2} = \frac{t(2T-t)}{T^2} \quad \text{(F1)}$$

I tym razem sprawdzimy poprawność tej hipotezy, posługując się dyskretną iteracją w zakresie 0 – T. Wykorzystamy jeszcze jedną zaletę komputera — poprzez zagnieżdżenie pętli programowych przeanalizujemy tendencję dla różnych wartości kroku iteracji.

3.4.1. Metoda wizualizacji prawdopodobieństwa

Ponieważ w tym przykładzie zarówno sama modelowana sytuacja jest nieco bardziej skomplikowana, jak i uogólnienie zależności wymusza wyższy stopień abstrakcji, zademonstrowana zostanie inna metoda często stosowana w praktyce rozwiązywania problemów takiej kategorii, czyli graficzna reprezentacja pól prawdopodobieństwa. Metodę stosuje się najczęściej w trzech podstawowych odmianach: 1D (liniowy), 2D (pole), 3D (figura przestrzenna). Powyższy przykład kwalifikuje się do reprezentacji na dwuwymiarowej płaszczyźnie, czyli nadaje się do zaprezentowania w wersji 2D. Metoda ta w ogólnym zarysie sprowadza się do przyjęcia dwu prostych założeń. Można ją zastosować:

- Jeśli funkcję opisującą rozkład prawdopodobieństwa da się zapisać w formie funkcji logicznej dwu niezależnych zmiennych, na przykład `P = {0, 1} = F(x, y)`.
- Jeśli zmienne niezależne `x`, `y` da się z wystarczającą dokładnością odwzorować z krokiem iteracji o wartości 1 piksela.

Można zbudować odwzorowanie pól prawdopodobieństwa na płaszczyźnie `XOY` i zwizualizować je bez uszczerbku dla dokładności odwzorowania na ekranie monitora.

Przetłumaczmy to teraz na ludzki język. Jeśli oznaczymy momenty nadejścia sygnałów przez: `0 <= x <= T` oraz `0 <= y <= T`, to cały obszar zmienności możemy przedstawić na płaszczyźnie `XOY` jako kwadrat o bokach `T * T`. Każdy spośród punktów tego kwadratu ma jakieś współrzędne `A1(x1, y1)`, czyli odpowiada jednej kombinacji czasów `x1`, `y1` nadejścia impulsów. Każdemu takiemu punktowi można dodatkowo przypisać wartość logiczną (typ `Boolean`) 1 lub 0 (inaczej `True` lub `False`), określającą jednoznacznie, czy taka kombinacja spełnia warunki zadania (interwał czasowy dostatecznie mały, urządzenie zadziała lub nie).

Dla naszego przykładu wykorzystanie pola 300×300 pikseli powinno okazać się zupełnie wystarczające, a to, że oś `Y` skierowana jest na ekranie w dół, w niczym nam nie przeszkadza. Logiczną funkcję przełączającą zrealizujemy w sposób najbardziej oczywisty — piksel spełniający kryteria zaświeci się na biało, piksele niespełniające wymagań pozostaną czarne (tak jest najczęściej w Turbo Pascalu, w Delphi zazwyczaj czyni się odwrotnie).

3.4.2. Oznaczenia — wyjściowe parametry iteracji

Aby powiodło się graficzne odwzorowanie, musimy oczywiście zainicjować tryb graficzny. Oprócz roboczych zmiennych `x`, `y` będziemy potrzebować stałej `T` (tu: 300 pikseli). Szablon programu będzie zatem wyglądał następująco:

```
Program PolaPrawdopodobienstwa;
Uses Graph, Crt;
Const ROLL = 50;                          {STALA:  Przesuniecie wykresu.}
Var Karta, Tryb, X, Y, T, tm : Integer;  {Zmienne pomocnicze trybu graficznego BGI}
                {Zmienne robocze z zakresu  0 ... T, czyli 0 ...300}
BEGIN
 Karta := DETECT;                         {Dostosuj sie do karty graficznej}
 Tryb := VGAHi;                           {Rozdzielczosc: wysoka, tryb DETECT -
ustawia domyslnie}
 InitGraph(Karta, Tryb, 'C:\TP\BGI');     {Zainicjuj tryb graficzny}
 T := 300;
 tm := 75;                                {Dwa zadane przedzialy czasu}

 ........
 ReadKey;                                 {Poczekaj, az obejrzymy wykres}
 CloseGraph;                              {Zamknij tryb graficzny przed wyjsciem
z programu}
END.
```

W miejscu wielokropka będziemy rozbudowywać roboczą część programu.

programowanie_strukturalne_DODATEK.indb 74 2010-07-05 11:54:48

3.4.3. Algorytm pętli iteracyjnych

W różnych rzeczywistych urządzeniach czasy `T` i `t` mogą przyjmować wartości z szerszego zakresu, zależnego najczęściej od tzw. stałych czasowych poszczególnych układów i systemów. W układach optoelektronicznych mogą to być piko- i nanosekundy, w układach elektronicznych — zazwyczaj nano- lub mikrosekundy, w urządzeniach i systemach mechanicznych czy hydraulicznych — minuty czy nawet godziny. Nie zmienia to jednak ani istoty zagadnienia, ani sposobu rozwiązania, choć dla wygody użytkownika można przeskalować rysunek.

W tym przykładzie, by nie komplikować i nie zaciemniać obrazu, przyjęto, że zmienne `x` i `y` będą zmieniać się co 1 piksel w zakresie 0 – 300. To, czy piksel ten odpowiada 1 nanosekundzie, czy 3 godzinom, nie ma tu znaczenia.

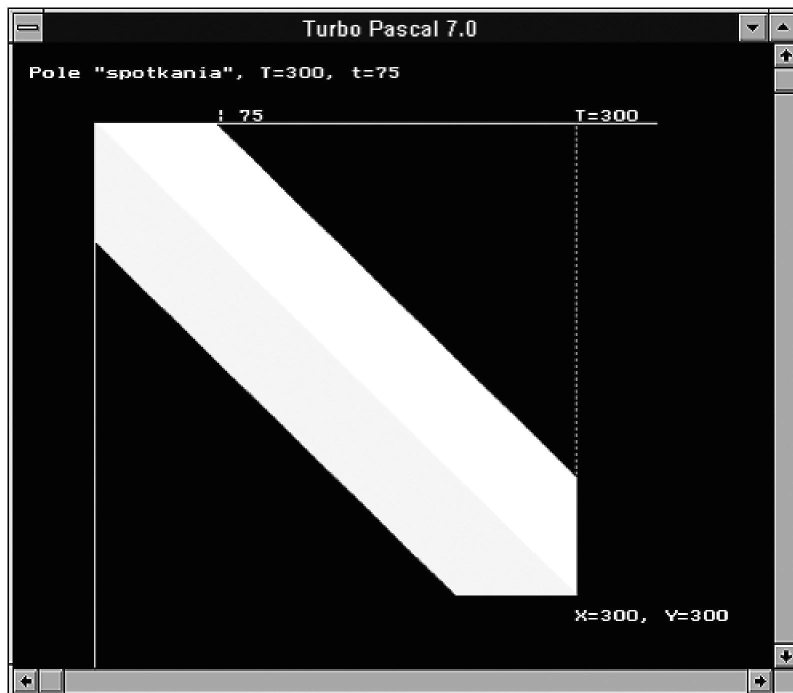
Dla przykładowego odwzorowania przyjmijmy: `T = 300`, `t = 75`. Zachowana została proporcja 1:4, by pozostać w obrębie bezpośredniej analogii do poprzednich rozważań.

Listing D12.PAS

```
Program PolaPrawdopodobienstwa;
Uses Graph, Crt;
Const ROLL = 50;                               {Przesuniecie wykresu}
Var Karta, Tryb, X, Y, T, tm : Integer;

BEGIN                                          {Inicjujemy tryb graficzny}
 Karta := DETECT;
 Tryb := VGAHi;
 Initgraph( Karta, Tryb, 'C:\TP\BGI' );
 If GraphResult <> 0 Then Halt
 Else
 Begin                                         {Rysujemy uklad wspolrzednych dla
wykresu}
  OutTextXY(10, 15, 'Pole "spotkania", T=300, t=75');
  OutTextXY( ROLL+75,  ROLL-8, '| 75'  );
  OutTextXY( ROLL+300, ROLL-8, 'T=300' );
  Line(ROLL, ROLL, ROLL, 400);
  Line(ROLL, ROLL, 400, ROLL);
  SetLineStyle(DottedLn, 0, NormWidth);        {Linia przerywana}
  Line(ROLL+300, ROLL, ROLL+300, ROLL+300);
  OutTextXY(ROLL+300, ROLL+310, 'X=300, Y=300');
  T := 300;                                    {SPRAWDZAMY WARUNEK}
```

```
      tm := 75;
      For X := 1 To T Do
       For Y := 1 To T Do
       Begin
        If ( Y <= X ) And ( (X-Y) <= tm )
        Then PutPixel ( ROLL+X, ROLL+Y, WHITE );
        If ( X <= Y ) And ( (Y-X) <= tm )
        Then PutPixel ( ROLL+X, ROLL+Y, YELLOW );
       End;
      ReadKey;
      CloseGraph;
     End;
    END.
```

Łatwo rozszerzyć ten program o zliczanie pikseli, pozwalające na przybliżone obliczenie prawdopodobieństwa w sposób analityczno-rachunkowy. Rysunek D16 przedstawia roboczy ekran programu.



Rysunek D16. Graficzna prezentacja ukazująca, jak rozkłada się prawdopodobieństwo

Na podstawie rysunku, który wykonał program symulujący, można z łatwością stwierdzić, że z punktu widzenia klasycznej geometrii analitycznej proste, które ograniczają pole, mają następujące równania:

```
y = x + t;
```

```
y = x - t;
```

Obliczenie pola takiej figury w sposób analityczny i porównanie jej z pełnym prawdopodobieństwem (prawdopodobieństwo „trafienia” w kwadrat `T * T`, czyli tu: 300×300, wynosi — zgodnie z założeniem — dokładnie 100%) pozwoli łatwo sprawdzić poprawność rozumowania teoretycznego. Komputerowe symulowanie i testowanie ma sens, a w wielu przypadkach może znacznie ułatwić i przyspieszyć poprawną ocenę rzeczywistych szans na sukces w różnych, bardziej skomplikowanych sytuacjach.

3.5. Czy cierpliwość popłaca?

Pytanie brzmi dość retorycznie, gdyż jednoznaczna odpowiedź bez dokładnej analizy poszczególnych przypadków może być trudna. Tym niemniej takie pytanie prawie każdy sobie czasami zadaje... Na zakończenie rozważań dotyczących paradoksu kwadransa rozwiążmy jeszcze jedno dość typowe zadanie. Zastanówmy się, jak długo należy czekać, by prawdopodobieństwo spotkania wzrosło do na przykład 50%, 90% itp.

Istnieją dwa możliwe rozwiązania: ręczne zliczanie (patrz wyżej) i sporządzenie wykresu funkcjonalnej zależności na podstawie analitycznej formuły funkcji. Gdy nie dysponujemy precyzyjną zależnością analityczną, ręczne zliczanie pozostaje jedyną alternatywą. W naszym jednakże przypadku mamy sprawdzony już wzór (F1). Istnieje tu wszakże jeszcze jeden dość typowy problem. Funkcję dwu niezależnych zmiennych `P = F( T, t )` należałoby wykreślać w konwencji 3D (tj. `z = f ( x, y );`). Tworzenie tablic trójwymiarowych i imitowanie trójwymiarowości na płaskim ekranie monitora jest w zasadzie możliwe, w tym jednak przypadku zależność można sprowadzić do płaszczyzny kartezjańskiej.

Po uważnym przyjrzeniu się zagadnieniu, łącznie ze wzorem (F1) i wykresami, można łatwo dostrzec, że skala bezwzględna nie ma dla prawdopodobieństwa żadnego znaczenia. Ważne jest nie to, czy `T` wynosi 60 minut, a `t` 15 minut, ani czy `T` jest równe 60 s, a `t` — 15 s. W obu tych przypadkach prawdopodobieństwa będą dokładnie równe. Ważny jest zatem jedynie stosunek wielkości, czyli proporcja `t / T` — i to ona decyduje o szansach na sukces. Przed przystąpieniem do tworzenia algorytmu można zatem zastąpić funkcję 3 zmiennych funkcją 2 zmiennych:

`P = F ( T, t )` --> `P = f ( k )` gdzie: `k = t / T`

{ `t` — oznaczane jest w programach przez `tm` }

zatem: `t = k * T`

programowanie_strukturalne_DODATEK.indb 77 2010-07-05 11:54:48

Formuła wyjściowa:

$$P = \frac{T^2 - 2\frac{(T-t)^2}{2}}{T^2} = \frac{t(2T-t)}{T^2} \quad \text{(F1)}$$

— po dokonaniu odpowiedniego podstawienia przyjmie bardziej jasną dla programisty postać:

$$P = \frac{kT(2T-kT)}{T^2} = \frac{2kT^2 - k^2T^2}{T^2} = 2k - k^2 \quad \text{(F2)}$$

Pochodna `P'(k) = 2 - 2k` dla `k` z przedziału `{0....1}` będzie zawsze nieujemna, więc nie możemy oczekiwać na wykresie naszej funkcji żadnego ekstremum. W tłumaczeniu na ludzki język oznacza to, że nie ma jakiejś szczególnie korzystnej proporcji `k = t / T`, która dawałaby nam maksymalnie duże prawdopodobieństwo sukcesu. Tym niemniej sprawdźmy, jak to naprawdę jest z tą cierpliwością, przy okazji demonstrując jeszcze jedną technikę bardzo przydatną przy rozpatrywaniu zagadnień matematycznych i inżynierskich.

Do tej pory stosowana była pętla `For` z domyślnym krokiem inkrementacji licznika równym +1. Pozwalało to ominąć zagadnienie dyskretyzacji ciągłej w istocie zmiennej. Tym razem krok iteracji zostanie odseparowany od skalowania rysunku. Pętlę programową skonstruujemy według następującego schematu:

a) zainicjowanie zmiennych,

b) słowo kluczowe `Repeat` („powtarzaj") — nagłówek/początek pętli,

c) ciało pętli — część obliczeniowa,

d) ciało pętli — zaokrąglenie i przeskalowanie wyników,

e) ciało pętli — rysowanie elementu wykresu,

f) ciało pętli — inkrementacja zmiennej (krok iteracji),

g) sprawdzenie warunku kontynuacji pętli — słowo kluczowe `Until` („dopóki").

Listing D13.PAS

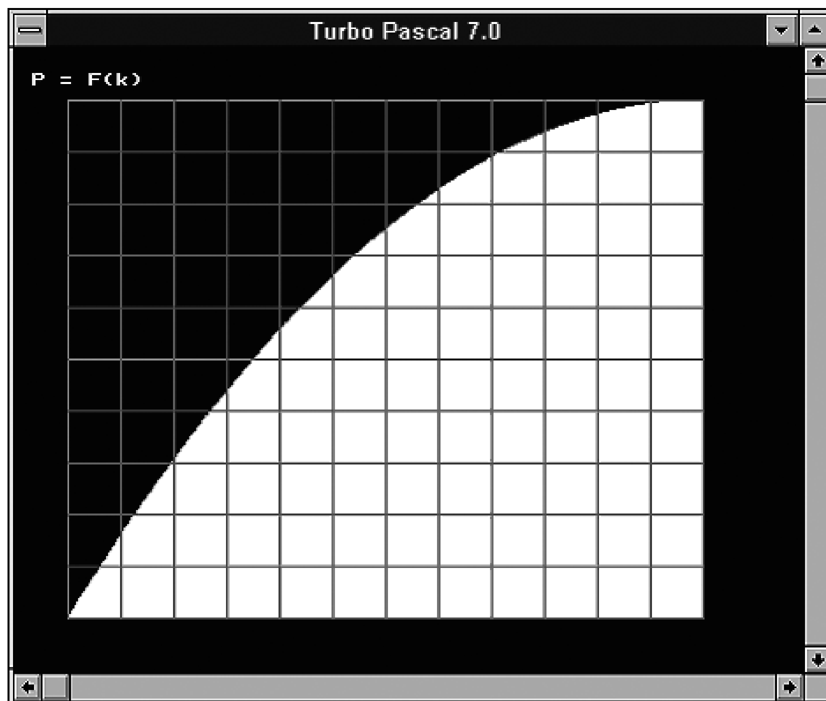
```
Program Cierpliwosc;
Uses Graph, Crt;
Var Karta, Tryb, x0, y0, X, Y, Skala_k, Skala_P : Integer;
 k, Dk, P, Xr, Yr : Real;

BEGIN
 Karta := DETECT;
 Tryb := VGAHi;
```

```
Initgraph( Karta, Tryb, 'C:\TP\BGI');
If GraphResult <> 0 Then Halt
Else
Begin
 OutTextXY(10, 15, 'P = F(k)');
 k := 0;                                   { a) }
 Dk := 0.001;
 Skala_k := 360;
 Skala_P := 300;
 x0 := 30;
 y0 := 330;
 Repeat                                    { b) }
 Begin
  P := 2*k - k*k;                          { c) }
  Xr := x0 + k * Skala_k;                  { d) }
  Yr := y0 - P * Skala_P;
  X := Round(Xr);
  Y := Round(Yr);
  MoveTo( X, y0 );                         { e) }
  LineTo( X, Y );
  k := k + Dk;                             { f) }
 End;
 Until ( k > 1.0 );                        { g) }
 SetColor(GREEN);                          {Rysujemy siatke dla wykresu}
 X := 0;
 Y := 0;
 Repeat
 Begin
  MoveTo( x0 + X, y0 );
  LineTo( x0 + X, 30 );
  MoveTo( x0, y0 - Y );
  If ( Y < 301) Then LineTo( x0 + Skala_k, y0 - Y);
  X := X + 30;
  Y := Y + 30;
 End;
```


programowanie_strukturalne_DODATEK.indb 79 2010-07-05 11:54:48

```
   Until ( X > 360 );
   ReadKey;
   CloseGraph;
  End;
END.
```

Jak widać na rysunku D17, aby szanse udanego spotkania przekroczyły magiczną granicę 50%, należałoby dodać do rutynowych 15 minut jeszcze około 3 – 4 minuty.



Rysunek D17. Jeśli czekamy 18 – 19 minut, szanse spotkania rosną powyżej 50%. Na osi OX jedna kratka to 5 minut, na osi OY jedna kratka to 10%

4

Język C — kody powstające w wyniku asemblacji

Pętle programowe pisane w C powodują wygenerowanie stosunkowo krótkich sekwencji języka maszynowego (asemblera). Dwa przykłady znajdują się w tabeli D.4[1].

Tabela D.4. Relacja między kodem języka C a kodem asemblera

Asembler	Język C
`MOV _suma,0`	`suma = 0;`
`MOV _n,1`	`n = 1;`
`_2:`	`start:`
`MOV AX,_n`	`suma += n;`
`MOV BX,_suma`	`n++;`
`ADD AX,BX`	`if ( n > 10 ) goto koniec;`
`MOV _suma,AX`	`else goto start;`
`MOV AX,_n`	`koniec:`
`ADD _n,1`	

[1] Asemblacja została wykonana za pomocą narzędzia `SMALL_C`, stąd wywołanie `CALL __le` (wywołaj funkcję porównującą, realizującą `operator<=();` LE oznacza *less or equal* — mniejsze lub równe). Podobnie `CALL __gt` to wywołanie funkcji operatorowej GT (*greater than* — większe niż), czyli `operator>`. W zależności od użytego kompilatora C może tu pojawić się na przykład `CMP AX, 10` (porównaj AX z 10), a następnie skok warunkowy `J(ump)`, na przykład `JLE`.
Jak widać, pętle strukturalne (`for`, `while`, `do-while`) są podczas asemblacji i tak rozkładane na konstrukcje `if (...) goto ... else ...`. Etykiety, na przykład `_13:`, są generowane automatycznie.

programowanie_strukturalne_DODATEK.indb 81 2010-07-05 11:54:48

<table>
<tr><td><pre>MOV BX,_n
MOV AX,10
CALL __gt
OR AX,AX
JNE $+5
JMP _3
JMP _4
_3:
JMP _2
_5:
_4:</pre></td><td></td></tr>
<tr><td><pre>MOV _suma,0
MOV _n,1
_12:
MOV AX,_n
MOV BX,_suma
ADD AX,BX
MOV _suma,AX
MOV AX,_n
ADD _n,1
MOV BX,_n
MOV AX,10
CALL __le
OR AX,AX
JNE $+5
JMP _13
JMP _12
_13:</pre></td><td><pre>suma = 0; n = 1;
do
{
  suma += n;
  n++;
} while (n <= 10);</pre></td></tr>
</table>

programowanie_strukturalne_DODATEK.indb 82 2010-07-05 11:54:48

4.1. Kod asemblera generowany przez Borland C++

Porównanie dwóch sposobów asemblacji programu w języku C wydaje się w tym miejscu zdecydowanie celowe. Kod *SMALL_C* zachowuje zewnętrzne nazwy zmiennych, co znacznie ułatwia jego zrozumienie, zachowuje się jednakże dość dziwnie, wywołując w celu wykonania operacji porównania zewnętrzną (nieznaną tu) funkcję operatorową. Kod generowany przez Borland C, a przeznaczony dla asemblera *TASM* nie posługuje się funkcją operatorową, ale za to nie pokazuje zewnętrznych nazw zmiennych. W tabeli D.5 podano kod asemblera odpowiadający przykładowemu kodowi C.

Tabela D.5. Porównanie kodu C i kodu asemblera

Kod w języku C	Wygenerowany kod asemblera
`void main()`	`_main proc near`
`{`	`push bp`
`int n , suma;`	`mov bp,sp`
	`mov bx,1`
`for (n = 1, suma = 0; n <= 10;`	`xor cx,cx`
`suma = suma + n++);`	`jmp short @1@86`
	`@1@58:`
`suma = 0; n = 1;`	`mov ax,bx`
`while ( n <= 10 )`	`inc bx`
`{`	`mov dx,cx`
`suma += n;`	`add dx,ax`
`n++;`	`mov cx,dx`
`}`	`@1@86:`
	`cmp bx,10`
`suma = 0; n = 1;`	`jle short @1@58`
`do`	`xor cx,cx`
`{`	`mov bx,1`
`suma += n;`	`jmp short @1@170`
`n++;`	`@1@142:`
`} while ( n <= 10 );`	`add cx,bx`
`}`	`inc bx`
	`@1@170:`
	`cmp bx,10`

programowanie_strukturalne_DODATEK.indb 83 2010-07-05 11:54:48

```
  jle   short @1@142
  xor   cx,cx
  mov   bx,1
@1@226:
  add   cx,bx
  inc   bx
  cmp   bx,10
  jle   short @1@226
  pop   bp
  ret
_main  endp
```

Zamiast funkcji operatorowej `__le()` występuje tu instrukcja asemblera `JLE` (skocz, jeśli mniejsze lub równe), a porównanie wykonywane jest przez instrukcję `CMP` (*Compare* — porównaj). I tym razem jednak nawet bez dogłębnej analizy widać wyraźnie, że kod maszynowy wygenerowany na podstawie programu w języku C będzie krótki i szybki.

programowanie_strukturalne_DODATEK.indb 84 2010-07-05 11:54:48

5

Maszynowa reprezentacja liczb całkowitych

Liczba całkowita typu `short int` stanowi w C++ *16-bitowe słowo* i może zostać zastosowana na przykład w taki sposób jak na listingu poniżej.

Listing D14.C

```
#pragma inline
void main()
{
  char *napis = "\nRazem warzyw:  $";
  int marchewki = 2, pietruszki = 5;
  asm {
        MOV     DX, napis
        MOV     AH, 9
        INT     33
        MOV     DX, marchewki
        ADD     DX, pietruszki
        ADD     DX, '0'
        MOV     AH, 2
        INT     33
      }
}
```

UWAGA

Jeśli w stosowanej wersji kompilatora C niedostępny jest asembler inline ani asembler zewnętrzny, w Borland/Turbo C/C++ można posłużyć się *pseudozmiennymi rejestrowymi* (ang. *register pseudovariables*). Nazwy poprzedzone są znakiem podkreślenia i odpowiadają rejestrom procesora bazowego dla rodziny Intela, tzn. 8086. Są to odpowiednio: `_AX`, `_BX`, `_CX`, `_DX`, `_FLAGS` itd.

Poniżej zaprezentowano analogiczny program posługujący się zmiennymi pseudorejestrowymi i wykonujący te same operacje.

Listing D15.C

```
#include <stdio.h>
#include <conio.h>
main()
{
  int marchewki = 10, pietruszki = 31;
  clrscr();
  _DX = marchewki;
  _DX = _DX + pietruszki;
  printf("\n Rejestr DX po dodaniu 31: DX = %X ", _DX);
  getch();
  return 0;
}
```

Wydruk tego programu będzie zgodny z oczekiwaniem:

```
Rejestr DX po dodaniu 31: DX = 29
```

29 szesnastkowo to 41 dziesiętnie, więc wszystko jest w porządku, ale wystarczy dodać jedną instrukcję wydruku:

```
int marchewki = 10, pietruszki = 31;
clrscr();
_DX = marchewki;
printf("\n Rejestr DX po wpisaniu 10: DX = %X ", _DX);
_DX = _DX + pietruszki;
printf("\n Rejestr DX po dodaniu 31: DX = %X ", _DX);
```

— i już wydruk programu stanie się co najmniej zaskakujący:

```
Rejestr DX po wpisaniu 10: DX = A Rejestr DX po dodaniu 31: DX = 1F
```


programowanie_strukturalne_DODATEK.indb 86 2010-07-05 11:54:49

Dlaczego wynik nie jest zgodny z oczekiwaniami? Otóż problem polega na tym, że funkcje mogą zmieniać zawartość rejestrów procesora. Funkcja `printf()`, zanim zakończyła działanie, wyzerowała rejestr `DX`. Po dodaniu zmiennej `pietruszki` mamy tam `1F` szesnastkowo, czyli 31 dziesiętnie.

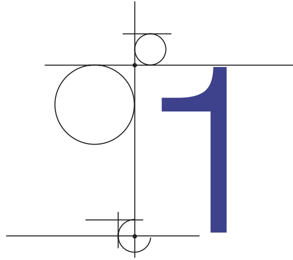
UWAGA

Komputer może się zepsuć i wtedy nie działa, ale jeśli działa, nie popełnia błędów. Jeśli wynik działania programu jest zaskakujący, to zawsze błąd popełnił programista.

programowanie_strukturalne_DODATEK.indb 88 2010-07-05 11:54:49

Część III

Ćwiczenia i zadania



Elementy Pascala

1.1. Ćwiczenia początkowe, środowisko tekstowe

Ćwiczenie 1.
Napisz i uruchom program wypisujący na ekranie kilka wierszy tekstu. Porównaj działanie procedur `Write()` i `WriteLn()`. Spróbuj za pomocą `Write()` wstawić odstępy pomiędzy słowa tekstu.

Ćwiczenie 2.
Wprowadź kilka błędów do swojego programu. Sprawdź, jakimi komunikatami reaguje kompilator na:

- brak średnika na końcu instrukcji,
- brak apostrofu przed napisem przeznaczonym do wyprowadzenia na monitor,
- pominięcie litery w słowie kluczowym (np. `Writen`).

Ćwiczenie 3.
Napisz program, który rozmieszcza znaki i teksty na środku ekranu (zamiast przy lewej krawędzi).

Ćwiczenie 4.
Posługując się `GotoXY()`, napisz program, który pisze tekst od tyłu (od prawej do lewej).

Ćwiczenie 5.
Spróbuj zastąpić pętlę typu `For` w wybranych przykładowych programach przez pętlę typu `Repeat...Until`.

Ćwiczenie 6.
Napisz program sumujący, dzielący i mnożący dwie wprowadzone przez użytkownika liczby całkowite.

Ćwiczenie 7.
Napisz program wczytujący imię, nazwisko użytkownika, a następnie sumujący te łańcuchy i wyprowadzający je jako jeden napis.

Ćwiczenie 8.
Stosując pętlę programową powtarzającą proste operacje arytmetyczne 1000, 100 000, 1 000 000 razy, porównaj szybkość operacji na liczbach o różnej długości, na przykład `Integer`, `Longint`, `Single`, `Real`, `Double`, `Extended`.

UWAGA

Licznik pętli nie może być typu `Integer`.

Ćwiczenie 9.
Narysuj na środku ekranu ramkę i umieść w jej wnętrzu dowolny napis.

Ćwiczenie 10.
Sprawdź, co się stanie, jeśli w małym oknie tekstowym typu `Window()` zechcesz umieścić zbyt długi tekst.

Ćwiczenie 11.
Napisz program generujący liczby losowe, który mógłby podpowiadać kombinacje liczb dla miłośników Dużego Lotka.

1.2. Ćwiczenia początkowe, środowisko graficzne

Ćwiczenie 1.
Narysuj domek, kopertę i bałwanka.

Ćwiczenie 2.
W rysunkach z poprzedniego ćwiczenia zastosuj wszystkie 16 kolorów z dostępnej palety.

Ćwiczenie 3.
Narysuj pęk prostych i tarczę strzelniczą (okręgi współśrodkowe).

Ćwiczenie 4.
Narysuj na ekranie kratkę:

a) zwykłą,

b) ukośną.

programowanie_strukturalne_DODATEK.indb 92 2010-07-05 11:54:49

Ćwiczenie 5.

Wypełnij dowolnym kolorem i wzorem cały ekran za wyjątkiem wnętrza wybranej figury.

Ćwiczenie 6.

Posługując się trzema technikami komputerowej animacji obrazów, napisz własny program pokazujący kółeczko odbijające się od krawędzi ekranu.

Ćwiczenie 7.

Sprawdź, ile różnych stron (`SetActivePage()`/`SetVisualPage()`) da się zrealizować na Twoim komputerze i jakie tryby graficzne są dostępne dla Turbo Pascala.

Ćwiczenie 8.

Sprawdź, ile bajtów zajmuje w pamięci wycinek obrazu (ekranu). Czy wielkość pamięci jest dokładnie proporcjonalna do pola powierzchni wycinka?

Ćwiczenie 9.

Wypełnij ekran okręgami i elipsami w przypadkowych kolorach i o przypadkowej wielkości. Po narysowaniu 1000 elips i okręgów utwórz na środku ekranu okienko graficzne z napisami.

Ćwiczenie 10.

Napisz program obrysowujący ramką nowo powstałe okno graficzne (`SetViewPort()`).

Ćwiczenie 11.

Sprawdź, jaką wysokość i szerokość w pikselach mają czcionki o rozmiarach 1, 2, 4. Zastosuj `TextHeight()` i `TextWidth()`. Czy czcionki domyślne (`default`) i potrójne (`triplex`) mają takie same wymiary w pikselach?

Ćwiczenie 12.

Rozszyfruj znaczenia i role funkcji i parametrów: `SetViewPort(1)`, `TextHeight('W')`, `GetMaxX()`, `GetMaxY()`, `ClipOn`.

1.3. Pytania powtórzeniowe

1. Wymień przykłady języków programowania niskiego i wysokiego poziomu. Wyjaśnij podobieństwa i różnice.
2. Czy komputer jest w stanie zrozumieć program napisany w języku Pascal? Co trzeba zrobić, by przetłumaczyć program napisany w języku Pascal na kod binarny zrozumiały dla komputera?
3. Po co wstawiane są komentarze do programów?
4. Jakie znaki rozpoczynają i kończą komentarze w językach, które już poznałeś:

   `'` (apostrof)

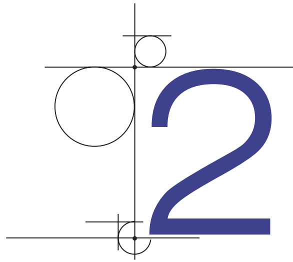
   `REM`

programowanie_strukturalne_DODATEK.indb 93 2010-07-05 11:54:49

```
//
{   }
/*  */
```

5. Czy program w Pascalu musi zaczynać się od słowa „`Program`”?
6. Do czego służy klauzula `Uses`?
7. Czy kompilator Pascala zwraca uwagę na komentarze zawarte w programie?
8. Czy `Begin`, `BEGIN` i `begin` to w Pascalu dokładnie to samo?
9. Jakie rodzaje plików tworzy kompilator Pascala?
10. Jakie rozszerzenie powinny mieć pliki źródłowe z naszymi programami w Pascalu?

programowanie_strukturalne_DODATEK.indb 94 2010-07-05 11:54:49



Elementy C i C++

2.1. Podstawy programowania w C i C++, pytania powtórzeniowe

1. Czy kompilator C „zwraca uwagę” na komentarze zawarte w programie?
2. Jakie rodzaje plików tworzy kompilator C?
3. Czy funkcja `exit()` zwraca wartość? A instrukcja `return`?
4. Co to jest plik nagłówkowy?
5. Czym różni się stała od zmiennej?
6. Czy w C/C++ stosuje się klauzulę `Var`, jak w Pascalu?
7. Instrukcje złożone (bloki instrukcji) w Pascalu tworzy się za pomocą słów `Begin-End`. W językach C/C++ nie ma tych słów. Czy mimo to można utworzyć blok instrukcji? Jeśli tak — w jaki sposób? Wybierz właściwą odpowiedź:
   a. Skoro nie ma `Begin` i `End`, to nie da się tworzyć instrukcji złożonych.
   b. Instrukcje złożone są w C/C++ niepotrzebne, ponieważ można tworzyć funkcje.
   c. Instrukcje złożone są ujęte w parę nawiasów `{...}`.
   d. Instrukcje złożone są tylko w C++, w C ich nie ma.
8. Czy `2 + 5 * 2` da taki sam wynik, jak `(2 + 5) * 2`?
9. Czy wyniki: `7 % 2` oraz `4 % 3` będą takie same?
10. Na czym polega podstawowa różnica między danymi typu `int` a danymi typu `float`?
11. Co to za znaki: `%c`, `%d`, `%f`? Która z poniższych odpowiedzi jest poprawna, a która nie?
    a. Są to specyfikatory formatu.
    b. `%c` służy do uzyskania formatu znakowego (`char`).
    c. `%d` służy do uzyskania formatu liczby całkowitej (`int`).

d. `%f` służy do uzyskania formatu liczby rzeczywistej (`float`).

e. `%d` można zastąpić poprzez `%i`.

12. Znaki są przechowywane w komputerze jako ciągi bitów. Ile bitów ma zmienna typu `char`?

a. 4

b. 8

c. 16

d. zależy od komputera i systemu operacyjnego

13. Jak zapisać następujące liczby w postaci wykładniczej:

2500 0.0025 25.0 –0.0025

14. Sprawdź, posługując się kompilatorem C/C++, czy zapisy: `#include <stdio.h>` oraz `#include "stdio.h"` są równoważne.

15. Co to jest `stdin`, `stdout`, `stderr`?

16. Liczba 32 szesnastkowo — ile to dziesiętnie?

a. 32

b. $32_{16} = 3 * 16^1 + 2 * 16^0 = 50_{10}$

c. 320

17. Czy wyprowadzane dane można wyrównać do lewej w polu wyjściowym zamiast do prawej?

18. Czym różnią się funkcje `putc()`, `puts()` i `putchar()`?

19. Która część w specyfikatorze `%10.3f` stanowi specyfikator minimalnej szerokości pola wyjściowego, a która jest specyfikatorem precyzji?

20. Dlaczego dołączamy do programów pliki nagłówkowe *<stdio.h>*, *<conio.h>*, *<stdlib.h>*?

21. Czy zapisy poniżej są równoważne?

```
z = z * x + y;     z = z *( x + y);     oraz     Z *= x + y;
```

22. Co oznacza zapis: `z %= 33`?

23. Na czym polega różnica między preinkrementacją (`++x`) a postinkrementacją (`x++`)?

24. Jaki będzie wynik działania poniższego kodu?

```
int x = 7, y = 5;
printf("Operacja: x / y daje wynik: %d\n", x / y);
printf("Operacja: (float)x / y daje wynik: %f\n", (float)x / y);
```

25. Jak nazywa się technika wymuszonej konwersji formatu użyta w powyższym przykładzie?

26. Czy minus jednoargumentowy i minus dwuargumentowy działają tak samo?

27. Które z operatorów — operatory relacji czy operatory arytmetyczne — mają wyższy priorytet? Co wykona się najpierw: działanie arytmetyczne czy porównanie?

28. Co zwraca wyrażenie relacji?

29. Pascal operuje typem `Boolean` o wartościach `True`/`False`. A jak w języku C/C++ są rozumiane wartości logiczne `True`/`False`?

30. Czym różnią się operatory = oraz ==?

31. Który z operatorów C/C++ odpowiada operatorowi := Pascala?

32. Dane są zmienne: `int x=15, y=4;`. Co zwrócą wyrażenia: `x / y` oraz `(float) x / y`?

33. Czy wyrażenia `y *= x + 5` oraz `y = y * x + 5` są równoważne?

34. Dane są zmienne: `int x=3, y=6;`. Jaka jest wartość zmiennej `z` po wykonaniu następującego wyrażenia: `z =( x * y == 18)`?

2.2. Instrukcje sterujące — pytania

1. Jak zadziała poniższa pętla? Czy to jest pętla nieskończona?

```
while (c != 'x')
{
  c = getc(stdin);
  putchar(c);
}
```

2. Czy `while` może występować na końcu pętli programowej? Czy poniższy kod jest poprawny?

```
i = 65;
do
{
  printf("Wartosc numeryczna znaku: %c wynosi: %d.\n", i, i);
  i++;
} while (i<72);
```

3. Poniższy kod zawiera typowy błąd. Na czym on polega? Jak zadziała program?

```
int suma, i;
for (i=0; i<8; i++);
  suma+=i;
```

4. Czy poniższy nagłówek sterujący pętli jest formalnie poprawny? Jak zadziała taka pętla?

```
for (i=0, j=10; i<10, j>0; i++, j--) { ....  }
```

5. Czy wolno stosować pętlę `for` tak:

```
for( ; ; )
{
  / * blok instrukcji */
}
```

Czy będzie to pętla nieskończona?

6. Czym różnią się pętle `while` i `do-while`?

7. Jeśli jedna pętla jest zagnieżdżona w drugiej, która z nich musi zakończyć się wcześniej — wewnętrzna czy zewnętrzna?

8. Czy dwie przykładowe pętle poniżej wykonają się tyle samo razy?

```
for(j=0; j<8; j++);
for(j=0; j<=8; j++);
```

9. Czy pętla `for`:

```
for(j=65; j<72; j++)
  printf("%c", j);
```

— jest równoważna następującej pętli typu `while`:

```
int k = 65;
while (k<72)
{
  printf("%c", k);
  k++;
}
```

10. Czy taka pętla jak poniżej wydrukuje cokolwiek?

```
/* a */
int k = 100;
while (k<100)
{
  printf("%c", k);
  k++;
}

/* b */
int k = 100;
do
{
```

```
  printf("%c", k);
  k++;
} while (k<100);
```

11. Na czym polega różnica między dwoma fragmentami kodu:

```
for(i=0, j=1; i<8; i++, j++)
  printf("%d + %d = %d\n", i, j, i+j);

// oraz

for(i=0, j=1; i<8; i++, j++);
  printf("%d + %d = %d\n", i, j, i+j);
```

12. Czy dopuszczalne jest takie zastosowanie operatora:

```
int size;
size = sizeof(int);
size = sizeof(666);
size = sizeof(123.456);
```

13. Uzupełnij tabelę wartości zwracanych przez operator `AND (&&)`.

Wyrażenie 1	Wyrażenie 2	Wartość zwrócona przez operator `&&`
1		1
1	0	
	1	0
0	0	

14. Jaki będzie wydruk poniższego programu?

```
int num = 0;
printf("Operator && zwraca: %d\n", (num%2 == 0) && (num%3 == 0));
num = 2;
printf("Operator && zwraca: %d\n", (num%2 == 0) && (num%3 == 0));
num = 3;
printf("Operator && zwraca: %d\n", (num%2 == 0) && (num%3 == 0));
num = 6;
printf("Operator && zwraca: %d\n", (num%2 == 0) && (num%3 == 0));
```


programowanie_strukturalne_DODATEK.indb 99 2010-07-05 11:54:50

15. Uzupełnij tabelę wartości zwracanych przez operator `OR` (`||`).

Wyrażenie 1	Wyrażenie 2	Wartość zwrócona przez operator `||`
1		1
	0	
0		
0	0	0

16. Jak zadziała poniższy program?

```
printf("Wpisz jednocyfrowa liczbe podzielna\ni przez 2, i przez 3.\n");
     for (num = 1; (num%2 != 0) || (num%3 != 0); )
        num = getchar() - 48;
```

17. Uzupełnij tabelę wartości zwracanych przez operator negacji `!`.

Wyrażenie	Wartość zwracana przez operator `!`
	1
1	

18. Jaki będzie wydruk poniższego programu?

```
num = 7;
printf("Dana jest zmienna num o wartości 7\n");
printf("!(num < 7) zwraca: %d\n", !(num < 7));
printf("!(num > 7) zwraca: %d\n", !(num > 7));
```

19. W ostatniej kolumnie tabeli wpisz brakujące oznaczenia. Do wyboru są: `XOR`, `OR`, `NEG`, `SHL`, `SHR`.

Operator	Opis	Oznaczenie
&	Bitowy iloczyn logiczny bit po bicie	`AND / I`
|	Bitowa suma logiczna	
^	Bitowa alternatywa wyłączna	
~	Bitowe uzupełnienie do 2 (komplementarne)	
>>	Przesunięcie bitów w prawo	
<<	Przesunięcie bitów w lewo	

20. Jakie wyniki zwróci poniższy program?

```
int x = 12345, y = 23456;
printf("Dana jest zmienna x = %u, czyli 0x%04X\n", x, x);
printf("oraz zmienna y = %u, czyli 0x%04X\n", y, y);
```

```
z = x & y;
printf("x & y zwraca: %6u, czyli 0x%04X\n", z, z);
z = x | y;
printf("x | y zwraca: %6u, czyli 0x%04X\n", z, z);
z = x ^ y;
printf("x ^ y zwraca: %6u, czyli 0x%04X\n", z, z);
printf("   ~x zwraca: %6u, czyli 0x%04X\n", ~x, ~x);
```

21. Czy poniższe stwierdzenia, jakoby w C/C++ istniał operator warunkowy, który ma aż 3 argumenty, są prawdziwe?

Uogólniony format operatora warunkowego jest następujący:

`x ? y : z` gdzie `x`, `y` i `z` to trzy operandy operatora warunkowego. Role tych operandów są następujące:

- `x` — wyrażenie logiczne, którego wartość logiczna jest sprawdzana (wyrażenie warunkowe),
- `y` — wyrażenie wynikowe, wykonywane, jeśli x zwraca wartość niezerową (jest traktowany jako logiczna prawda, inaczej: warunek `x` spełniony),
- `z` — wyrażenie wynikowe, wykonywane, jeśli `x` zwraca wartość zerową (jest traktowany, jako logiczny fałsz, inaczej: warunek `x` nie został spełniony).

Rezultatem jest więc `y`, jeśli warunek `x` zwraca wartość `TRUE`, bądź `z` — w przeciwnym wypadku. Dla przykładu wyrażenie: `x > 0 ? 'T' : 'N'` zwróci wartość `'T'`, jeśli bieżąca wartość zmiennej `x` jest rzeczywiście większa od zera. W przeciwnym razie wyrażenie zwróci wartość `'N'`.

22. Jak zadziała poniższy program?

```
x = sizeof(int);
printf("%s\n", (x == 2) ? "Typ danych int ma 2 bajty." :
  "Typ int nie ma 2 bajtow");
printf("Maksymalna wartosc typu int to: %d\n",
  (x != 2) ? ~(1 << x * 8 - 1) : ~(1 << 15));
```

23. Na czym polega różnica między operatorami `|` oraz `||`?

24. Dlaczego `1 << 3` jest równoważne `1 * 2`^{3}?

25. Jakie wartości zwrócą następujące wyrażenia: `(x=1)&&(y=10)` oraz `(x=1)&(y=10)`?

26. Dane są zmienne: `x` = 96, `y` = 1, `z` = 69. Jaką wartość zwróci wyrażenie: `!y ? x==z : y`?

27. Jaką wartość zwrócą wyrażenia: `~0011000000111001` oraz `~1100111111000110`?

28. Dane są `x` = 0xEFFF oraz `y` = 0x1000. Jakie wartości zwrócą: `~x` oraz `~y` w formacie szesnastkowym?

programowanie_strukturalne_DODATEK.indb 101 2010-07-05 11:54:50

29. Dane `x` = 15. Jaką wartość zwrócą wyrażenia: `(x%2==0)||(x%3==0)` oraz `(x%2==0)&&(x%3==0)`?

30. Czy `8>>3` jest równoważne `8 / 2`^{3}. A co oznacza `1<<3`?

31. Który bit może być używany jako bit znaku w liczbach całkowitych?

32. Kiedy i do czego stosuje się słowa kluczowe `short` oraz `long`?

33. Czy funkcja `sin()` oczekuje argumentu w stopniach, czy w radianach?

34. Czy w poniższym przykładzie `x` i `y` będą sobie równe?

```
int x = 0x8765;
unsigned int y = 0x8765;
```

35. Jeśli dane całkowite mają format 16-bitowy, jaki jest format szesnastkowy liczby dziesiętnej –12 345?

36. Który specyfikator: `%ld` czy `%lu` powinien być zastosowany, by zadać format zmiennej typu `unsigned long int`?

37. Jak nazywa się plik nagłówkowy, który trzeba dołączyć, aby korzystać z funkcji matematycznych?

38. Jak działa poniższa instrukcja?

```
if (x>0) printf("Pierwiastek z x wynosi: %f\n", sqrt(x));
```

39. Jak zadziała poniższy kod?

```
for (i=0; i<=100; i++) if ((i%2 == 0) && (i%3 == 0)) printf("   %d\n", i);
```

40. Które liczby zostaną dodane, a które pominięte? Jaka będzie suma?

```
sum = 0;
for (i=1; i<8; i++)
{
  if ((i==3) || (i==5))
    continue;
  sum += i;
}
```

41. Dlaczego zazwyczaj trzeba do instrukcji `switch-case` dodawać instrukcję `break`?

42. Dane jest `x` = 0. Czy poniższe operacje arytmetyczne zostaną wykonane?

```
if (x != 0) y = 123 / x + 456;
```

43. Dane: `operator` = '-' oraz `x` = 4, `y` = 2. Jaka będzie końcowa wartość zmiennej `x` po wykonaniu instrukcji `switch`:

```
switch(operator)
{
  case '+':  x += y;
  case '-':  x -= y;
```


programowanie_strukturalne_DODATEK.indb 102 2010-07-05 11:54:50

```
  case '*':  x *= y;
  case '/':  x /= y;
  default : break;
}
```

44. Jaka będzie wartość całkowitej zmiennej numerycznej `x` po wykonaniu poniższego kodu?

```
int i, x;
x = 1;
for(i=2; i<10; i++)
{
  if (i%3==0) continue;
  x += i;
}
```

45. Które spośród deklaracji tworzą zwykłe zmienne, a które stanowią deklaracje wskaźników? Czym różnią się te deklaracje?

```
char c, *d;
int x, *p;
float y, *z;
```

46. Które spośród instrukcji inicjują zwykłe zmienne, a które stanowią zainicjowanie wskaźników? Czym różnią się te instrukcje?

```
c = 'A';  d = &c;
x = 1234; p = &x;
```

47. Co oznacza pojęcie kroku wskaźnika (skoku wskaźnika)?

48. Jaki sens mają następujące operacje:

```
int x, y, *p1, *p2;
p1 = &x;  p2 = &y;
x = p2 - p1;
```

Jaką wartość zawiera teraz zmienna `x`?

49. Poniżej zainicjowane zostały dwa wskaźniki. Co wskazują po takim zainicjowaniu? Do czego można zastosować *NULL-pointer*?

```
char *ptr_c;
int *ptr_x;
ptr_c = ptr_x = 0;
```

50. Czy wskaźnik pusty może wskazywać poprawne dane?

51. Czy w wyrażeniach przytoczonych poniżej symbol `*` oznacza operator dereferencji, czy operator mnożenia?

programowanie_strukturalne_DODATEK.indb 103 2010-07-05 11:54:50

```
* ptr
x * y
y *= x + 5
*y  *= *x + 5
```

52. Co zwrócą wyrażenia: `ptr_int` oraz `*ptr_int`, jeśli adres zmiennej `x` wynosi `0x1F38` i dane jest:

```
int x = 10;
int *ptr_int;
ptr_int = &x;
```

53. Ile i jakich zmiennych zostanie utworzonych w pamięci po takiej deklaracji?

```
int array_int[2][3] ={ { 1, 2, 3 }, { 4, 5, 6 } };
```

54. W Pascalu stosuje się słowo `Array`. Czy dopisanie w powyższym przykładzie `array_int` nie stanowi błędu formalnego?

55. W tablicach wielowymiarowych indeksy wzdłuż każdego wymiaru rozpoczynają się od zera: `Tab2[0][0];` `Tab3[0][0][0];` czy od jedynki: `Tab2D[1][1];` `Tab3D[1][1][1];`?

56. Czy można zadeklarować tablicę trójwymiarową tak jak poniżej?

```
int 3D_Tab[0][0][0];
```

57. Jak można odwoływać się do elementu tablicy, posługując się wskaźnikiem? Czy poniższe odwołania są poprawne? Czy obydwa zadziałają identycznie?

```
char Tabl[2][5], *wskaznik1;
wskaznik1 = Tabl;
wskaznik1 = Tabl[0][0];
```

58. Do czego i w jakich tablicach może się przydać znak zerowy `'\0'`?

59. Czy poniższe zainicjowanie tablicy jest poprawne? Jeśli nie, to na czym polega błąd?

```
char lista[][] = {
'A', 'a',
'B', 'b',
'C', 'c',
'D', 'd'
};
```

60. Iluwymiarowe są poniższe tablice?

```
char array1[3][19];
float array3[][8][16];
char array4[][80];
```


programowanie_strukturalne_DODATEK.indb 104 2010-07-05 11:54:50

61. Na czym polega błąd w zainicjowaniu elementów poniższej tablicy?

```
int dane[3];
dane[1] = 1;
dane[2] = 2;
dane[3] = 3;
```

62. Czy zainicjowanie tablicy znakowej za pomocą łańcucha znaków odpowiada instrukcji (1), czy instrukcji (2)? Uzasadnij odpowiedź.

```
char array_ch[] = "Hello!";
char arr_str[6] ={ 'H', 'e', 'l', 'l', 'o', '!' };           // (1)
char array_ch[7] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };  // (2)
```

63. Czym różnią się poniższe dwie deklaracje?

```
char znak = 'x';
char string = "x";
```

64. Do czego służy i jak działa funkcja `strlen()`?

65. Jak działa poniższa instrukcja? Jak należy zadeklarować zmienne `x` i `y`?

```
scanf("%d %d", &x, &y);
```

66. Jakie typy danych wczytuje funkcja `scanf()`?

67. Z ilu typowych części składa się zazwyczaj funkcja w języku C?

a) Funkcja składa się normalnie z 6 części: typu funkcji, nazwy funkcji, argumentów funkcji, nawiasu otwierającego, ciała funkcji, nawiasu zamykającego.

b) Liczba tych części zmienia się zależnie od rodzaju funkcji.

c) Niezbędna jest tylko nazwa funkcji, reszta jest opcjonalna.

2.3. Programowanie strukturalne — zadania

1. Stosując funkcję `cprintf()`, wydrukuj swoje nazwisko w 16 kolorach.

2. Jakie komunikaty o błędach (*Error*) lub ostrzeżenia (*Warning*) pojawią się, jeśli spróbujesz skompilować poniższy program?

```
#include <stdlib.h>
#include <stdio.h>
main()
{
  printf("To program z Zadania 2.\n");
  exit("0");
}
```


programowanie_strukturalne_DODATEK.indb 105 2010-07-05 11:54:50

3. Jakie komunikaty o błędach pojawią się, jeśli spróbujesz skompilować poniższy program?

```
void main()
{
  printf("To program z Zadania 3.\n");
  return 0;
}
```

4. Dane są dwie instrukcje: `x = 3;` oraz `y = 5 + x;`. Jak zbudować z nich jeden blok instrukcji?

5. Sprawdź, czy dzielenie całkowite `134 / 100` oraz `17 / 10` da taki sam wynik?

6. Sprawdź, czy wynik dodawania `3000 + 1.0` będzie liczbą zmiennoprzecinkową. A wynik dzielenia `3000 / 1.0`? Ile bajtów zajmuje ten wynik?

7. Napisz program drukujący na ekranie wartości numeryczne kodów ASCII liter `Z`, `A` oraz `z`, `a`.

8. Dane są dwie wartości numeryczne: 82 i 104. Napisz program drukujący na ekranie znaki o kodach ASCII 82 i 104.

9. Czy zmiennej typu `int`, która jest reprezentowana w komputerze przez 16 bitów, można przypisać wartość całkowitą 32 768?

10. Dana jest deklaracja: `double x = 123.456;`. Napisz program drukujący wartość zmiennej `x` w formacie zmiennoprzecinkowym i w formacie wykładniczym.

11. Napisz program wyprowadzający na ekran numeryczną wartość kodu znaku specjalnego: `\t`. Najpierw przypisz `'\t'` do zmiennej znakowej `char zmienna = '\t';`.

12. Jakie wyniki wydrukuje poniższy kod?

```
printf("%X                  %x              %d\n", 13, 13, 13);
printf("%X                  %x              %i\n", 14, 14, 14);
printf("%X                  %o              %d\n", 15, 15, 15);
```

13. Jaki będzie wydruk poniższego programu? Czy potrafisz odpowiedzieć bez uruchamiania i wykonania tego kodu?

```
#include <stdio.h>
void main( void )
{
  int num1, num2, num3, num4, num5;
  num1 = 1; num2 = 12;
  num3 = 123; num4 = 1234; num5 = 12345;
  printf("%8d %-8d\n", num1, num1);
  printf("%8d %-8d\n", num2, num2);
```


programowanie_strukturalne_DODATEK.indb 106 2010-07-05 11:54:50

```
    printf("%8d %-8d\n", num3, num3);
    printf("%8d %-8d\n", num4, num4);
    printf("%8d %-8d\n", num5, num5);
}
```

14. Napisz program wyprowadzający na ekran trzy znaki: `'H'`, `'e'`, `'j'` — razem, jako jeden wyraz.

15. Wydrukuj na ekranie dwie liczby: 123 i 123.456, wyrównując je do lewej w polu wyjściowym.

16. Wydrukuj na ekranie trzy liczby dziesiętne: 15, 150, 1500 w formacie szesnastkowym.

17. Napisz program wykonujący tzw. echo, czyli wczytujący znak od użytkownika za pomocą funkcji `getch()` i powtarzający go na ekranie przy pomocy funkcji `putch()`.

18. Czy (i ewentualnie jakie) komunikaty o błędach lub ostrzeżenia pojawią się przy kompilacji i uruchomieniu poniższego programu?

```
void main()
{
    int c;
    c = getchar();
    putchar(c);
}
```

19. Rozbuduj program z zadania 18. tak, by pracował na czystym ekranie i czekał na naciśnięcie klawisza przez użytkownika.

20. Stosując funkcję `cprintf()`, dodaj do poprzedniego programu (z zadania 19.) informację dla użytkownika, co program robi.

21. Dane są zmienne: `int x` = 1, `y` = 3. Napisz program drukujący na ekranie rezultaty następujących wyrażeń:

```
x += y,   x += -y,   x -= y,   x -= -y,   x *= y,   x %= y   oraz   x *= -y .
```

22. Napisz program, który po zainicjowaniu zmiennej `x` poprzez przypisanie jej wartości 1 wyprowadza na ekran rezultaty dwóch następujących instrukcji:

```
printf("Inkrementacja x++ daje wynik: %d\n", x++);
printf("Zmienna x zawiera: %d\n", x);
```

— i czeka na naciśnięcie klawisza.

23. Czy zmiana zapisu zmieni wynik z poprzedniego zadania?

```
printf("x=x++ daje wynik: %d\n", x=x++);
printf("Zmienna x zawiera wartosc: %d\n", x);
```


programowanie_strukturalne_DODATEK.indb 107 2010-07-05 11:54:50

24. Ten program ma porównywać wartości dwóch zmiennych: `x` i `y`, sprawdzając, czy zmienne te są sobie równe. Na czym polega błąd w programie?

```
#include <stdio.h>
void main()
{
  int x, y;
  x = y = 0;
  printf("Wynik porownania zmiennych: %d\n", x = y);
}
```

25. Dane: `x` = 123 oraz `y` = 3. Napisz program drukujący na ekranie wartości wyrażeń: `x>>y` oraz `x<<y`.

26. Napisz program drukujący na ekranie wartości wyrażeń: `0xFFFF ^ 0x8888` oraz `0xABCD & 0x4567`.

27. Napisz program i sprawdź, jaki wynik da wyrażenie: `0xDCBA | 0x1234`.

28. Posługując się operatorem warunkowym `?:` i instrukcją `for`, napisz program, który wczytuje znaki wprowadzane przez użytkownika, aż do momentu wprowadzenia znaku `q`. W drugim polu instrukcji `for` zastosuj wyrażenie `x!='q' ? 1 : 0`.

29. Zastosuj w programie następujące modyfikatory/specyfikatory formatu:

`%hd`, `%hi`	dziesiętnie, całkowita, krótka:	`short int`
`%hu`	jw., ale bez znaku:	`short unsigned int`
`%ld`, `%Ld`	dziesiętnie, całkowita, długa:	`long int`
`%lu`, `%Lu`	jw., ale bez znaku	`long unsigned int`

30. Skonstruuj program demonstrujący działanie poniższego przykładowego kodu. Jaki wydruk będzie rezultatem jego wykonania?

```
x = 45.0;                        /* 45 stopni */
x *= 3.141593 / 180.0;           /* zamiana stopni na radiany */
printf("Sinus 45 st:   %f.\n", sin(x));
printf("Cosinus 45 st: %f.\n", cos(x));
printf("Tangens 45 st: %f.\n", tan(x));
```

31. Dany jest kąt 30 stopni. Napisz program obliczający sinus i tangens tego kąta.

32. Napisz program obliczający nieujemny pierwiastek kwadratowy z liczby `0x19A1`. Wskazówka: zastosuj funkcję `sqrt()` lub `pow()`.

33. Napisz program wczytujący znaki z klawiatury. Jeśli będą to znaki `A`, `B`, `C`, `D`, wyprowadź na ekran ich kody numeryczne. Wskazówka: wymagane jest zastosowanie instrukcji `switch`.

programowanie_strukturalne_DODATEK.indb 108 2010-07-05 11:54:51

34. Napisz program wczytujący znaki ze standardowego wejścia, aż do wprowadzenia znaku `'q'` lub `'Q'`.

35. Skonstruuj program, który odczytuje adresy zmiennych, wykorzystując poniższy kod. Przenieś plik wykonywalny EXE na inny komputer i sprawdź, czy adresy zmiennych pozostają takie same. Oto fragment kodu do zastosowania:

```
char c;
int x;
float y;
printf("c: adres=0x%p, zawartosc=%c\n", &c, c);
printf("x: adres=0x%p, zawartosc=%d\n", &x, x);
printf("y: adres=0x%p, zawartosc=%5.2f\n", &y, y);
```

UWAGA

Specyfikator formatu `%p` stosowany w funkcji `printf()` jest objęty standardem ANSI C. Jeśli kompilator (inny niż Borland C/C++) nie obsługuje specyfikatora `%p`, możesz zastosować w funkcji `printf()` specyfikator `%u` lub `%lu` w celu uzyskania i prawidłowego wydruku adresów zmiennych.

36. Dane są zmienne: wskazywana `x` oraz wskazująca `ptr`. Jaką wartość będzie zawierać zmienna `x` po wykonaniu poniższego przypisania?

```
int x = 123;
int *ptr;
ptr = &x;
*ptr = 456;
```

37. Dane są zmienne: `int x` = 5, `y` = 6. Napisz program, który obliczy wartość iloczynu tych zmiennych i zapisze wynik końcowy pod adresem zajmowanym przez zmienną `x`. Użyj wskaźników do wykonania wszystkich tych operacji.

38. Za pomocą operatora `sizeof()` sprawdź, jaki jest rozmiar wskaźników na Twoim komputerze. Zastosuj modyfikatory `far`, `near` i powtórz sprawdzenie.

```
int near *ptr1;
int far *ptr2;
```

39. Dane są trzy zmienne całkowite: `int x` = 128, `y` = 1024, `z` = 2048. Napisz program drukujący na ekranie ich adresy i zawartość.

40. Napisz program, który zmiennej zmiennoprzecinkowej równej początkowo: `double flt_num = 123.456`, za pomocą wskaźnika do danych typu `double` przypisuje nową wartość: 6543.210.

41. Dana jest zmienna znakowa: `char ch = 'A'`. Napisz program zmieniający zawartość tej zmiennej na 67_{10}, używając wskaźnika do zmiennej `char *p`.

programowanie_strukturalne_DODATEK.indb 109 2010-07-05 11:54:51

42. Napisz program inicjujący tablicę (wektor), a następnie drukujący całą jej zawartość na ekranie. Zastosuj konstrukcję z pętlą `for`, jak poniżej.

```
int i;
int T[10];
for (i=0; i<10; i++)
{
  T[i] = i + 1;
  printf("Element tablicy T[%d] ma wartosc: %d.\n", i, T[i]);
}
```

43. Wydrukuj tę samą zawartość tablicy, posługując się wskaźnikiem.

44. Zastosuj w programie poniższy fragment kodu i sprawdź, czy obydwa adresy i obydwie wartości będą identyczne, czy też różne.

```
int i, lista[10], *ptr;
ptr = lista;
printf( "Adres startowy: 0x%p\n", ptr);
ptr = &lista[0];
printf( "Adres pierwszego elementu: 0x%p\n", ptr);
```

45. Dana jest tablica dwuwymiarowa:

```
char list_ch[][2] = {
 '1', 'a',
 '2', 'b',
 '3', 'c',
 '4', 'd',
 '5', 'e' };
```

Napisz program mierzący całkowity obszar pamięci zajmowany przez tę tablicę i wydrukuj wszystkie elementy na ekranie.

46. Dana jest następująca tablica:

```
double data[5] = { 1.12345, 2.12345, 3.12345, 4.12345, 5.12345  };
```

Zmierz ilość pamięci zajmowanej przez tę tablicę, wydrukuj wyniki na ekranie.

47. Co w następującej funkcji zapisane jest nieprawidłowo?

```
int 3integer_add(int x, int y, int z)
{
  int wynik;
  wynik = x + y + z;
  return wynik;
}
```


programowanie_strukturalne_DODATEK.indb 110 2010-07-05 11:54:51

48. Co w następującej funkcji zapisane jest nieprawidłowo?

```
int integer_add(int x, int y, int z)
{
  int wynik;
  wynik = x + y + z   return wynik;
}
```

49. Napisz w języku C funkcję, która będzie mnożyć dwie liczby całkowite i zwracać wynik.

50. Jaka jest wartość całkowitej zmiennej numerycznej `int x` po wykonaniu poniższego kodu?

```
x = 1;
for (i=2; i<10; i++)
{
  if (i%3==0) continue;
  x += i;
}
```

51. Napisz program wczytujący znaki ze standardowego wejścia (`stdin`). Jeśli będą to znaki A, B, C, wyprowadź na ekran ich kody numeryczne. Wymagane jest zastosowanie instrukcji `switch`.

52. Napisz program wczytujący znaki ze standardowego wejścia, aż do wprowadzenia znaku `'q'`.

53. Zastąp w przykładach wszystkie instrukcje `switch` drabinkami `if-else-if`.

54. W poniższym przykładzie wyeliminuj instrukcję `goto`. Czy można się bez niej obejść?

```
int x = 1;
...
etykieta:
x++;
if (x > 5) goto koniec;
goto etykieta;
koniec:
```

55. Zapisz kod z zadania 54:

a) posługując się tylko pętlą `for`,

b) posługując się tylko słowami kluczowymi `do-while`,

c) stosując `break` lub `continue`.

programowanie_strukturalne_DODATEK.indb 111 2010-07-05 11:54:51

programowanie_strukturalne_DODATEK.indb 112 2010-07-05 11:54:51

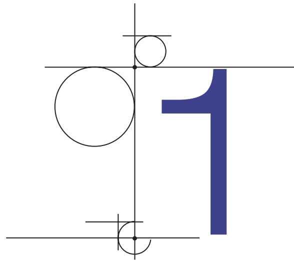
Część IV

Delphi

Delphi to w zasadzie obiektowy Pascal, ale trochę rozbudowany. Projektując język Delphi, Borland wprowadził do Pascala pewne rozszerzenia, wynikające z graficznej orientacji i podyktowane dążeniem do uproszczenia działania automatycznych kreatorów kodu. W tej części omówiono krótko najważniejsze modyfikacje, których zrozumienie jest ważne, byśmy poprawnie pojmowali niektóre automatycznie wygenerowane fragmenty kodu i byśmy mogli bezkonfliktowo rozbudowywać kody własnych aplikacji.

Oprócz typowej dla Delphi notacji z kropką, konstruowanie wizualnych aplikacji w Delphi nie wymaga (do pewnego momentu) dogłębnej znajomości paradygmatu OOP, czyli zasad programowania obiektowego. Obydwie istotne metodyki, tj. zasady programowania zdarzeniowego i reguły programowania obiektowego, zostały omówione w sposób systematyczny w podręczniku, w części poświęconej C++.

programowanie_strukturalne_DODATEK.indb 113 2010-07-05 11:54:51



Rozszerzenia i specyfika języka Object Pascal w Delphi

Choć w zasadzie stale pozostajemy w kręgu języka Pascal, programy tworzone w Delphi różnią się od programów napisanych w Pascalu zarówno samą filozofią środowiska operacyjnego, jak i sposobem organizacji kodu. Filozofia środowiska DOS polegała na tym, że program napisany w Pascalu w typowej sytuacji był jedyną pracującą aplikacją i kontaktował się ze środowiskiem operacyjnym najczęściej w trybie tekstowym. Wykorzystywał także przerwania systemowe DOS. Aplikacje Delphi mają swoje graficzne okno, ale w Windows najczęściej nie są jedynymi działającymi programami. Z systemem operacyjnym kontaktują się poprzez komunikaty o zdarzeniach (a nie poprzez przerwania), a środowisko jest graficzne. Z tych kilku dość istotnych powodów w samej koncepcji programowania musiał pojawić się inny sposób myślenia. Etapy konstruowania aplikacji w Delphi prześledzimy w dalszych rozdziałach tej części. Zacznijmy od rozszerzeń języka Pascal wprowadzonych w Object Pascalu i w Delphi.

1.1. Tablice otwarte

W Borland Pascal 7 wprowadzono koncepcję tzw. **tablic otwartych** (ang. *open arrays*), które w Delphi stosowane są dość często. Pozwalają one na tworzenie uniwersalnych funkcji i procedur, które manipulują statycznymi tablicami o różnej wielkości. Przykładowo, po zadeklarowaniu funkcji w sposób następujący:

```
Function CalcAverage( X : Array of Real ) : Real;     { oblicz srednia }
```

— argumentem `X` tej funkcji będzie tablica typu otwartego (rozmiar tablicy jest nieokreślony) o typie bazowym `Real`. Poniższy fragment kodu ilustruje konstrukcję i użycie takiej funkcji:

```
Type
 Array1 = Array [1..10] Of Real;   { dwie tablice o roznej wielkosci }
 Array2 = Array [1..20] Of Real;
Var
 X1 : Array1;
 X2 : Array2;
Function CalcAverage( X : Array of Real ) : Real;
Begin
 { Instrukcje definiujace cialo funkcji. }
End;

BEGIN
 GetData(X1);
 GetData(X2);
 WriteLn( 'Srednia z tablicy X1 = ', CalcAverage( X1 ));
 WriteLn( 'Srednia z tablicy X2 = ', CalcAverage( X2 ));
END.
```

Po słowie kluczowym `Type` następuje zdefiniowanie dwóch typów użytkownika: `Array1`, `Array2`.

Delphi dodatkowo pozwala na przekazywanie tablic wprost w roli argumentu do wywoływanej funkcji. Wymaga to użycia ujętej w nawiasy kwadratowe listy, na której wartości poszczególnych elementów tablicy są rozdzielone przecinkami (ang. *comma-delimited list*). Oto przykład:

```
Function CalcAverage( X : Array of Real ) : Real;
Begin
 { Instrukcje definiujace cialo funkcji. }
End;

BEGIN
 WriteLn( 'Srednia z tablicy X1 = ',
  CalcAverage( [ 1.1, 2.2, 3.3, 4.0 ] ));
 WriteLn( 'Srednia z tablicy X2 = ',
  CalcAverage([ 44.1, 21.2, 3.0, 40.0, 99.0, 5.897, 6.0 ] ));
END.
```

1.2. Zmienna Result używana do zwrotu wartości przez funkcje

Delphi ułatwia tworzenie definicji funkcji poprzez automatyczną deklarację lokalnej zmiennej `Result` (rezultat) w obrębie każdej funkcji. Typ tej zmiennej jest zgodny z zadeklarowanym typem wartości zwracanej przez daną funkcję. W istocie zmienna `Result` to jedynie alias (alternatywna nazwa) nazwy funkcji. Przypisanie (ang. *assignment*) tej zmiennej jakiejkolwiek wartości zastępuje przypisanie tej samej wartości identyfikatorowi (nazwie) funkcji. Stosowanie takiej pomocniczej zmiennej jest wygodniejsze, gdyż nie koliduje z ewentualnym rekursywnym wywoływaniem funkcji. Tabela D.6 zawiera porównanie trzech równoważnych zapisów z użyciem zmiennej `Result`.

Tabela D.6. Sposoby użycia zmiennej Result w definicji funkcji

`Function F3(X : Real) : Real;`
`Begin`
` F3 := X * X * X;`
`End;`
	`Function F3(X : Real) : Real;`
	`Begin`
	` Result := X * X * X;`
	`End;`
		`Function F3(X : Real) : Real;`
		`Begin`
		` Result := X;`
		` Result := Result*X;`
		` Result := Result*X;`
		`End;`
		{ *bez rekursywnych wywolan F3()* }

UWAGA

Dopóki nie nastąpi przypisanie wartości, wartość zmiennej `Result` pozostaje nieokreślona.

1.3. Typ wartości zwracanej przez funkcję

Delphi pozwala, by funkcja zwracała wartość wynikową dowolnego typu. Może to być bazowy (prosty) typ danych, złożony typ danych, typ standardowy lub typ zdefiniowany przez użytkownika. Wyjątkami od tej zasady są „obiekty starego typu" (zgodne z konwencją Pascal 7) oraz pliki typu: `Text` lub `File Of`. W tych przypadkach funkcje Delphi mogą zwracać tylko wskaźniki do tych obiektów. Oto przykład funkcji zwracającej wynik typu rekordowego zdefiniowanego przez użytkownika (rekord odpowiada liczbie zespolonej o postaci `X + iY`):

```
Type
 Zespolone = Record
  X : Real;
```

```
  Y : Real;
 End;

Function DodajZespolone( L1, L2 : Zespolone ) : Zespolone;
Begin
 Result.X := L1.X + L2.X;
 Result.Y := L1.Y + L2.Y;
End;

Var Liczba1, Liczba2, Liczba3 : Zespolone;

BEGIN
 Liczba1.X := 1.0;
 Liczba1.Y := 2.0;
 Liczba2.X := 3.0;
 Liczba2.Y := 4.0;
 Liczba3 := DodajZespolone( Liczba1, Liczba2 );
 WriteLn( 'Wynik zespolony = ' , Liczba3.X  ' + ' , Liczba3.Y , 'i' );
END.
{ Oczekiwany wynik:  4 + 6i  }
```

Ta funkcja, posługując się predefiniowaną zmienną lokalną `Result`, zwraca rekord składający się z pola składowej rzeczywistej `X` i pola składowej urojonej `Y` liczby zespolonej.

1.4. Przekazanie argumentów przez wartość lub poprzez referencję

Zastosowanie funkcji nie jest jedynym sposobem zwracania wartości wynikowej z podprogramu — funkcje i procedury są łącznie nazywane **podprogramami** (ang. *subroutines*). Wbrew sztywnemu podziałowi, który wprowadzają systematycy, pascalowskie procedury mogą także modyfikować wartość zewnętrznej zmiennej, zwracając w ten sposób nowe wartości do programu wywołującego. Istota rozwiązania polega na sposobie przekazania zmiennej jako argumentu w momencie wywołania funkcji lub procedury.

Istnieją dwa sposoby przekazania zmiennej jako argumentu do funkcji lub do procedury:

- *by reference* — przez referencję do zmiennej,
- *by value* — przez wartość.

programowanie_strukturalne_DODATEK.indb 118 2010-07-05 11:54:51

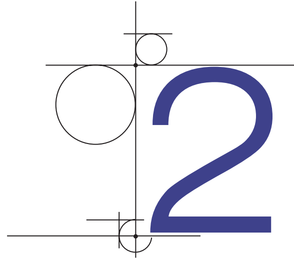
Jeśli przekazujemy zmienną przez wartość, funkcja/procedura tworzy sobie własną tymczasową kopię i bez względu na operacje wykonywane w ciele funkcji/procedury wartość zewnętrznej zmiennej pozostaje bez zmian. Jeśli natomiast przekazujemy zmienną poprzez referencję (odwołanie do niej), funkcja/procedura może zmienić (i zazwyczaj zmienia) wartość zmiennej zewnętrznej w programie wywołującym. Jeśli zastosujemy składnię:

```
Procedure DodajJeden( Liczba : Integer );     { przez wartosc }
Begin
 Liczba := Liczba + 1;
 WriteLn( 'Wewnatrz funkcji Liczba = ' , Liczba );
End;
```

— jako argument zostanie przekazana bieżąca wartość parametru wywołania i zmienna zewnętrzna pozostanie nienaruszona. Jeśli jednak dodamy słowo kluczowe `Var`, procedura zmodyfikuje zewnętrzną zmienną. Poniższy program demonstruje tę subtelną różnicę.

```
Procedure DodajJeden( Liczba : Integer );
Begin
 Liczba := Liczba + 1;
 WriteLn( 'Wewnatrz funkcji Liczba = ', Liczba );
End;

Procedure DodajJeden2( Var Liczba : Integer );
Begin
 Liczba := Liczba + 1;
End;

Var X : Integer;

BEGIN
 X := 0;
 DodajJeden( X );
 WriteLn( 'Zgodnie z oczekiwaniem X bez zmian = ' , X );
 DodajJeden2( X );
 WriteLn( 'A teraz zewn. zmienna X  = ' , X );
END.
```

Określenie „przez referencję” nie jest najszczęśliwsze, gdyż w istocie funkcja/procedura otrzymuje *wskaźnik do zmiennej*. Jest to instrukcja, w którym miejscu w pamięci umieścić nową, zmodyfikowaną wartość.



Przenoszenie programów Turbo Pascala do Delphi

Programy pisane w Turbo Pascalu lub Borland Pascalu mogą pracować w środowisku DOS w trybie tekstowym lub graficznym. Oba rodzaje programów można przenieść do środowiska Delphi/Windows, choć postępowanie w obu tych przypadkach musi być różne.

Wyobraźmy sobie, że mamy stary kod aplikacji graficznej napisanej w Turbo Pascalu, posługującej się grafiką BGI przeznaczoną dla środowiska DOS.

Listing D16.PAS

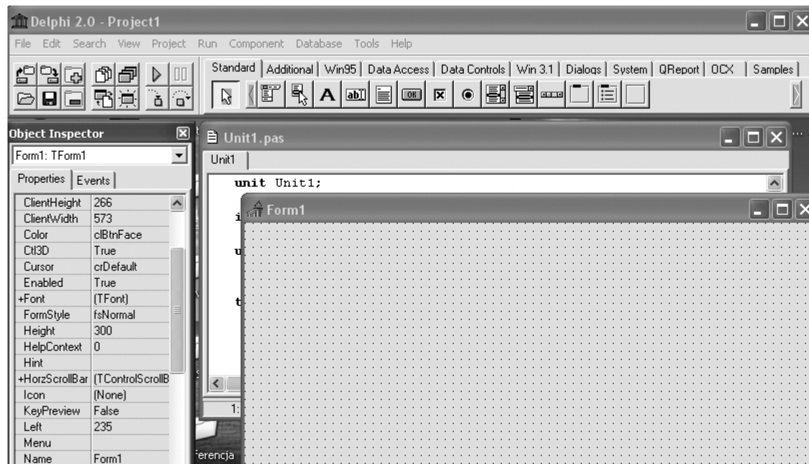
```
Program BGIGRAPH;
Uses Graph, Crt;
Var Karta, Tryb : Integer;

BEGIN
 Karta := Detect;
 InitGraph( Karta, Tryb, 'C:\TP\BGI');
 If GraphResult <> grOk  Then  Halt( 1 );
 SetBkColor( WHITE );
 SetColor( RED );
 Circle( 100, 100, 50 );
 Line( 20, 150, 100, 100 );
 Line( 100, 100, 180, 150 );
 OutTextXY( 10, 20, 'TEKST W APLIKACJI GRAFICZNEJ');
```

```
    Readln;
    CloseGraph;
  END.
```

Aby przenieść ten program do Delphi, należy wykonać następujące czynności przygotowawcze:

1. Uruchomić dowolny edytor, na przykład Notatnik Windows.
2. Otworzyć w Notatniku plik, w tym przypadku *D16.PAS*.
3. Zminimalizować Notatnik (by nie przeszkadzał w dalszej pracy).

Teraz uruchamiamy Delphi (patrz: rys. D18). Ponieważ w różnych szkołach mogą być stosowane różne wersje Delphi, tę pierwszą transformację zaprezentujemy, posługując się najstarszą, 32-bitową wersją Delphi 2. W nowszych wersjach możliwości (dostępnych komponentów, funkcji itp.) będzie więcej, ale zasady postępowania pozostają niezmienne, więc użytkownicy posługujący się nowszymi wersjami bez problemu dadzą sobie z tym zadaniem radę.



Rysunek D18. Delphi po uruchomieniu

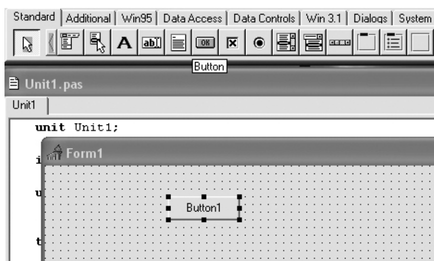
Na początku środowisko projektowe Delphi składa się z 4 okien:

- *Delphi 2.0 Project1* — menu i paleta komponentów wizualnych,
- *Object Inspector* — okno właściwości komponentów,
- *Unit1.pas* — okno edytora kodu,
- *Form1* — okno edytora graficznego.

Jednym ze sposobów rozwiązania naszego problemu może być wykonanie następujących czynności:

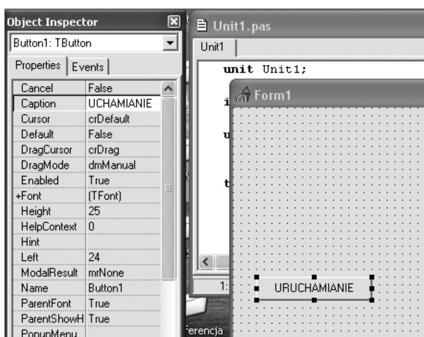
1. Na palecie komponentów wizualnych kliknięciem myszki wybieramy szósty od lewej przycisk sterujący `Button`. Po kliknięciu myszką dowolnego punktu okna

aplikacji *Form1* przycisk zostaje osadzony tak, jak pokazano to na rysunku D19 poniżej.



Rysunek D19.
Przycisk sterujący osadzony w oknie edytora

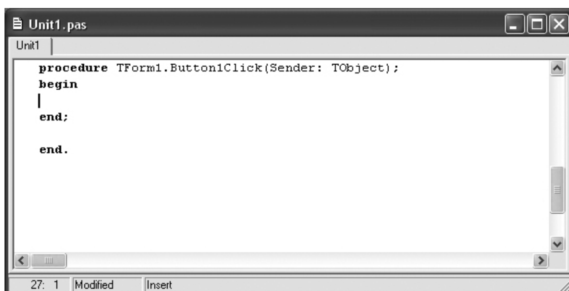
2. W oknie *Object Inspector* w polu właściwości `Caption` (napis na komponencie) wpisujemy na przykład `URUCHAMIANIE`. Przycisk możemy przeciągnąć myszką w dowolne miejsce. Możemy także zmienić jego rozmiary tak, by cały napis był widoczny, tak jak pokazano to na rysunku D20.



Rysunek D20.
Napis na przycisku

3. Podwójne kliknięcie myszką przycisku przenosi nas do okna edytora kodu. Kursor zostaje automatycznie umieszczony wewnątrz procedury `Button1Click()`.

Delphi generuje typowy, domyślny szablon kodu. Ta procedura to tzw. *Event Handler*, czyli **procedura obsługi zdarzenia**. Co bowiem, zdaniem Delhi, może przydarzyć się przyciskowi? Najbardziej typowe zdarzenie to `Click`, czyli kliknięcie go myszką. Nawet nietrudno się domyślić, prawda? Delphi jednakże nie wie, czego oczekuje programista po kliknięciu przycisku, czyli jak powinna przebiegać obsługa takiego zdarzenia. Pomiędzy słowa `begin` i `end;` (patrz: rysunek D21) musimy zatem wpisać własny kod.



Rysunek D21.
Miejsce wpisywania kodu

Procedurę `Button1Click()` definiujemy poprzez skopiowanie i wstawienie do jej wnętrza całego programu z Notatnika (*Copy/Paste*, *Ctrl+C/Ctrl+V*).

Teraz musimy dokonać pewnych modyfikacji kodu (edycja kodu — patrz rysunek D.13), wynikających z różnic między grafiką BGI dla DOS a grafiką GDI w Windows.

A. Usuwamy inicjację trybu graficznego (`InitGraph()`, `GraphResult`) oraz procedurę zamykającą tryb graficzny `CloseGraph()`. Tryb graficzny panuje w Windows z definicji i nie musimy go inicjować.

B. Pozostałe instrukcje graficzne poprzedzamy słowem `Canvas` (płótno) z kropką, na przykład:

   ```
   Line()    →    Canvas.Line()
   ```

C. Ponieważ nie wszystkie instrukcje BGI mają swoje dokładne odpowiedniki, niektóre musimy nieco zmodyfikować, na przykład:

   ```
   OutTextXY()    →    Canvas.TextOut()
   ```

   Podobnie rysowanie linii:

   ```
   Canvas.Line( 20, 150, 100, 100 );
   Canvas.Line( 100, 100, 180, 150 );
   ```

   — musi zostać zastąpione przez:

   ```
   Canvas.MoveTo( 20, 150 );
   Canvas.LineTo( 100, 100 );
   Canvas.LineTo( 180, 150 );
   ```

   — natomiast `Circle()` musi zastąpić elipsa wpisana w prostokąt o zadanych współrzędnych.

D. Ponieważ w nieco inny sposób określa się kolory tła i rysunku, czerwone pióro rysujące linie ustawiamy w Delphi tak:

   ```
   Canvas.Pen.Color := clRED;
   ```

   — a białe tło formularza tak:

   ```
   Form1.Color := clWHITE;
   ```

   Przy okazji zwróćmy uwagę, że właściwości obiektu `Form1` można ustawiać także w okienku *Object Inspector*. Z kolei właściwości pola rysunkowego (obiektu `Canvas`) nie można ustawiać w sposób statyczny, a tylko podczas działania programu (jest to tzw. `RunTime Property` — właściwość dostępna w czasie działania aplikacji).

E. Usuwamy jedną nadmiarową parę słów `BEGIN`, `END`.

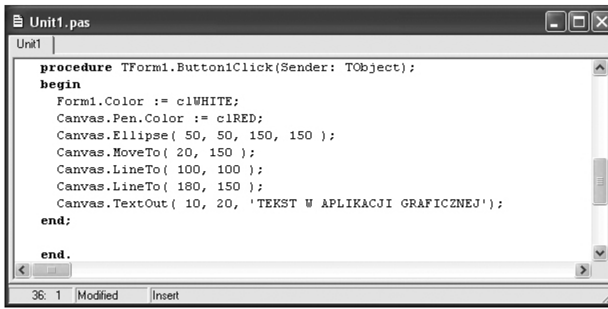
F. Usuwamy zbędną klauzulę `Uses Graph, Crt;`.

G. Usuwamy deklarację zbędnych zmiennych: `Var Karta, Tryb : Integer`.

Po dokonaniu tych modyfikacji kod Delphi wygląda tak jak na listingu poniżej.

Listing D17.PAS

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Color := clWHITE;
  Canvas.Pen.Color := clRED;
  Canvas.Ellipse( 50, 50, 150, 150 );
  Canvas.MoveTo( 20, 150 );
  Canvas.LineTo( 100, 100 );
  Canvas.LineTo( 180, 150 );
  Canvas.TextOut( 10, 20, 'TEKST W APLIKACJI GRAFICZNEJ');
end;
```

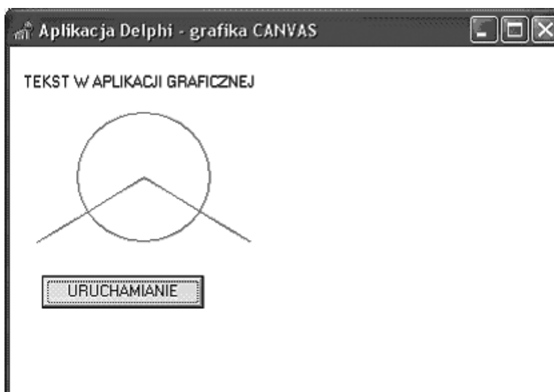


Rysunek D22. Kod, wstawiony do szablonu procedury obsługi zdarzenia, wykona się po kliknięciu przycisku

Aplikacja, choć nieco niedoskonała, jednak działa, rysując wymagane elementy graficzne. Dla porządku ustawmy właściwość w oknie inspektora:

```
Form1 | Caption = Aplikacja Delphi - grafika CANVAS
```

4. Teraz wystarczy już tylko wybrać polecenie *Run/Run* i uruchomić aplikację (rysunek D23).



Rysunek D23. Aplikacja Delphi w działaniu

Zwróćmy uwagę, że kliknięcie za pierwszym razem powoduje tylko zmianę tła formularza na biały, natomiast dopiero ponowne naciśnięcie przycisku drukuje tekst i rysuje

figury. Aby uniknąć dwukrotnego klikania, możemy ustawić kolor formularza na biały w oknie inspektora:

```
Form1 | Color = clWhite
```

— a kod procedury uzupełnić o fragment umożliwiający zniknięcie i (lub) zablokowanie klawisza, jeśli chcemy rysować tylko raz:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.Pen.Color := clRED;
  Canvas.Ellipse( 50, 50, 150, 150 );
  Canvas.MoveTo( 20, 150 );
  Canvas.LineTo( 100, 100 );
  Canvas.LineTo( 180, 150 );
  Canvas.TextOut( 10, 20, 'TEKST W APLIKACJI GRAFICZNEJ, v. 2');
  Button1.Visible := False;     { niewidoczny }
  Button1.Enabled := False;     { niedostępny }
end;
```

Zgodnie z przewidywaniem klawisz stanie się niewidoczny. Zwróćmy uwagę, że niewidoczne elementy sterujące mogą działać (choć taka zabawa w kotka i myszkę z użytkownikiem nie bardzo miałaby sens). Dopiero ustawienie właściwości `Enabled` (dostępny) w stan `False` (fałsz) blokuje klawisz na stałe.

UWAGA

W tym przykładzie nie modyfikujemy reszty kodu wygenerowanego przez Delphi automatycznie.

2.1. Aplikacje pracujące w trybie tekstowym

Opisane wcześniej dwa sposoby przekazywania argumentów do funkcji/procedur Pascala — poprzez referencję (z użyciem słowa `Var`) i poprzez wartość (bez stosowania słowa `Var`) — zostaną połączone w tym podrozdziale z „udawaniem” pracy w trybie tekstowym (tak jak Turbo Pascal w DOS). Jedno rozwiązanie nasuwa się samo. Wystarczy użyć metody `Canvas.TextOut()` zamiast `Write()` czy `WriteLn()` i tekst zostanie wyprowadzony na ekran. Program przykładowy można by zapisać tak:

```
{ Krok 1 - jeszcze niegotowe  }


X := 0;
DodajJeden( X );
```


programowanie_strukturalne_DODATEK.indb 125 2010-07-05 11:54:52

```
Canvas.TextOut( 10, 20, 'X bez zmian = ' , X );
....
```

Pozostaje problem zamiany liczby całkowitej — wartości zmiennej `X` — na łańcuch znaków. Musimy w tym celu posłużyć się zdefiniowaną w module `SysUtils` funkcją biblioteczną `IntToStr()`, która zwraca liczbę całkowitą w postaci łańcucha znaków (cyfr). Po takim zmodyfikowaniu kod procedury `Button1Click()` będzie wyglądać następująco:

```
procedure TForm1.Button1Click(Sender: TObject);
var X : Integer;
begin
  X := 0;
  DodajJeden( X );
  Canvas.TextOut( 10, 20, 'Zgodnie z oczekiwaniem X bez zmian = '
    + IntToStr( X ) );
  DodajJeden2( X );
  Canvas.TextOut( 10, 40, 'Zgodnie z oczekiwaniem X bez zmian = '
    + IntToStr( X ) );
end;
```

Graficzny układ formularza i klawisza uruchamiającego może pozostać bez zmian, ale co z deklaracjami i definicjami naszych własnych procedur? Deklaracje (prototypy) należy umieścić w części interfejsowej modułu (rozpoczynającej się słowem `interface`), a definicje w części implementacyjnej (po słowie `implementation`). Zwróćmy uwagę, że zmienne lokalne, potrzebne tylko w obrębie danej procedury, deklarujemy pomiędzy nagłówkiem (prototypem) a początkiem ciała procedury, tj. przed pierwszym słowem kluczowym `begin`. Oto pełny tekst modułu *D16.PAS* po dokonaniu takich modyfikacji.

Listing D18.PAS

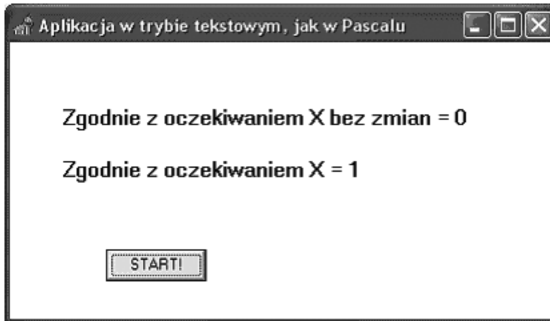
```
unit Unit1;


interface


uses  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;


type
  TForm1 = class(TForm)
```

```
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

procedure DodajJeden( Liczba : Integer );
procedure DodajJeden2( var Liczba : Integer );

var  Form1: TForm1;

implementation

{$R *.DFM}

procedure DodajJeden( Liczba : Integer );
begin
  Liczba := Liczba + 1;
end;

procedure DodajJeden2( var Liczba : Integer );
begin
  Liczba := Liczba + 1;
end;

procedure TForm1.Button1Click(Sender: TObject);
var  X : Integer;
begin
  X := 0;
  DodajJeden( X );
  Canvas.TextOut( 10, 20, 'Zgodnie z oczekiwaniem X bez zmian = '
    + IntToStr( X ) );
  DodajJeden2( X );
```


programowanie_strukturalne_DODATEK.indb 127 2010-07-05 11:54:53

```
      Canvas.TextOut( 10, 40, 'Zgodnie z oczekiwaniem X = '
        + IntToStr( X ) );
    end;
    end.
```

Działanie aplikacji wyraźnie potwierdza (co widać na rysunku D24), że przekazanie argumentu przy użyciu słowa kluczowego `var` powoduje zmodyfikowanie zmiennej `X`, która jest zewnętrzną zmienną w stosunku do procedury `DodajJeden2()`.



Rysunek D24.
Aplikacja Pascala w Delphi

Po przeniesieniu do Delphi aplikacja Turbo Pascala działa zgodnie z oczekiwaniem.

Sposób ten jest dość skomplikowany. W kolejnym podrozdziale omówiona zostanie prostsza metoda przenoszenia tekstowych aplikacji z Turbo Pascala do Delphi.

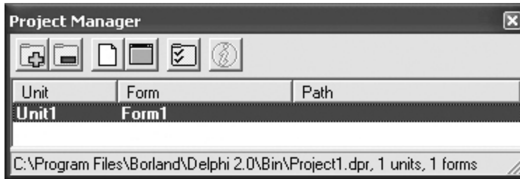
2.2. Aplikacje konsoli w Delphi

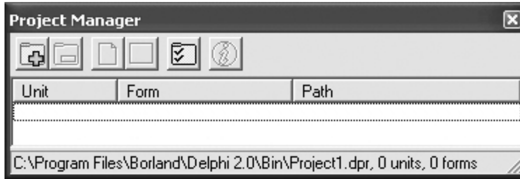
Sposób budowania aplikacji konsoli (które w Windows pracują w trybie tekstowym) jest inny w różnych wersjach Delphi. W Delphi 1 można utworzyć projekt kategorii *CrtApp* (aplikacja konsoli) i dołączyć plik *WinCrt*.

W Delphi 2 nie ma już takiej możliwości, ale istnieje stosunkowo prosta metoda, by ominąć wszelkie niewygody. Po rozpoczęciu pracy nad nowym projektem aplikacji (*File/New Application*) Delphi automatycznie tworzy strukturę projektu składającą się z kilku plików źródłowych i z zasobów graficznych — pliku z projektem formularza. Menedżer projektu dostępny z menu *View/Project Manager* pozwala dołączać lub usuwać pliki składowe (Rys. D.25).

Polecenie *Remove unit* pozwala usunąć automatycznie wygenerowany moduł *Unit1.pas* z projektu Delphi.

Przycisk ze znakiem *[+]* dodaje, a przycisk ze znakiem *[–] (Remove unit)* pozwala usunąć plik modułu, na przykład `Unit1`. Zwróćmy uwagę, że usuwając moduł *Unit1.pas*, usuwamy jednocześnie z projektu formularz (w wierszu statusu pojawi się wpis *0 forms*).





Rysunek D25. Formularz Form1 i moduł Unit1 zostały usunięte

Pozostał nam jednak plik *Project1.dpr*. Rozszerzenie *DPR* pochodzi od *Delphi PRoject*. Za pomocą polecenia *View/Project Source* możemy poddać edycji kod źródłowy pliku projektu. Jego postać wyjściowa będzie wyglądać tak:

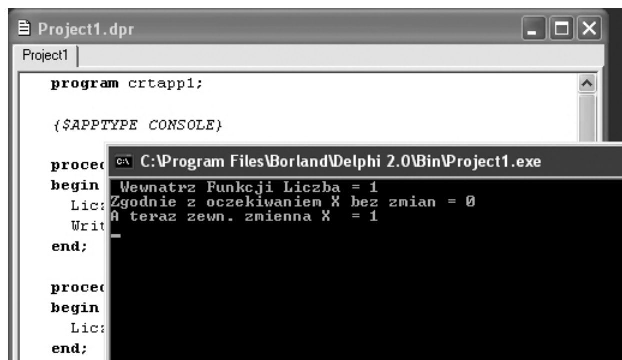
```
program Project1;
uses Forms;

{$R *.RES}

begin
  Application.Initialize;
  Application.Run;
end.
```

Ten automatycznie utworzony szkielet jest nam zupełnie nieprzydatny, ale możemy skopiować do tego pliku kod Turbo Pascala przeznaczony do pracy w trybie tekstowym, doprowadzając go na przykład do poniższej postaci:

```
program crtapp1;

{$APPTYPE CONSOLE}

procedure DodajJeden( Liczba : Integer );
begin
  Liczba := Liczba + 1;
  WriteLn( ' Wewnatrz Funkcji Liczba = ', Liczba );
end;
```

```
procedure DodajJeden2( Var Liczba : Integer );
begin
  Liczba := Liczba + 1;
end;

var  X : Integer;

begin
  X := 0;
  DodajJeden( X );
  WriteLn( 'Zgodnie z oczekiwaniem X bez zmian = ' , X );
  DodajJeden2( X );
  WriteLn( 'A teraz zewn. zmienna X  = ' , X );
  ReadLn;
end.
```

Po dodaniu dyrektywy kompilatora: `{$APPTYPE CONSOLE}` określającej docelowy typ aplikacji, kompilator Delphi utworzy aplikację konsoli o zadanej nazwie, na przykład *CRTAPP1.EXE*, która po wydaniu polecenia *Run/Run* będzie uruchamiać się w oknie sesji DOS, jak pokazano na rysunku D26.



Rysunek D26. Aplikacja konsoli w Delphi działa tak jak w Turbo Pascalu

To oczywiście najprostszy sposób uruchamiania kodów Turbo Pascala w środowisku Delphi. Możliwość ta rzadko służy programistom do tworzenia prawdziwych aplikacji, ale często przydaje się do testowania stosunkowo krótkich, a ważnych fragmentów kodu aplikacji. Uniezależnienie się na chwilę od całej graficznej maszynerii pozwala skoncentrować się na detalach, a diabeł zazwyczaj tkwi właśnie w szczegółach.

2.3. Aplikacje konsoli tworzone za pomocą kreatora

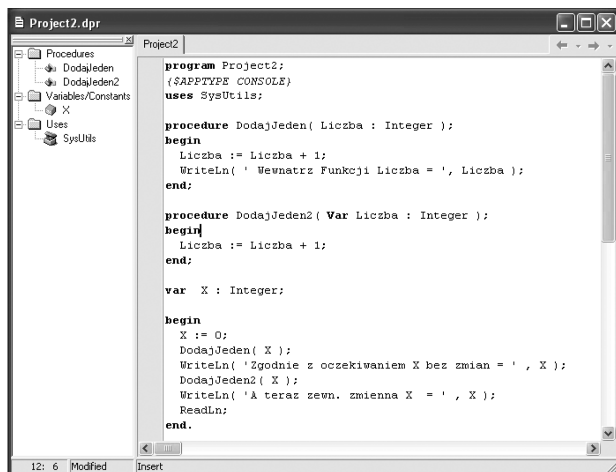
W nowszych wersjach Delphi mamy do dyspozycji rozbudowane automatyczne kreatory. Na przykładzie Delphi 5 pokazano tu, jak do utworzenia aplikacji konsoli można wykorzystać kreatora aplikacji.

Po wybraniu z menu polecenia *File/New...* z zakładki *New* możemy wybrać stosowny kreator (rysunek D27).

Rysunek D27. Kreator Delphi pozwala wybrać rodzaj aplikacji

Po wybraniu *Console Application* pojawia się okno edytora kodu (rysunek D28) z wygenerowanym szablonem:

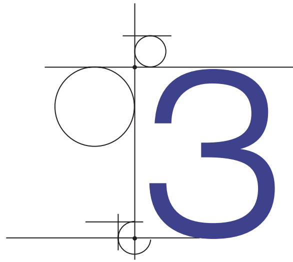
```
program Project2;

{$APPTYPE CONSOLE}

uses SysUtils;


begin
  // Insert user code here   { tu wstaw kod }
end.
```



Rysunek D28. Okno edytora kodu Delphi przygotowane do kompilacji i uruchomienia

Aplikacja po uruchomieniu ruszy w oknie sesji DOS (lub DosBox, w Win 7), podobnie jak pokazano to na jednym z poprzednich rysunków. Dodatkowo boczny panel okna edytora kodu w Delphi pozwala łatwiej zorientować się w konstrukcji programu, pokazując listę zmiennych, stałych i procedur.

programowanie_strukturalne_DODATEK.indb 132 2010-07-05 11:54:53



O konwersji typów danych

Ponieważ problem wyprowadzenia danych tekstowych na ekran w Delphi zazwyczaj sprowadza się do manipulowania łańcuchami znakowymi, warto choćby zasygnalizować typowy kłopot programistów — konieczność częstych konwersji typów danych. Służą do tego niektóre funkcje biblioteczne Delphi, ale czasem warto napisać własną funkcję.

3.1. Własne funkcje konwersji typów

W przypadku liczb rzeczywistych można dzięki własnej funkcji na przykład zapanować nad precyzją odwzorowania, czyli ilością cyfr znaczących liczby, z których zostanie utworzony łańcuch znakowy. Oto przykład takiej funkcji konwersji:

```
function RealNaString( L : Real; Szer, Miejsca : Integer ) : String;
var StringBufor : String;
begin
  Str( L:Szer:Miejsca, StringBufor );
  Result := StringBufor;
end;
```

Przykład poniżej obrazuje wywołanie tej funkcji w prostym kodzie przykładowej aplikacji konsoli w celu zaokrąglenia liczby zmiennoprzecinkowej i wydrukowania jej tylko z dwoma cyframi po przecinku.

Listing D19.PAS

```
function RealNaString( L : Real;  Szer, Miejsca : Integer ) : String;
var StringBufor : String;
begin
  Str( L:Szer:Miejsca, StringBufor );
```

```
  Result := StringBufor;
end;


begin
  WriteLn( RealNaString( 1.23456, 3, 2 ) );
  ReadLn;
end.
```

Jak łatwo się zorientować, rezultatem tego programu będzie 1.23. Procedura biblioteczna `Str(X [: Width [: Decimals ]]; var S);` dokonuje konwersji wyrażenia numerycznego `X` na łańcuch tekstowy i zapisuje go do bufora `S`.

Jeśli w Delphi pobieramy dane wpisane przez użytkownika do pola edycyjnego, pobieramy w istocie łańcuchy znaków, które następnie trzeba poddać konwersji — z łańcucha znaków na liczbę, całkowitą lub rzeczywistą. Występuje więc często potrzeba wykonania konwersji odwrotnej. I tym razem można pokusić się o napisanie i przetestowanie własnej funkcji. Ma to tę zaletę, że pisząc własne funkcje, panujemy nad tym, co się dzieje.

```
function StrNaReal( StringBufor : String ) : Real;
var ErrorCode : Integer;
  Temp : Real;
begin
  if Length(StringBufor) = 0
  then Result := 0
  else
  begin
    Val( StringBufor, Temp, ErrorCode );
    if ErrorCode = 0
    then Result := Temp
    else Result := 0;
  end;
end;
```

Działanie tej funkcji konwersji można sprawdzić, umieszczając jej wywołanie w prostym, szkieletowym programie testującym.

Listing D20.PAS

```
function StrNaReal( StringBufor : String ) : Real;
var ErrorCode : Integer;
  Temp : Real;
```


programowanie_strukturalne_DODATEK.indb 134 2010-07-05 11:54:53

```
begin
  if Length(StringBufor) = 0
  then StrNaReal := 0
  else
  begin
    Val( StringBufor, Temp, ErrorCode );
    if ErrorCode = 0
    then StrNaReal := Temp
    else StrNaReal := 0;
  end;
end;

var Bufor : String;

begin
  Bufor := '12345.6789';
  WriteLn( 'KONWERSJA OK: ', StrNaReal( Bufor ) );
  Bufor := '1.2.3.4.5';
  WriteLn( 'BZDURA: ', StrNaReal( Bufor ) );
  ReadLn;
end.
```

Wydruk tego programu będzie następujący:

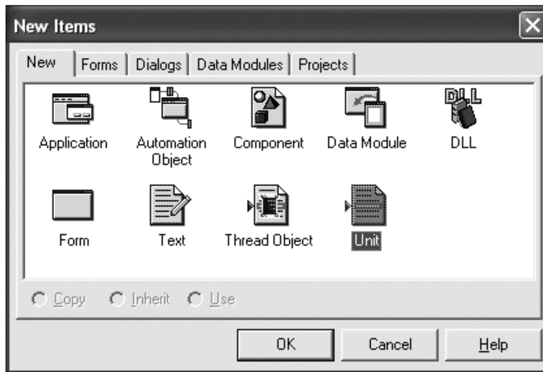
```
KONWERSJA OK:  1.23456789000000E+0004
BZDURA:  0.00000000000000E+0000
```

Własne funkcje i procedury można wykorzystywać wielokrotnie. Można przechowywać je w plikach tekstowych, ale można także stworzyć dla nich odrębny moduł. Z punktu widzenia metodyki programowania to cenna możliwość. Moduł taki można udostępniać innym programistom.

3.2. Funkcje konwersji typów w Delphi

Tak opracowane własne funkcje najlepiej przechowywać w odrębnym module, który możemy nazwać w zasadzie dowolnie (np. `AMUnit`). Aby utworzyć taki moduł, należy wybrać polecenie *File/New...*. Pojawi się widoczne na rysunku D29 okienko *New Items* (Delphi 2), w którym klikamy ikonę *Unit* i zamykamy okienko przyciskiem *OK*.

programowanie_strukturalne_DODATEK.indb 135 2010-07-05 11:54:53



Rysunek D29.
Własny moduł Delphi — okienko kreatora

Do wygenerowanego szkieletu:

```
unit Unit1;

interface

implementation

end.
```

— wstawiamy kody własnych funkcji w części implementacyjnej, a deklaracje (prototypy) kopiujemy dodatkowo do części interfejsowej. Po wykonaniu tych operacji kod modułu powinien wyglądać tak:

```
unit AMUnit;    { Nazwa zmieni się automatycznie po zapisaniu: File/Save As... }

interface

function RealNaString( L : Real;  Szer, Miejsca : Integer ) : String;
function StrNaReal( StringBufor : String ) : Real;

implementation

function RealNaString( L : Real;  Szer, Miejsca : Integer ) : String;
var StringBufor : String;
begin
  Str( L:Szer:Miejsca, StringBufor );
  Result := StringBufor;
end;
```

```
function StrNaReal( StringBufor : String ) : Real;
var ErrorCode : Integer;
  Temp : Real;
begin
  if StringBufor[0] = #0
  then Result := 0
  else
  begin
    Val( StringBufor, Temp, ErrorCode );
    if ErrorCode = 0
    then Result := Temp
    else Result := 0;
  end;
end;
end.
```

Jeśli zamierzamy użyć tych funkcji, musimy dodać moduł do klauzuli `Uses`, na przykład:

```
Uses AMUnit, Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms;
```

Jeśli teraz pole edycyjne, do którego użytkownik wpisuje liczby rzeczywiste, będzie w naszej aplikacji nazywać się `Edit1` (domyślna nazwa), sposób użycia może być następujący:

```
Uses AMUnit, .....
...
procedure TForm1.Button1Click( Sender : TObject );
var Liczba : Real;
begin
  Liczba := StrNaReal( Edit1.Text );
  ....
```

Zapis `Edit1.Text`, utrzymany w konwencji obiekt-kropka-właściwość, oznacza łańcuch znaków znajdujący się w danej chwili w polu edycyjnym `Edit1`.

W tabeli D.7 podano funkcje biblioteczne Delphi (*Unit SysUtils*) służące do konwersji najczęściej stosowanych typów danych.

programowanie_strukturalne_DODATEK.indb 137 2010-07-05 11:54:53

Tabela D.7. Najczęściej stosowane funkcje konwersji w Delphi

NAZWA	Znaczenie	Co robi?
`IntToStr`	*Integer To String*	Liczba całkowita na łańcuch znaków
`IntToAscii`	*Integer To ASCII*	Liczba całkowita na łańcuch znaków ASCII
`IntToHex`	*Integer To Hexadecimal*	Liczba całkowita dziesiętna na postać szesnastkową
`StrToInt`	*String To Integer*	Łańcuch cyfr na liczbę całkowitą
`StrToIntDef`	*String To Integer or Default*	Jw., ale w razie błędu zwraca wartość domyślną
`StrToFloat`	*String To Floating Point Number*	Łańcuch znaków na liczbę zmiennoprzecinkową
`StrToDate`	*String To Date*	Łańcuch znaków na typ data
`StrToTime`	*String To Time*	Łańcuch znaków na typ czas
`StrToDateTime`	*String To Date/ Time*	Łańcuch znaków na typ data/czas
`FloatToStr`	*Float To String*	Liczba zmiennoprzecinkowa na łańcuch znaków
`FloatToStrF`	*Float To String Formatted*	Liczba zmiennoprzecinkowa na łańcuch formatowany (typy: `Single`, `Double`, `Extended`)
`FloatToText`	*Floating Point Number To Text*	Liczba zmiennoprzecinkowa na tekst

Jest tych funkcji znacznie więcej, ale nie chodzi o zapamiętanie ich nazw, gdyż nie to uczyni z Ciebie programistę, lecz o zapamiętanie, że liczna grupa takich funkcji istnieje, a ich sens i zastosowanie są takie, jak opisano powyżej.

UWAGA

W Delphi składnia typowa dla Turbo Pascala:

```
If StringBufor[0] = #0
```

— może powodować komunikat o błędzie: *Element 0 inaccessible, use Length* (zerowy element niedostępny, użyj funkcji `Length`). Po dokonaniu takiej modyfikacji:

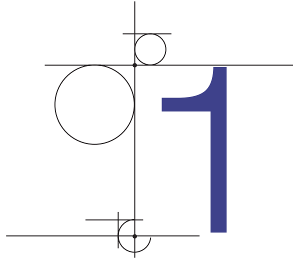
```
if ( Length( StringBufor ) = 0 ) then Result := 0 else { ... }
```

— kompilacja powiedzie się, a plik *AMUnit.dcu* pojawi się w docelowym folderze, tu: *\LIB*.

programowanie_strukturalne_DODATEK.indb 138 2010-07-05 11:54:54

Część V

C++ — zaawansowane programowanie obiektowe



O referencjach w C++

Typowy (domyślny) sposób przekazywania argumentów do funkcji w C++ polega na tzw. **przekazaniu przez wartość** i jest inny niż w Pascalu czy Basicu. Ponieważ do C/C++ większość adeptów dojrzewa po przebrnięciu przez Basic lub Pascal, programiści ci obciążeni są już pewnymi nawykami i schematami myślenia, których do C++ niestety nie da się zastosować. To, co w Basicu wygląda zrozumiale:

```
PRINT X          REM Wyprowadź bieżącą wartość zmiennej X
INPUT X          REM Pobierz wartość zmiennej X
```

— równie zrozumiale wygląda w Pascalu:

```
WriteLn(X);      { Wyprowadź bieżącą wartość zmiennej X }
ReadLn(X);       { Pobierz wartość zmiennej X }
```

W C++ może to wyglądać równie niewinnie:

```
cout << X;       //Wyprowadź wartość zmiennej X
cin >> X;        //Pobierz wartość zmiennej X
```

— ale czasem przyjmuje formę zapisu wyraźnie dualnego:

```
printf("%d", X);   //Wyprowadź wartość zmiennej X
scanf("%d", &X);   //Pobierz wartość zmiennej X
```

Na czym polega różnica? W pierwszym przypadku — do wyprowadzania istniejących już danych:

```
PRINT,
WriteLn(),
printf()
```

— w celu poprawnego działania funkcji powinniśmy przekazać jej **bieżącą wartość argumentu** X (ang. *by current value*), natomiast adres zmiennej w pamięci nie jest

funkcji potrzebny. Dla Basica, Pascala i C++ bieżąca wartość zmiennej kojarzona jest z jej identyfikatorem, w tym przypadku z `X`.

W drugim jednakże przypadku (pobranie danych i umieszczenie ich pod właściwym adresem pamięci) jest inaczej. Funkcji zupełnie nie interesuje bieżąca wartość zmiennej `X`, jest jej natomiast do poprawnego działania potrzebny adres zarezerwowany dla zmiennej `X` w pamięci. Basic i Pascal postępują dokładnie tak samo jak poprzednio:

```
INPUT X      i      Read(X), ReadLn(X);
```

`X` nie oznacza dla Pascala i Basica bieżącej wartości zmiennej, lecz oznacza (domyślnie) przekazanie do funkcji adresu zmiennej `X` w pamięci. Funkcje biblioteczne oczywiście „wiedzą”, co dostały, i dalej już same manipulują danymi we właściwy sposób. W C++ jest inaczej. Zapis:

```
Funkcja(X);
```

— oznacza w praktyce, że zostaną wykonane następujące operacje:

- Spod adresu pamięci przeznaczonego dla zmiennej `X` zostanie (zgodnie z zadeklarowanym formatem) odczytana bieżąca wartość zmiennej `X`.
- Wartość `X` zostanie zapisana na stos (`PUSH X`).
- Zostanie wywołana funkcja `Funkcja()`.
- `Funkcja()` pobierze wartość argumentu ze stosu (zgodnie z formatem zadeklarowanym w prototypie funkcji `Funkcja()`).
- `Funkcja()` zadziała zgodnie ze swoją definicją i jeśli ma coś do zwrócenia, zwróci wynik.

Jak widać, funkcja w C/C++ „nie wie”, gdzie w pamięci umieszczony był przekazany jej argument:

- Funkcja komunikuje się z własnym programem lub funkcją wyższego rzędu (wywołującą) tylko za pośrednictwem stosu.
- Funkcja dostaje swoją „kopię” argumentu, na którym działa.
- Funkcja nie ma wpływu na „oryginał” argumentu, który pozostaje bez zmian.

Czym właściwie jest referencja zmiennej w C++? Jest to alternatywny sposób odwołania się do zmiennej. Zacznijmy od trywialnego przykładu odwołania się do tej samej zmiennej mającej swoją nazwę `zmienna` i referencję `ksywa`.

```
#include <iostream.h>

main()

{

  int zmienna;

  int& ksywa;

 ...
```

Aby `ksywa` oznaczała tę samą zmienną, referencję należy zainicjować:

```
int& ksywa = zmienna;
```

Zainicjujemy zmienną `zmienna` i będziemy robić z nią cokolwiek (np. inkrementować). Jednocześnie sprawdzimy, czy odwołania do zmiennej za pomocą nazwy i referencji pozostaną równoważne.

Listing D21.CPP

```
#include <iostream.h>


main()
{
  int zmienna = 6666;
  int& ksywa = zmienna;

  cout << '\n' << "Zmienna" << " Ksywa";
  cout << '\n' << zmienna << '\t' << ksywa;

  for (register int i = 0; i < 5; i++, zmienna += 100)
      cout << '\n' << zmienna << '\t' << ksywa;

  return 0;
}
```

Wydruk powinien wyglądać tak:

```
Zmienna Ksywa
6666    6666
6666    6666
6766    6766
6866    6866
6966    6966
7066    7066
```

Zastosujmy program poprzedni, ale zainicjujmy referencję powtórnie.

```
  int pierwsza = 1111, druga = 2222, trzecia = 3333;

  int& ksywa = pierwsza;
  cout << "\nZmienna i ADRES   Ksywa i ADRES: \n\n";
  cout << pierwsza << "\t\t" << ksywa << '\n';
```

```
    cout << hex << &pierwsza << "\t\t" << &ksywa;

    ksywa = druga;
    cout << "\n\n";
    cout << dec << druga << "\t\t" << ksywa << '\n';
    cout << hex << &druga << "\t\t" << &ksywa;

    ksywa = trzecia;
    cout << "\n\n";
    cout << dec << trzecia << "\t\t" << ksywa << '\n';
    cout << hex << &trzecia << "\t\t" << &ksywa << '\n';
```

Tym razem wydruk programu będzie wyglądał tak:

```
Zmienna i ADRES       Ksywa i ADRES:

1111                  1111
0x2af10ffe            0x2af10ffe

2222                  2222
0x2af10ffc            0x2af10ffe

3333                  3333
0x2af10ffa            0x2af10ffe
```

Adres referencji „ksywa" pozostaje niezmienny, natomiast kolejne zmienne lokalne funkcji `main()` mają coraz mniejsze adresy, zmieniające się o dwa bajty. Tym razem referencja zachowuje się jak oddzielna zmienna.

Na koniec tych rozważań jeszcze przykład zainicjowania referencji poprzez przypisanie. Obie referencje są referencjami do tej samej stałej:

```
int& stala = 12345;
int& kontrolna = stala;
cout << '\n' << stala << '\t' << kontrolna;
```

1.1. Referencje w roli argumentów funkcji

Zamiast przekazywać do funkcji kopię argumentu-obiektu, można przekazać wskaźnik lub referencję. Jest to szczególnie sensowne w wypadku dużych tablic, struktur lub obiektów.

programowanie_strukturalne_DODATEK.indb 144 2010-07-05 11:54:54

Listing D22.CPP

```
#include <iostream.h>

/*  Przyklad duzej struktury: ----------------------------  */

struct Duza
{
  int numer;
  char TAB[30000L];                        //  duza tablica znakowa
} Wielka_Struktura = { 44, "To ja - struktura ! \
 \b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b"};

/*  Prototypy dwu funkcji, do ktorych struktura bedzie
    przekazywana jako argument: --------------------------  */

void WolnaF(Duza);            //  przekazanie kopii (przez wartosc)
void SzybkaF(Duza&);          //  przekazanie przez referencje

main()
{
  cout << '\n' << "To chwile potrwa...: ";
  for(register int i = 0; i < 10000; i++)
    WolnaF(Wielka_Struktura);

  cout << "\n Teraz bedzie szybciej ! ";
  for(i = 0; i < 10000; i++)
    SzybkaF(Wielka_Struktura);

  return 0;
}
/*  Definicje obu funkcji: ------------------------------- */

void WolnaF(Duza x)
{
  cout << x.numer;
```

```
  cout << "  " << x.TAB;
}

/*  ---- a tu przekazanie poprzez referencje: ---*/
void SzybkaF(Duza& x)
{
  cout << x.numer;
  cout << "  " << x.TAB;
}
```

Jeśli przy szybkości zegara konkretnego komputera różnica w prędkości działania funkcji będzie mało przekonująca, należy zmienić liczbę pętli w programie.

Listing D23.CPP

```
#include <iostream.h>

struct Duza
{
  int numer;
  char TAB[1000];
} Str = { 777, "To ja - struktura"};

/*  -  Prototypy funkcji: ---------------------------  */

void Fp(Duza*);          //  zastosowanie wskaznika do struktury
void Fr(Duza&);          //  zastosowanie referencji struktury

main()
{
  Fp(&Str);                //  funkcji przekazujemy adres argumentu
  Fr(Str);                 //  funkcji przekazujemy referencje

  return 0;
}

/*  - Definicja funkcji ze wskaznikiem: --------------------  */
void Fp(Duza *p)
```

```
{
  cout << '\n' << p->numer;          // sposob zastosowania
  cout << '\n' << p->TAB;            // wskaznika
}

/* - Definicja funkcji z zastosowaniem referencji: --------  */
void Fr(Duza& p)
{
  cout << '\n' << p.numer;           // sposob zastosowania
  cout << '\n' << p.TAB;             // referencji
}
```

UWAGA

Jeśli jedna funkcja (wywołująca, tu: `main()`) przekazuje referencję innej funkcji (wywoływanej, tu: `Fr()`), to trzeba pamiętać, że funkcja wywoływana działa na kopii referencji należącej do funkcji wywołującej.

Możemy udostępnić funkcji zewnętrzne dane (np. `scanf()`), ale jeśli zapomnimy o specyfice notacji w C/C++, dane te zostaną zmodyfikowane przez funkcję! W przykładzie poniżej funkcja `SwapF()` dokonuje zamiany zewnętrznych parametrów należących do funkcji wywołującej.

```
#include <iostream.h>
class Data
{
  int dz, mc, rok;
public:
  Data() {};                  //Pusty konstruktor domyslny
  Data(int x, int y, int z) {dz = x; mc = y; rok = z;}
  void Pokazuj();             //Metoda z prawem dostepu do danych
};

void Data::Pokazuj(void)  { cout << dz << '.' << mc << '.' << rok; }
void SwapF(Data&, Data&);
void Pokazuj(Data&, Data&);
```

```
main()
{
  Data dzis(11, 10, 2009);
  Data kiedys(24, 12, 1999);

  Pokazuj(dzis, kiedys);
  SwapF(dzis, kiedys);
  Pokazuj(dzis, kiedys);

  return 0;
}

//  ----- Funkcja zmieniajaca zewnetrzne dane dzieki referencji

void SwapF(Data& dt1, Data& dt2)
{
  Data schowek;
  schowek = dt1;
  dt1 = dt2;
  dt2 = schowek;
}

void Pokazuj(Data& dzis, Data& kiedys)
{
  cout << "\n Dzis mamy:  ";
  dzis.Pokazuj();
  cout << "\n A wtedy byl: ";
  kiedys.Pokazuj();      //Tu wywolujemy metode
}
```

W przykładzie warto przyjrzeć się uważnie:

- sposobowi dostępu do wewnętrznych danych obiektu,
- inicjowaniu obiektów `dzis`, `kiedys` i `schowek`.

1.2. Funkcje mogą zwracać i wskaźnik, i referencję

Funkcja może nie tylko pobierać referencje do argumentów, ale także zwracać referencje. Jeśli funkcja zwraca referencję, należy to zasygnalizować C++ w deklaracji funkcji:

```
typ& nazwa_funkcji(lista argumentow);
```

Funkcja `DajdateF()` z poniższego przykładu zwraca referencję:

```
#include <iostream.h>
#include <stdlib.h>

class Data
{
public:
  int dz, mc, rok;
};

/*  - Tablica zlozona z obiektow klasy Data: ---------  */

Data Urodziny[] = { {15, 9, 66}, {2, 7, 55}, {3, 11, 79} };

/*  - Funkcja pobierajaca date z tablicy: -------------  */

Data& DajdateF(int);

main(int argc, char *argv[])
{
  if (argc > 1)
  {
    Data& x = DajdateF(atoi(argv[1]));
    cout << '\n' << "Funkcja odpowiada: " << '\n';
    cout << x.dz << '.' << x.mc << '.' << x.rok;
  }
  return 0;
}
```

```
/* Definicja funkcji: ---------------------------------- */

Data& DajdateF(int i)
{
  return (Urodziny[i - 1]);
}
```

Dialog z programem powinien wyglądać tak:

```
C:\>program 1
Funkcja odpowiada:
15.9.66
C:\>program 2
Funkcja odpowiada:
2.7.55
C:\>program 3
Funkcja odpowiada:
3.11.79
```

Wskaźniki i referencje do obiektów funkcjonują podobnie jak wskaźniki do struktur. Operator -> pozwala na dostęp zarówno do danych, jak i do funkcji. Dla przykładu wykorzystamy obiekt naszej prywatnej klasy `Licznik`.

```
class Licznik
{
public:
  char moja_litera;
  int ile;
  Licznik(char z) { moja_litera = z; ile = 0; }
  void Skok_licznika(void) { ile++; }
};
```

Aby w programie można było odwołać się do obiektu nie poprzez nazwę, a za pomocą wskaźnika, zadeklarujemy wskaźnik do obiektów klasy `Licznik`. Wskaźnik w programie możemy zastosować na przykład tak:

```
Licznik *p;
p->Skok_licznika();
```

Przed użyciem należy zainicjować wskaźnik w taki sposób, by wskazywał na obiekt-licznik. Inicjujemy go w taki sam sposób jak każdy inny wskaźnik.

```
p = &Obiekt;
```

Poniżej zademonstrowany został program przykładowy w całości.

programowanie_strukturalne_DODATEK.indb 150 2010-07-05 11:54:54

Listing D24.CPP

```
#include <ctype.h>
#include <iostream.h>

class Licznik
{
public:
  char moja_litera;
  int ile;
  Licznik(char z) { moja_litera = z; ile = 0; }
  void Skok_licznika(void) { ile++; }
};

main()
{
  char znak;
  cout << "\nPodaj litere do zliczania: ";
  cin >> znak;

  Licznik Obiekt1(znak), Obiekt2('a'), *p1, *p2;
  p1 = &Obiekt1;
  p2 = &Obiekt2;
  cout << "\nWpisz ciag znakow ";
  cout << "zakonczony kropka [.] i [Enter] \n";
  while(1)
  {
    cin >> znak;
    if(znak == '.') break;
    if(znak == p1->moja_litera) p1->Skok_licznika();
    if(znak == p2->moja_litera) p2->Skok_licznika();
  }
  cout << "\nBylo " << p1->ile;
  cout << " liter: " << p1->moja_litera;
  p1 = p2;
  cout << "\nBylo " << p1->ile;
```


programowanie_strukturalne_DODATEK.indb 151 2010-07-05 11:54:54

```
  cout << " liter: " << p1->moja_litera;
  return 0;
}
```

1.3. Wskaźnik specjalny this

Poświęcimy teraz chwilę pewnemu specjalnemu wskaźnikowi. Specjalnemu (i ważnemu) na tyle, że dosłużył się w C++ własnego słowa kluczowego `this`.

Każdej funkcji/metodzie zadeklarowanej wewnątrz klasy zostaje w momencie wywołania w niejawny sposób przekazany wskaźnik do obiektu (w stosunku do którego funkcja ma zadziałać). Wskaźnik wskazuje funkcji w pamięci ten obiekt, którego członkiem jest dana funkcja. Bez istnienia takiego właśnie wskaźnika nie moglibyśmy stosować spokojnie funkcji, ponieważ nie moglibyśmy odwoływać się do pola obiektu, nie wiedząc jednoznacznie, o który obiekt chodzi. Program posługuje się automatycznie niejawnym wskaźnikiem do obiektu (ang. *implicit pointer*). Możemy wykorzystać ten istniejący, choć do tej pory niewidoczny dla nas wskaźnik, posługując się słowem kluczowym `this` (ten). Korzystając z tego wskaźnika, funkcja może bez cienia wątpliwości zidentyfikować obiekt, z którym pracuje, a nie obiekt przypadkowy. Jeśli w programie zadeklarujemy klasę `Klasa`, a wewnątrz tej klasy metodę `Pokazuj()`:

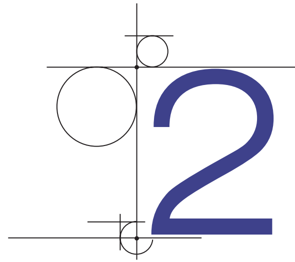
```
class Klasa
{
  int dane;
public:
  void Pokazuj();
  ...
void Klasa::Pokazuj(void) { cout << dane; }
```

— zdefiniowanie funkcji `Pokazuj()` z zastosowaniem wskaźnika `this` i notacji wskaźnikowej (->), jak poniżej, będzie równoważne:

```
void Klasa::Pokazuj(void)
{
  cout << this->dane;
}
```

Przypomnijmy, że taka notacja wskaźnikowa oznacza: „Wyprowadź zawartość pola `dane` obiektu, na który wskazuje wskaźnik” (ponieważ jest to wskaźnik `this`, więc chodzi o własny obiekt).

programowanie_strukturalne_DODATEK.indb 152 2010-07-05 11:54:55



Funkcje wirtualne i klasy abstrakcyjne

2.1. Funkcje wirtualne

Funkcje wirtualne przypominają funkcje rozbudowywane dzięki mechanizmowi przeciążania funkcji. Jeśli zdefiniowaliśmy w klasie bazowej funkcję wirtualną, to w klasie pochodnej możemy definicję tej funkcji zastąpić nową definicją. Przekonajmy się o tym na przykładzie. Zacznijmy od zadeklarowania funkcji wirtualnej (za pomocą słowa kluczowego `virtual`) w klasie bazowej. Zadeklarujemy jako funkcję wirtualną funkcję `Oddychaj()` w klasie `CZwierzak`:

```
class CZwierzak
{
public:
  void Jedz();
  virtual void Oddychaj();
};
```

Wyobraźmy sobie, że chcemy zdefiniować klasę pochodną `CRybka`. Niby to zwierzak, ale rybki nie oddychają w taki sam sposób jak inne obiekty klasy `CZwierzak`. Funkcję `Oddychaj()` trzeba zatem będzie napisać w dwu różnych wariantach. Obiekt `Szarik` może tę funkcję odziedziczyć bez zmian i sapać spokojnie, z obiektem `Sardynka` gorzej:

```
class CZwierzak
{
public:
  void Jedz();
  virtual void Oddychaj() { cout << "Sapie..."; }
};
```

```
class CPiesek : public CZwierzak
{
  char imie[30];
} Szarik;

class CRybka : public CZwierzak
{
  char imie[30];
public:
  void Oddychaj() { cout << "A ja nie sapie, acha."; }
} Sardynka;
```

W klasie pochodnej w deklaracji funkcji słowo kluczowe `virtual` już nie występuje. W klasie pochodnej funkcja

```
CRybka::Oddychaj()
```

— potrafi więcej niż w przypadku zwykłego overloadingu funkcji. Funkcja `CZwierzak::Oddychaj()` zostaje przesłonięta, mimo że liczba i typ argumentów pozostają bez zmian. Taki proces — bardziej drastyczny niż przeciążanie — nazywany jest **przesłanianiem** lub **nadpisywaniem funkcji** (ang. `function overriding`). W programie przykładowym `Szarik` będzie oddychał, a `Sardynka` nie.

Listing D25.CPP

```
#include <iostream.h>

class CZwierzak
{
public:
  void Jedz();
  virtual void Oddychaj() {cout << "\nSapie...";}
};

class CPiesek : public CZwierzak
{
  char imie[30];
} Szarik;

class CRybka : public CZwierzak
```


programowanie_strukturalne_DODATEK.indb 154 2010-07-05 11:54:55

```
{
  char imie[30];
public:
  void Oddychaj() {cout << "\nSardynka: A ja nie oddycham.";}
} Sardynka;

main()
{
  Szarik.Oddychaj();
  Sardynka.Oddychaj();
  return 0;
}
```

Funkcja `CZwierzak::Oddychaj()` została w obiekcie `Sardynka` przesłonięta przez funkcję `CRybka::Oddychaj()` — nowszą wersję funkcji/metody pochodzącej z klasy pochodnej.

Przeciążenie (overloading) funkcji opierało się na dodatkowych informacjach, które C++ dołącza podczas kompilacji do funkcji, a które dotyczą liczby i typów argumentów danej wersji funkcji. W przypadku funkcji wirtualnych jest inaczej. Aby wykonać przesłanianie kolejnych wersji funkcji wirtualnej w taki sposób, funkcja we wszystkich „pokoleniach" musi mieć taki sam prototyp, tj. pobierać taką samą liczbę parametrów tych samych typów oraz zwracać wartość tego samego typu. Jeśli tak się nie stanie, C++ potraktuje różne prototypy tej samej funkcji w kolejnych pokoleniach zgodnie z zasadami przeciążania funkcji.

W przypadku funkcji wirtualnych o wyborze wersji funkcji decyduje to, wobec którego obiektu (której klasy) funkcja została wywołana. Jeśli wywołamy funkcję dla obiektu `Szarik`, C++ wybierze wersję:

```
CZwierzak::Oddychaj();
```

Wobec obiektu `Sardynka` zostanie zastosowana wersja:

```
CRybka::Oddychaj();
```

W C++ wskaźnik do klasy bazowej może także wskazywać na klasy pochodne, więc zastosowanie funkcji wirtualnych może dać pewne ciekawe efekty. Jeśli zadeklarujemy wskaźnik `*p` do obiektów klasy bazowej `CZwierzak *p`, a następnie zastosujemy ten sam wskaźnik do wskazania obiektu klasy pochodnej:

```
p = &Szarik;     p->Oddychaj();
...
p = &Sardynka;   p->Oddychaj();
```

— zażądamy w taki sposób od C++ rozpoznania właściwej wersji wirtualnej metody `Oddychaj()` i jej wywołania we właściwym momencie. C++ może rozpoznać, którą wersję funkcji należałoby zastosować, tylko na podstawie typu obiektu, wobec którego funkcja została wywołana. I tu pojawia się pewien problem. Kompilator, wykonując kompilację programu, nie wie, co wskaże wskaźnik. Ustawienie wskaźnika na konkretny adres nastąpi dopiero w czasie wykonania programu (*run-time*). Kompilator „wie" zatem tylko tyle:

```
p->Oddychaj()();      //ktora wersja Oddychaj()    ???
```

Aby mieć pewność, co w tym momencie będzie wskazywał wskaźnik, kompilator musiałby wiedzieć, w jaki sposób przebiegnie wykonanie programu. Takie wyrażenie może zostać podane w czasie wykonywania programu dwojako: raz, gdy wskaźnik będzie wskazywał obiekt `Szarik` (inaczej), a drugi raz — obiekt `Sardynka` (inaczej):

```
if( p == &Szarik )
  CZwierzak::Oddychaj();
else
  CRybka::Oddychaj();
```

Taki efekt działania funkcji wirtualnych nazywa się **polimorfizmem uruchomieniowym** (ang. *run-time binding*, *late binding*). Z kolei przeciążenie funkcji i operatorów daje efekt tzw. polimorfizmu kompilacji (ang. *compile-time binding*). Ponieważ wszystkie wersje funkcji wirtualnej mają taki sam prototyp, nie istnieje inna metoda stwierdzenia, którą wersję funkcji należy zastosować. Wyboru właściwej wersji funkcji można dokonać tylko na podstawie typu obiektu, do którego należy wersja funkcji-metody.

Różnica między polimorfizmem przejawiającym się na etapie kompilacji i polimorfizmem przejawiającym się na etapie uruchomienia programu jest nazywana również wczesnym albo późnym polimorfizmem. W przypadku wystąpienia wczesnego polimorfizmu (*compile-time*, *early binding*) C++ wybiera wersję funkcji do zastosowania, już tworząc plik *.OBJ*. W przypadku późnego polimorfizmu C++ wybiera wersję funkcji (poddanej przesłanianiu — *overriding*) do zastosowania po sprawdzeniu bieżącego kontekstu i zgodnie z bieżącym wskazaniem wskaźnika.

Przyjrzyjmy się dokładniej zastosowaniu wskaźników do obiektów w przykładowym programie. Utworzymy hierarchię złożoną z klasy bazowej i pochodnej w taki sposób, by klasa pochodna zawierała jakiś unikalny element, na przykład niewystępującą w klasie bazowej funkcję.

```
class CZwierzak
{
public:
  void Jedz();
  virtual void Oddychaj() {cout << "\nSapie...";}
};
```

```
class CPiesek : public CZwierzak
{
  char imie[20];
  void Szczekaj() { cout << "Szczekam !!!"; }
} Szarik;
```

Jeśli teraz zadeklarujemy wskaźnik do obiektów klasy bazowej, na przykład `CZwierzak *p`, to za pomocą tego wskaźnika możemy odwołać się także do obiektów klasy pochodnej oraz do elementów obiektu klasy pochodnej, na przykład do funkcji:

```
p->Oddychaj();
```

Pojawia się tu jednakże pewien problem. Jak wskazać za pomocą wskaźnika element klasy pochodnej, który nie został odziedziczony i którego nie ma w klasie bazowej? Rozwiązanie jest proste — wystarczy zażądać od C++, by chwilowo zmienił typ wskaźnika z obiektów klasy bazowej na obiekty klasy pochodnej. W przypadku funkcji `Szczekaj()` w naszym programie wyglądałoby to tak:

```
CZwierzak *p;
...
p->Oddychaj();
p->Szczekaj();              //ZLE !
(CPiesek*)p->Szczekaj();    //Poprawnie !
```

Dzięki funkcjom wirtualnym, tworząc klasy bazowe, pozwalamy późniejszym użytkownikom na rozbudowę funkcji-metod w najwłaściwszy ich zdaniem sposób. Dziedzicząc, możemy przejmować z klasy bazowej tylko to, co nam odpowiada.

2.2. Klasy abstrakcyjne

Wolno nam umieścić funkcję wirtualną w klasie bazowej, nie definiując jej wcale. W klasie bazowej umieszczamy tylko deklarację-prototyp funkcji. W następnych pokoleniach klas pochodnych mamy wtedy pełną swobodę i możemy zdefiniować funkcję wirtualną w dowolny sposób, adekwatny do potrzeb danej klasy pochodnej. Możemy na przykład do klasy bazowej dodać prototyp funkcji wirtualnej `funkcja_eksperymentalna()`, nie definiując jej w klasie bazowej. Sens umieszczenia takiej funkcji w klasie bazowej polega na uzyskaniu pewności, iż wszystkie klasy pochodne odziedziczą funkcję `funkcja_eksperymentalna()`, ale każda z klas pochodnych wyposaży tę funkcję we własną definicję.

Takie postępowanie może okazać się szczególnie uzasadnione przy tworzeniu biblioteki klas (ang. *class library*) przeznaczonej dla innych użytkowników. C++ w wersji instalacyjnej ma już kilka gotowych bibliotek klas. Funkcje wirtualne, które nie zostaną zdefiniowane i w związku z tym nie mają ciała funkcji, nazywane są **funkcjami w pełni wirtualnymi** (ang. *pure virtual function*). Jeśli chcemy zadeklarować funkcję

programowanie_strukturalne_DODATEK.indb 157 2010-07-05 11:54:55

`CZwierzak::Oddychaj()` jako funkcję w pełni wirtualną, trzeba tę informację w jakiś sposób przekazać kompilatorowi C++. Nie możemy zadeklarować jej tak:

```
class CZwierzak
{
...
public:
  virtual void Oddychaj();
  ...
};
```

— a następnie pominąć definicję (ciało) funkcji. Takie postępowanie C++ uznałby za błąd, a funkcję za zwykłą funkcję wirtualną, tyle że niedopracowaną przez programistę. Naszą intencję musimy zaznaczyć już w definicji klasy w taki sposób:

```
class CZwierzak
{
...
public:
  virtual void Oddychaj() = 0;   // <-- funkcja w pelni wirtualna
  ...
};
```

Informacją dla kompilatora, że chodzi o funkcję w pełni wirtualną, jest wpis `= 0` po prototypie funkcji. Definiując klasę pochodną, musimy rozbudować funkcję w pełni wirtualną, na przykład:

```
class CZwierzak
{
public:
  virtual void Oddychaj() = 0;
  ...
};
class CPiesek : public CZwierzak
{
public:
  void Oddychaj() { cout << "Oddycham..."; }
  ...
};
```

Przykładem takiej funkcji jest funkcja `Mow()` z przedstawionego poniżej programu. Deklarujemy ją jako w pełni wirtualną, ponieważ różne obiekty klasy `Czlowiek` i klas pochodnych

```
class Czlowiek
{
public:
  void Jedz(void);
  virtual void Mow(void) = 0;    //funkcja w pelni WIRTUALNA
};

class Niemowle : public Czlowiek
{
public:
  void Mow(void);          // Tym razem BEZ slowa virtual
};
/* Tu definiujemy metode wirtualna: -------------------- */
void Niemowle::Mow(void) { cout << "Nie Umiem Mowic!\n"; }
```

mogą mówić na różne sposoby... Obiekt klasy `Niemowle` nie umie mówić wcale, ale z innymi obiektami może być inaczej. Wyobraźmy sobie na przykład obiekt klasy `Zona`. W kolejnym pokoleniu obiektów definicja wirtualnej metody `Mow()` mogłaby wyglądać tak jak na poniższym listingu.

Listing D26.CPP

```
#include <iostream.h>

class Czlowiek
{
public:
  void Jedz(void);
  virtual void Mow(void) = 0;
};

void Czlowiek::Jedz(void) { cout << "MNIAM, MNIAM..." << endl; }

class Zona : public Czlowiek
{
public:
  void Mow(void);           //Zona mowi swoje
};                          //bez wzgledu na argumenty - typ void
```

```
void Zona::Mow(void)
{
  cout << "NIE MAM CO NA SIEBIE WLOZYC!" << endl;
}

class Niemowle : public Czlowiek
{
public:
  void Mow(void);
};

void Niemowle::Mow(void) { cout << "Nie Umiem Mowic! " << endl; };

int main(void)
{
  Niemowle Dziecko;
  Zona Moja_Zona;

  Dziecko.Jedz();
  Dziecko.Mow();
  Moja_Zona.Mow();
  return 0;
}
```

Przykładowa klasa `Czlowiek` jest **klasą abstrakcyjną**. Jeśli spróbujemy dodać do powyższego programu na przykład:

```
Czlowiek Facet;
Facet.Jedz();
```

— uzyskamy komunikat o błędzie: *Cannot create a variable for abstract class „Czlowiek”* (Nie mogę utworzyć zmiennej dla klasy abstrakcyjnej `Czlowiek`).

Od angielskich nazw *Abstract Class* (klasa abstrakcyjna) oraz *Abstract Data Types* (abstrakcyjne typy danych) klasy abstrakcyjne bywają w skrócie nazywane *ABC* lub częściej *ADT*.

programowanie_strukturalne_DODATEK.indb 160 2010-07-05 11:54:55

3

Polimorfizm — przykład zastosowania FIFO i LIFO

Polimorfizm, czyli możliwość uzyskiwania różnej wewnętrznej budowy obiektów i różnych sposobów działania metod, wymaga zastosowania pewnych elementów. Wyjaśnijmy dokładniej ich specyfikę.

3.1. Klasy abstrakcyjne raz jeszcze

Wyobraźmy sobie, że programista pracujący z C++ ma wizję narzędzia pozwalającego na rozwiązanie pewnych problemów i chciałby umieścić wspólne, typowe własności funkcjonalne w pojedynczym module. Następnie mógłby udostępniać taki standardowy moduł innym programistom w celu indywidualnego modyfikowania odpowiednich funkcji i tworzenia końcowych, różniących się szczegółami programów. Właśnie w tym celu wymyślono programowanie obiektowe.

UWAGA

Klasy abstrakcyjne mają następujące właściwości:

- Zawierają co najmniej jedną funkcję w pełni wirtualną w obrębie definicji klasy.
- Mogą być używane tylko do dziedziczenia i tworzenia na ich podstawie klas pochodnych.
- Żadna klasa zawierająca choćby jedną funkcję w pełni wirtualną nie może być użyta do utworzenia obiektu. Obiekt klasy abstrakcyjnej nie zostanie utworzony przez żaden kompilator C++.

Klasa abstrakcyjna to narzędzie programisty, które umożliwia zdefiniowanie podstawowych własności funkcjonalnych (ang. *core functionality*), podczas gdy szczegóły techniczne mogą zostać zdefiniowane później. Klasy abstrakcyjne są wykorzystywane w dziedziczeniu (więcej — nadają się tylko do dziedziczenia).

3.2. Funkcje w pełni i nie w pełni wirtualne

Funkcja (metoda) wirtualna (ang. *virtual method*, *virtual member function*) jest deklarowana w obrębie definicji klasy bazowej. Prototyp metody wirtualnej jest poprzedzony słowem kluczowym `virtual`. Oznacza to, że klasy potomne używają tylko jednej postaci tej funkcji w celu uniknięcia niejednoznaczności.

Klasa potomna może nadpisać (ang. *override*) definicję metody wirtualnej poprzez redefiniowanie tej metody. Nowa definicja będzie używana przez wszystkie obiekty danej klasy potomnej.

Jeśli funkcja-metoda zostanie zadeklarowana w klasie bazowej jako wirtualna, musi pozostać funkcją-metodą wirtualną we wszystkich klasach pochodnych od tej klasy bazowej. W przypadku redefinicji metody wirtualnej w klasie pochodnej można użyć słowa kluczowego `virtual` lub też pominąć to słowo. Poniższe zapisy w klasie pochodnej są sobie równoważne:

`virtual void f1();` jest równoważne: `void f1();`

Funkcja w pełni wirtualna (ang. *pure virtual function — PVF*) to funkcja szczególna. Zadeklarowanie w klasie bazowej funkcji w pełni wirtualnej polega na przyrównaniu jej do zera:

```
virtual void f1( void ) = 0;   // To jest deklaracja PVF
```

Jeśli to zrobimy, to nie wolno nam podawać żadnej definicji dla tej funkcji. Funkcja taka nie robi nic i nic robić nie może, przeszkadza natomiast swojej klasie w utworzeniu choćby jednego obiektu danej klasy. Ma to jednak swoje zalety. Jeśli na podstawie klasy zawierającej taką funkcję metodą dziedziczenia utworzymy jakąś klasę pochodną, funkcja w pełni wirtualna zostanie odziedziczona wraz z innymi komponentami klasy bazowej. Jeśli pozostanie nadal funkcją w pełni wirtualną, nadal nie pozwoli utworzyć żadnego obiektu tejże klasy pochodnej. Jest to zatem metoda zmuszająca programistę, by w klasie pochodnej nadpisał funkcję w pełni wirtualną i opracował dla niej konkretną definicję, ponieważ inaczej nie uda się utworzyć ani jednego obiektu.

Funkcje wirtualne pozwalają zachować metodzie tę samą nazwę, umożliwiając jednocześnie zupełnie różne działanie w stosunku do różnych klas (potomnych) i różnych obiektów, jeśli w klasach potomnych dokonana zostanie ich redefinicja. Cecha ta pozwala na uzyskiwanie różnego działania metody o tej samej nazwie w różnych punktach hierarchii klas. I to właśnie nazywane jest **polimorfizmem** (ang. *polymorphism*). Aby wykazywać polimorfizm, funkcje-metody w C++ muszą być deklarowane jako funkcje wirtualne.

programowanie_strukturalne_DODATEK.indb 162 2010-07-05 11:54:56

3.3. Sekwencje FIFO i LIFO, czyli o specyfikacji algorytmu

W komputerowym żargonie często używa się słów kolejka, *FIFO* i *LIFO*. Chodzi o pewien sposób przechowywania danych w pamięci i metodę manipulowania tymi danymi, co w najbardziej ogólny sposób oddaje właśnie termin „sekwencja" lub „sekwencyjny dostęp do danych".

Przeanalizujemy różne sposoby tworzenia kolejki i określimy ich cechy wspólne. Następnie wspólne cechy umieścimy w abstrakcyjnej klasie bazowej, by wykorzystać je poprzez dziedziczenie do utworzenia własnych dwu klas pochodnych: `Stack` (stos) i `Queue` (kolejka).

Zacznijmy od wyjaśnienia tych skrótów. *FIFO* (ang. *First In, First Out*) oznacza zwyczajną kolejkę, z której wychodzi jako pierwszy i jako pierwszy zostaje obsłużony ten, kto przyszedł najwcześniej. *LIFO* (*Last In, First Out*) oznacza proces odkładania na stos, a później zdejmowania z niego, kiedy jako pierwszy zostaje pobrany z kolejki i obsłużony ten, kto trafił na stos jako ostatni (a ten pierwszy musi czekać aż do końca). Kolejka typu *LIFO* nazywana jest inaczej stosem.

Przetłumaczymy to teraz na język specyfikacji algorytmu. Zacznijmy od specyfikacji dla kolejki *FIFO*:

- Nowo utworzona kolejka *FIFO* jest pusta.
- W kolejce *FIFO* mieści się maksymalnie 9 znaków.
- Nowy znak może być dodany tylko na koniec kolejki *FIFO*.
- Dane mogą być pobierane tylko z początku kolejki *FIFO*.
- Jeśli kolejka jest pusta, drukowany jest komunikat.
- W razie próby dodania do kolejki więcej niż 9 znaków wyprowadzony zostanie komunikat o błędzie.

Opracujmy specyfikację określającą sposób działania obiektu klasy *LIFO* (`Stack`).

Każdy obiekt klasy *LIFO* powinien mieć następujące własności:

- Nowo utworzona kolejka `Stack` jest pusta.
- W kolejce `Stack` mieści się maksymalnie 9 znaków.
- Nowy znak może być dodany tylko na początek kolejki (odłożony na szczyt stosu).
- Dane mogą być pobierane tylko z początku kolejki.
- Jeśli stos jest pusty, drukowany jest komunikat.
- W razie próby dodania do stosu więcej niż 9 znaków wyprowadzony zostanie komunikat o błędzie.

programowanie_strukturalne_DODATEK.indb 163 2010-07-05 11:54:56

Z punktu widzenia programisty ma sens zakodowanie cech wspólnych tylko raz (w obrębie klasy bazowej), a następnie za pomocą dziedziczenia utworzenie dwóch klas potomnych. Dopiero w obrębie tych klas potomnych można kodować indywidualne, specyficzne cechy. Właśnie do takich celów służą klasy abstrakcyjne. Uogólnienie polega na wybraniu cech wspólnych (ang. *common properties*), pozwalających na wspólne traktowanie. Jeśli przyjrzymy się uważnie dwóm powyższym kolejkom, następujące cechy okażą się wspólne (podobne lub identyczne):

1. Nowo utworzona kolejka *LIFO* jest pusta.
   Nowo utworzona kolejka *FIFO* jest pusta.
2. W kolejce *LIFO* mieści się maksymalnie 9 znaków.
   W kolejce *FIFO* mieści się maksymalnie 9 znaków.
3. Jeśli stos *LIFO* jest pusty, drukowany jest komunikat.
   Jeśli kolejka *FIFO* jest pusta, drukowany jest komunikat.
4. Jeśli nastąpi próba dodania więcej niż 9 znaków, wyprowadzony zostanie komunikat o błędzie.

A teraz różnice:

1. Nowy znak może być dodany tylko na początek kolejki *LIFO*.
   Nowy znak może być dodany tylko na koniec kolejki *FIFO*.
2. Dane mogą być pobierane tylko z początku kolejki *LIFO*.
   Dane mogą być pobierane tylko z początku kolejki *FIFO*.

Własności 1, 2, 6 są identyczne i doskonale nadają się do ujęcia ich w abstrakcyjnej klasie bazowej. Przyjrzyjmy się pozycji 3. Nowy znak jest dodawany na szczyt stosu lub na koniec kolejki typu *FIFO*. W praktyce polega to na umieszczeniu kolejnego, nowego znaku w pierwszym wolnym elemencie tablicy znakowej. Różnica polega więc nie na sposobie dokładania do kolejki, a na sposobie usuwania z kolejki. Skoro tak, także tę operację możemy umieścić we wspólnej abstrakcyjnej klasie bazowej.

Mamy 4 wspólne punkty specyfikacji, które możemy zakodować w abstrakcyjnej klasie bazowej. Dla wygody umieścimy klasę bazową w pliku nagłówkowym, a następnie w programie dokonamy dziedziczenia. Najpierw zrobimy to na przykładzie klasy pochodnej `Stack`, a później na przykładzie klasy pochodnej `Queue`. Klasa bazowa będzie nazywać się `Sequence` i będzie umieszczona w pliku nagłówkowym *SEQUENCE.H*.

Listing D27.CPP

```
// Plik naglowkowy SEQUENCE.H

class Sequence
{
protected:
```


programowanie_strukturalne_DODATEK.indb 164 2010-07-05 11:54:56

```
    int back;
    char data[10];

  public:
    virtual void Poke(char ch);
    virtual void Pop(void) = 0;
    virtual void Peek(void) = 0;
    Sequence();
  };

  Sequence::Sequence()
  {
    back = 0;
  }

  void Sequence::Poke(char ch)
  {
    if (back < 9)
    {
      back++;
      data[back] = ch;
      cout << endl;
    }
    else
      cout << endl << "Przykro mi - pelno!" << endl << endl;
  }
```

3.4. Obsługa stosu

Napiszmy teraz krótki program główny `main()`, zawierający dyrektywę dołączającą plik nagłówkowy `#include "SEQUENCE.H"`. Należy pamiętać, by plik *SEQUENCE.H* i plik źródłowy programu **.CPP* znajdowały się w tym samym folderze (alternatywnie można podać w dyrektywie `#include` pełną ścieżkę dostępu). W obrębie programu umieścimy deklarację, na przykład `Sequence Obiekt1;`. Deklaracja jest tak prosta, ponieważ konstruktor jest bezargumentowy. Podczas próby kompilacji takiego programu powinno się uzyskać komunikat o błędzie: *Error: Cannot create instance of abstract class* (Błąd: Nie mogę utworzyć obiektu klasy abstrakcyjnej) lub podobny, zależnie od wersji C++. Jeśli komunikat się pojawi, wszystko jest w porządku.

Kod wykorzystuje tablicę znakową oraz metodę, która określa, gdzie znajduje się koniec kolejki. Koniec ten może być zarówno ostatnim elementem w kolejce *FIFO*, jak i szczytowym (a więc ostatnim) elementem na szczycie stosu. Odpowiednie metody nazwano tu:

- `Poke()` — odłóż na stos lub dodaj do końca kolejki.
- `Pop()` — zdejmij (domyślnie: ze stosu *LIFO*).
- `Peek()` — pobierz (domyślnie: z kolejki *FIFO*).

Oprócz tego jest jeszcze bezargumentowy konstruktor `Sequence()`, który ma za zadanie zainicjować pustą kolejkę (tj. technicznie — tablicę znakową). Metoda `Poke()` może zostać zdefiniowana raz w klasie bazowej, a następnie jedynie po prostu odziedziczona. Metody `Pop()` i `Peek()` będą w każdej z klas potomnych działać inaczej, więc w klasie bazowej możemy je zadeklarować jako funkcje w pełni wirtualne, a zdefiniować indywidualnie dopiero w klasach potomnych.

Zadanie polega na dokonaniu dziedziczenia klasy `Stack` od klasy bazowej `Sequence` i dodaniu definicji dla wspomnianych wyżej funkcji w pełni wirtualnych. Jeśli zapomnimy choćby o jednej, obiektu nie będzie. Dopiero zdefiniowanie metod `Pop()` oraz `Peek()` pozwoli utworzyć bez protestów kompilatora obiekt klasy `Stack`.

3.4.1. Dziedziczenie klasy pochodnej Stack

W pierwszym wierszu definicji za pomocą polecenia `public Sequence` wskazujemy klasę bazową i sposób dziedziczenia. Jeśli po deklaracjach metod `Pop()` i `Peek()` nie umieścimy przyrównania do zera, każdy kompilator C++ będzie spodziewać się ich definicji. Skoro nie pozostaną w pełni wirtualne, a przeciwnie — staną się w tym pokoleniu normalnymi, ściśle zdefiniowanymi funkcjami, można będzie utworzyć obiekty klasy `Stack`.

```
class Stack : public Sequence
{
public:
  Stack();               // bezargumentowy konstruktor
  void Pop( void );      // dwie metody przeznaczone do zdefiniowania
  void Peek( void );
};
```

3.4.2. Zdejmowanie bajtów ze stosu

Skoro już dokonaliśmy deklaracji, musimy zdefiniować funkcję `Pop()`. To jest ten moment, gdy następuje przystosowanie klasy abstrakcyjnej do konkretnych, indywidualnych potrzeb. Fachowcy nazywają takie działanie **adaptacją** (*customizing*).

Aby usunąć znak ze szczytu stosu, musimy spowodować, by ten szczyt przesunął się o jeden element tablicy w dół. Co wskaże nam, gdzie znajduje się szczyt stosu (czyli

który element został zapisany w tablicy jako ostatni i jest ostatnim niepustym)? Posłużymy się w tym celu licznikiem elementów tablicy, który w tym przykładzie został nazwany `back`. Metoda powinna być przygotowana na dwie możliwe sytuacje. Jeśli na stosie są znaki (są wykorzystane elementy tablicy, licznik jest większy od zera), należy cofnąć licznik. W przeciwnym razie licznik pozostaje równy zeru, a metoda jedynie wyprowadza na ekran komunikat, że stos jest pusty i nie można niczego zdjąć.

```
void Stack::Pop( void )
{
  if ( back > 0 )
  {
    back--;
  }
  else
  {
    cout << endl << "STOS JEST PUSTY!";
    cout << endl << endl;
  }
}
```

3.4.3. Podglądanie bajtów na szczycie stosu

Odziedziczone z klasy bazowej pole danych `back` wskazuje ostatni element tablicy, czyli bajt na szczycie stosu. Sytuacja, gdy stos jest pusty, jest nam sygnalizowana przez stan `back == 0` (taki sposób sygnalizowania stanu fachowcy nazywają **flagowaniem** — ang. *flagged*). Sprawdzimy, czy stos nie jest pusty. Jeśli tak, wyprowadzimy komunikat. Jeśli nie, odczytamy, co jest na szczycie stosu: `data[ back ]`.

```
void Stack::Peek( void )
{
  if ( back == 0 )
  {
    cout << endl << "STOS JEST PUSTY!";
    cout << endl << endl;
  }
  else
  {
    cout << endl << data[ back ] << endl << endl;
  }
}
```


programowanie_strukturalne_DODATEK.indb 167 2010-07-05 11:54:56

3.4.4. Konstruktor klasy Stack a konstruktor klasy bazowej

Konstruktor klasy `Stack()` wywołuje jedynie konstruktor klasy bazowej `Sequence()`, i to bez przekazywania parametrów, ponieważ obydwa są bezargumentowe. Konstruktor klasy bazowej ma za zadanie wyzerować pola danych, co oznacza, że kolejka jest pusta. Stos został utworzony i jest pusty. Skoro już istnieje, może swą obecność zaanonsować:

```
Stack::Stack() : Sequence()
{
  cout << "STOS UTWORZONY I PUSTY." << endl << endl;
}
```

Listing poniżej demonstruje implementację klasy `Stack()` na podstawie dziedziczenia od abstrakcyjnej klasy bazowej `Sequence()`.

Listing D28.CPP

```
#include <iostream.h>
#include <stdlib.h>
#include "SEQUENCE.H"   // UWAGA: Koniecznie dolaczmy nasz plik

class Stack : public Sequence
{
public:
  Stack();
  void Pop(void);
  void Peek(void);
};

void Stack::Pop(void)
{
  if (back > 0)
  {
    back--;
  }
  else
  {
    cout << endl << "STOS PUSTY!";
```

```
    cout << endl << endl;
  }
}

void Stack::Peek(void)
{
  if (back == 0)
  {
    cout << endl << "STOS PUSTY!";
    cout << endl << endl;
  }
  else
  {
    cout << endl << data[back];
    cout << endl << endl;
  }
}

Stack::Stack() : Sequence()
{
  cout << "STOS UTWORZONY." << endl << endl;
}

char menu(void);

main()
{
  char ch;
  char moby;

  Stack S;
  while (1)
  {
    ch = menu();
    switch(ch)
```

```
    {
      case '1' :
        cout << "Podaj znak: ";
        cin >> moby;
        S.Poke(moby);
      break;

      case '2' :
        S.Pop();
      break;

      case '3' :
        S.Peek();
      break;
      case '4' :
        exit(0);
    }
  }
  return(0);
}

char menu(void)

{
  char choice;

  cout << "1...Dodaj znak" << endl;          // Dodaj do stosu
  cout << "2...Zdejmij znak" << endl;        // Usun ze stosu
  cout << "3...Pokaz co masz" << endl;       // Pokaz wierzcholek stosu
  cout << "4...Koniec" << endl << endl;      // Koniec programu
  cout << "Wybierz: ";                       // Wybierz: 1,2,3 lub 4
  cin >> choice;                             // choice – to wybor
  return (choice);
}
```

3.5. Kolejka

Na podstawie abstrakcyjnej klasy bazowej `Sequence` utworzymy klasę pochodną `Queue`. Wystarczy przeprowadzić dziedziczenie i zdefiniować metody wirtualne `Pop()` i `Peek()` oraz własny konstruktor.

```
Class Queue : public Sequence
{
public:
  Queue();            // bezargumentowy konstruktor
  void Pop( void );   // dwie metody przeznaczone do zdefiniowania
  void Peek( void );
};
```

Jedną z dwu istotnych różnic między kolejką typu *LIFO* (`Stack`) a *FIFO* jest sposób, w jaki dane opuszczają kolejkę. W przypadku stosu po prostu zapominaliśmy o istnieniu ostatniego elementu, dekrementując licznik. Reszta zawartości tablicy pozostawała bez zmian. Przy okazji warto tu zwrócić uwagę na pewną osobliwość. Otóż po takim cofnięciu licznika, czyli, innymi słowy, usunięciu wskazania, fizyczna zawartość ostatniego elementu tablicy (bajtu pamięci) nie ulegała zmianie, a jednak z punktu widzenia programu przestawał on istnieć. Miejsce w pamięci było od tego momentu traktowane jako wolne.

W kolejce *FIFO* jest inaczej. Usuwany z kolejki element tablicy rzeczywiście zmienia swoją fizyczną lokalizację w pamięci RAM (i jeśli nie będzie buforującego elementu tablicy, element taki fizycznie zniknie z pamięci). Usuwamy tam pierwszy element tablicy, a wszystkie pozostałe elementy przesuwają się w tablicy w dół o jedno miejsce.

```
void Queue::Pop( void )
{
  int index;
  if ( back > 0 )
  {
    for ( index = 1; index < back; index++ )
    {
      data[ index ] = data [ index + 1 ];
    }
    back--;
  }
  else
  {
```

```
      cout << endl << "Kolejka jest PUSTA!";
      cout << endl << endl;
    }
  }
```

3.5.1. Przeglądanie zawartości kolejki

Druga istotna różnica między stosem a kolejką polega na tym, że na stosie mogliśmy podejrzeć zawsze tylko ostatni element znajdujący się na szczycie stosu. W kolejce możemy przeglądać całą zawartość. Oto odpowiedni kod definicji metody `Peek()`, który ma stanowić redefinicję metody w pełni wirtualnej, odziedziczonej z klasy bazowej.

```
void Queue::Peek( void )
{
  int x;
  if ( back == 0 )
  {
    cout << endl << "Kolejka - PUSTA.";
    cout << endl << endl;
  }
  else
  {
    for ( x = 1; x <= back; x++ )
    {
      cout << endl << data[ x ] << TAB ;
    }
    cout << endl << endl;
  }
}
```

Podobnie jak w przypadku klasy `Stack`, konstruktor ma za zadanie tylko wywołanie konstruktora klasy bazowej `Sequence`. I tym razem następuje zainicjowanie pustej kolejki i komunikat o utworzeniu nowego obiektu:

```
Queue:: Queue() : Sequence()
{
  cout << "Utworzylem nowy obiekt Queue" << endl << endl;
}
```

UWAGA

Poniżej zestawiono informacje, które warto zapamiętać.

- Klasa abstrakcyjna jest narzędziem służącym do projektowania, pozwalającym na zdefiniowanie podstawowych cech funkcjonalnych, wspólnych dla wielu klas obiektów. Specyficzne, charakterystyczne cechy można pozostawić do zdefiniowania później.
- Każda klasa abstrakcyjna ma w obrębie swojej definicji co najmniej jedną metodę, będącą funkcją w pełni wirtualną.
- Klasy abstrakcyjne mogą być wykorzystywane wyłącznie w procesie dziedziczenia do tworzenia nieabstrakcyjnych klas pochodnych.
- Żadna klasa zawierająca choćby jedną metodę w pełni wirtualną nie może być wykorzystana do utworzenia obiektu danej klasy.
- W klasie pochodnej musimy zdefiniować funkcję w pełni wirtualną, przydzielając jej konkretne działanie. Tylko wtedy możliwe będzie utworzenie obiektu takiej klasy pochodnej.
- Prototyp funkcji wirtualnej w definicji klasy bazowej musi być poprzedzony słowem kluczowym *virtual*.
- Klasy pochodne posługują się tylko jedną, wspólną wersją metody wirtualnej, unikając w ten sposób niejednoznaczności. Ta pojedyncza definicja takiej funkcji wirtualnej jest używana przez wszystkie klasy wchodzące w skład drzewa hierarchicznego w schemacie dziedziczenia.
- Prototyp funkcji w pełni wirtualnej jest przyrównywany do zera: `virtual void f1() = 0;` i nie wolno dla takiej funkcji dodać jej definicji w danej klasie (stanowiącej abstrakcyjną klasę bazową).
- Jeśli funkcja została zadeklarowana jako wirtualna w klasie bazowej, pozostaje ona metodą wirtualną we wszystkich klasach pochodnych. W klasach pochodnych można pominąć słowo kluczowe `virtual`.
- Polimorfizm to zdolność do posiadania przez metodę o takiej samej nazwie różnych definicji i różnych sposobów działania wobec różnych klas i obiektów wchodzących w skład hierarchicznego drzewa dziedziczenia. Dla każdej z klas pochodnych (a wobec tego i dla obiektów tychże klas) można zdefiniować inny, najbardziej stosowny sposób działania metody polimorficznej.

programowanie_strukturalne_DODATEK.indb 173 2010-07-05 11:54:56

4

Przykład dziedziczenia od klasy bazowej ios

Klasa bazowa `ios` posłużyła do utworzenia kilku klas pochodnych. Obiekty tych klas i zmienne klasy `ios` są wygodnym narzędziem do obsługi operacji wejścia/wyjścia. Obiekty-strumienie wejścia/wyjścia `cin`, `cout` są wygodnymi narzędziami programisty. Czasem jednakże niezbędne bywa, zwłaszcza przy bardziej złożonych operacjach numerycznych, dodatkowe formatowanie danych wejściowych i wyjściowych. Mamy do dyspozycji trzy sposoby formatowania danych:

- funkcje-metody formatujące (ang. *formatting member functions*),
- flagi-znaczniki formatu (ang. *formatting flags*),
- manipulatory (ang. *manipulators*).

4.1. Funkcje-metody formatujące

Klasa obiektów `ios` (*Input/Output Streams* — strumienie We/Wy) zawiera metody umożliwiające formatowanie danych. Podobnie jak w przypadku funkcji `printf()` w klasycznym C operuje się tu pojęciem pola (ang. *field*) danych. Szerokość (`width`) tegoż pola może być kontrolowana (sprawdzana) i ustawiana za pomocą funkcji-metody, której nazwa jest dość łatwa do przewidzenia i odgadnięcia:

```
ios::width()
```

Ponieważ funkcja należy do klasy `ios`, należy odwoływać się do niej, stosując operator widoczności `::` (*scope*). Funkcja jest dwuwariantowa dzięki zastosowaniu techniki przeciążania (overloadingu). Wywołana bez parametru, zwraca bieżącą szerokość pola:

```
int ios::width();        // <-- globalnie, w stosunku do klasy ios
cin.width(...);          // <-- lokalnie, w odniesieniu do obiektu cin
```

```
cout.width(...);
```

Wywołana z parametrem, ustawia nową szerokość pola:

```
int ios::width(int);
```

Sposoby zastosowania tej metody są zwykle następujące:

- na wejściu — ustawia maksymalną liczbę znaków, które mogą zostać wczytane;
- na wyjściu — ustawia minimalną szerokość pola.

Jeśli bieżąca szerokość pola jest mniejsza niż wyspecyfikowana, pole zostaje uzupełnione do zadanej szerokości za pomocą znaków wypełniających (*fill characters*). Jeśli szerokość pola wyjściowego jest większa od wyspecyfikowanej, wartość wyspecyfikowana przez metodę `width()` zostaje zignorowana.

Domyślna szerokość pola wynosi 0. Oznacza to, że pole wejściowe nie będzie ani obcinane (*truncated*), ani uzupełniane (*padded*). Szerokość pola jest ustawiana na 0 po każdej operacji wstawiania do strumienia (ang. *insertion*). Oznacza to w praktyce, że każda operacja strumieniowa wymaga odrębnego ustawiania szerokości pola. Podobny skutek daje zastosowanie manipulatora `setw` (*set width*).

Oto przykłady zastosowania metody `ios::width` na wejściu w celu ograniczenia liczby wczytywanych znaków:

```
const int MAX_LITER = 12;
char imie[MAX_LITER];
cout << "Podaj imie: " << "(maks. " << MAX_LITER - 1 << " znakow: ";
cin.width(MAX_LITER - 1);
cin >> imie;
```

Zachowanie formatu ASCIIZ (dwunasty znak zastrzeżony dla wartownika `'\0'`) wymaga podania użytkownikowi liczby znaków zmniejszonej o 1.

Zastosowanie tejże funkcji-metody na wyjściu może na przykład spowodować wyrównywanie znaków w obrębie pola (*right/left justify*):

```
int SZER_POLA = 10;
int X = 1867, Y = 20041, Z=18;
cout.width(SZER_POLA);
cout << X << '\t';
cout.width(SZER_POLA);
cout << Y << '\t';
cout << Z;        // <-- tu szerokosc pola = default, 0
```

Aby kontrolować ewentualne uzupełnianie pola do zadanej szerokości przez wybrane znaki alfanumeryczne, można zastosować metodę `fill()` działającą na podobnej zasadzie, co opisana powyżej metoda `width()`.

programowanie_strukturalne_DODATEK.indb 175 2010-07-05 11:54:57

```
char ios::fill();       // <-- zwraca biezacy znak wypelniajacy
char ios::fill(char);   // <-- zwraca biezacy znak i ustawia nowy
```

Domyślnym znakiem wypełniającym jest *spacja*. Podobny efekt można uzyskać także za pomocą manipulatora `setfill`. Oto przykład uzupełniania pola zerami:

```
const int SZER_POLA = 8;
const char WYPEL_ZNAK = '0';
int X = 1234, Y = 1234567;
cout.fill( WYPEL_ZNAK );

cout.width( SZER_POLA );
cout << X << '\t';

cout.width( SZER_POLA );
cout << Y << '\t';
```

4.2. Manipulowanie danymi numerycznymi

Do określania precyzji danych numerycznych zmiennoprzecinkowych służy metoda o łatwej do przewidzenia nazwie `precision()`. W przypadku wywołania bezargumentowego tradycyjnie zwraca bieżącą precyzję stosowaną w strumieniach We/Wy. Wywołana z argumentem, ustawia precyzję (wewnętrzną zmienną obiektu) i dodatkowo zwraca poprzednio stosowaną wartość:

```
int ios::precision();
int ios::precision(int);
```

Jej użycie ma sens, jeśli domyślna wartość precyzji liczb zmiennoprzecinkowych stosowana w strumieniach We/Wy nie jest w danym przypadku najodpowiedniejsza.

Domyślna precyzja (ang. *default precision*) to sześć cyfr dziesiętnych. Jeśli ustawione zostaną flagi: `scientific` (format wykładniczy) lub `fixed` (ustalone), funkcja-metoda `precision()` kontroluje tylko liczbę cyfr znaczących (ang. *significant digits*) po przecinku. Jeśli żadna z wymienionych powyżej flag nie zostanie ustawiona, funkcja-metoda `precision()` kontroluje całkowitą liczbę cyfr znaczących — i przed przecinkiem, i po nim. Podobny efekt można uzyskać za pomocą manipulatora `setprecision`.

Zastosowanie tej metody jest proste:

```
float X = 1234.123567;
double Y = 22.1234567890;
cout.precision(4);
cout << Y << '\t';      // <-- 4 cyfry, plus  zaokraglenie
```


programowanie_strukturalne_DODATEK.indb 176 2010-07-05 11:54:57

```
cout << X << '\t';        // <-- 4 cyfry i zaokraglenie

/* po ustawieniu flagi fixed zmienia sie sposob formatowania: */

cout.setf( ios::fixed, ios::floatfield );
cout << X << '\t';                    // <-- wyprowadzi liczbe z zadana precyzja
```

Flagi formatowania (ang. *formatting flags*) mają tu decydujący wpływ na sposób reprezentacji danych numerycznych. Z punktu widzenia programisty warto wiedzieć, że flagi te stanowią pola bitowe w obrębie jednej zmiennej typu `long`. Mogą zatem być w zasadzie „składane” (ang. *OR-ing*). Są jednak od tej zasady wyjątki. Na przykład flagi wyrównywania `right`, `left`, `internal` wzajemnie się wykluczają (patrz niżej), podobnie flagi `dec`, `hex` i rzadziej stosowana `oct` (ósemkowo).

```
long ios::flags();        // zwraca biezace ustawienia flag
long ios::flags(long)     // ustawia nowe wartosci flag, zwraca poprzednie ustawienia
```

Najwygodniej w tym celu posługiwać się parą funkcji-metod: `setf()` / `unsetf()` (*SET Flag* — ustaw flagę, *UNSET Flag* — skasuj).

```
long ios::setf(long)       // <-- w argumencie long musza byc ustawione
                           // odpowiednie bity flag
long ios::unsetf(long)     // <-- bity ustawione na 1 - kasuja flagi
```

W przykładzie powyżej i w praktyce programowania częściej stosuje się drugą wersję wymienionej metody:

```
long ios::setf(long, long)
```

W tym przypadku ustawianie bitów flag jest trochę bardziej złożone. Te bity flag, które mają stan 1 w drugim parametrze, zostają ustawione tak, jak wynika to z wartości odpowiednich bitów pierwszego parametru. Działanie poszczególnych flag jest następujące:

- `ios::left` — wyrównywanie do lewej w polu wyjściowym,
- `ios::right` — wyrównywanie do prawej w polu wyjściowym,
- `ios::internal` — znak „minus” do lewej, cyfry do prawej,
- `ios::skipws` — ignoruj spacje na wejściu (dokładniej *ws = white space* — znaki niewidoczne),
- `ios::dec` — liczby wyprowadzaj dziesiętnie (*DECimal*),
- `ios::hex` — liczby wyprowadzaj szesnastkowo,
- `ios::showbase` — w przypadku liczb szesnastkowych dodaj przedrostek `0x` na wyjściu,
- `ios::uppercase` — duże litery; liczby szesnastkowe: A – F, a nie a – f,
- `ios::scientific` — liczby w postaci wykładniczej, na przykład 1.23E3,

programowanie_strukturalne_DODATEK.indb 177 2010-07-05 11:54:57

- `ios::fixed` — ustalone miejsce przecinka, na przykład 123.123,
- `ios::stdio` — bufory `stdout` i `stderr` są zerowane po każdej operacji wstawienia do strumienia.

Abyśmy mogli przystąpić do stosowania funkcji `setf(long, long)`, należy wyjaśnić jeszcze rolę kilku stałych, które występują jako drugi parametr funkcji:

`ios::basefield` — trzeba zastosować jako drugi parametr, jeśli zmieniamy dec <--> hex <--> oct (podstawę);

`ios::adjustfield` — jeśli ustawiamy wyrównywanie (np.: left);

`ios::floatfield` — jeśli operujemy danymi zmiennoprzecinkowymi.

Oto kilka przykładów praktycznego zastosowania flag formatujących.

Listing D29.CPP

```
#include <iostream.h>
#include <limits.h>
#include <conio.h>

int main()
{
  int i, X=1234;
  float F=1.2345678E6;
  double D=3.0E9;
  clrscr();
  cout << "Demo formatowania ios: " << endl;
  cout << "Standard:\n F (float), D (double), X (int):\n";
  cout << F << '\t' << D << '\t' << X << endl;
  cout << "precision(4):" << endl;
  cout.precision( 4 );
  cout << F << '\t' << D << '\t' << X << endl;
  cout << "Wpisz liczbe int poprzedzajac 4 SPACJAMI." << endl;
  cin >> i;
  cout << "Wpisales: " << i << endl;
  cout << "Wpisz poprawnie - lub nie liczbe typu int: " << endl;
  cin.ignore(INT_MAX, '\n');
  cin.unsetf(ios::skipws);
  cin >> i;
  cin.good() ? (cout << "\nOK: i = " << i << endl)
```

```
      : (cout << "Nieprawidlowa wartosc!" << endl);
    long StareFlagi = cout.flags();
    cout << "128 to szesnastkowo: ";
    cout.setf(ios::showbase | ios::hex);
    cout << 128 << endl;
    cout << "Dzialanie flagi 'fixed': " << endl;
    cout << "Zwroc uwage na niedokladnosc! \n";
    cout.setf(ios::fixed, ios::floatfield);
    cout << "F=" << F << "  <-- TUUU!" << '\t';
    cout << "D=" << D << endl;
    cout << "KONIEC TESTOWANIA...dowolny klawisz...";
    getch();
    return 0;
  }
```

Wprowadzając różne liczby, możemy za pomocą tego programu w wygodny sposób przetestować różne sposoby formatowania wejścia/wyjścia. Przykładowy wydruk podczas pracy programu może wyglądać np. tak:

```
Demo formatowania ios:
Standard:
 F (float), D (double), X (int):
1234567.75  3e+09  1234
precision(4):
1.2346e+06  3e+09  1234
Wpisz liczbe int poprzedzajac 4 SPACJAMI.
    123
Wpisales: 123
Wpisz poprawnie - lub nie liczbe typu int:
123

OK: i = 123
128 to szesnastkowo: 0x80
Dzialanie flagi 'fixed':
Zwroc uwage na niedokladnosc!
F=1234567.75  <-- TUUU! D=3000000000
KONIEC TESTOWANIA...dowolny klawisz...
```


programowanie_strukturalne_DODATEK.indb 179 2010-07-05 11:54:57

4.3. Manipulatory

Proste manipulatory różnią się w swoim działaniu od opisanych wyżej metod i flag tym, że zmiany wprowadzone za ich pomocą mają charakter nie jednorazowy, lecz ciągły i pozostają w mocy aż do ich anulowania (za wyjątkiem manipulatora `setw`). Oto zestawienie najczęściej stosowanych operatorów wraz z krótkim opisem ich działania.

- `endl` (*end of line character*, `\n`) — wstawia do strumienia znak nowego wiersza i wywołuje manipulator `flush`;
- `ends` (*end of string*, `\0`) — koniec łańcucha znakowego;
- `flush` — forsuje wyprowadzenie całego bufora wyjściowego na urządzenie wyjścia i czyści ten bufor;
- `dec` (*decimal*) — konwersja na liczby dziesiętne;
- `oct`, `hex` — konwersja odpowiednio na liczby ósemkowe i szesnastkowe.

Oto prosty przykład zastosowania i działania manipulatorów.

Listing D30.CPP

```
#include <iostream.h>

#include <iomanip.h>

#include <conio.h>


int main()

{

  int i;

  clrscr();

  cout << "Demo manipulatorow ios: " << endl;

  cout << "Podaj liczbe typu int: ";

  cin >> i;

  if (!cin )

  {

    cout << "Cos tu nie tak!" << endl;

    cout << "Jeszcze raz, ale teraz poprawnie prosze: ...";

  }

  else

  {

    cout << "Podales: " << dec << i       << " dziesietnie" << endl;

    cout << "To jest HEX: " << hex << i << endl;
```

```
    cout << "A osemkowo: " << oct << i << endl;
  }
  cout << "...dowolny klawisz...";
  getch();
  return 0;
}
```

Dialog z programem może wyglądać na przykład tak:

```
Demo manipulatorow ios:
Podaj liczbe typu int: 175
Podales: 175 dziesietnie
To jest HEX: af
A osemkowo: 257
...dowolny klawisz...
```

Dość interesujące może być w praktyce zastosowanie obiektu klasy `ostrstream` (buforowany strumień wyjścia z możliwością wykorzystania tablic). W takim buforze można „składać" komunikaty przed ich wyprowadzeniem do strumienia wyjściowego. Kolejny listing przedstawia prosty przykład takiego zastosowania strumienia wyjściowego.

Listing D31.CPP

```
#include <strstream.h>
#include <time.h>
#include <conio.h>

int main()
{
  char Bufor[80] = "Podam Ci date i czas: ";
  time_t czas;

  clrscr();
  ostrstream obiekt( Bufor, sizeof(Bufor),
    ios::out | ios::app );
  time( &czas );
  obiekt << ctime( &czas ) << ends;
  cout << "Wyprowadzam wynik z bufora: " << endl;
  cout << obiekt.str();
```

```
    cout << "\n\n" << "...dowolny klawisz...";
    getch();
    return 0;
  }
```

Dialog z programem przykładowym powinien wyglądać tak:

```
Wyprowadzam wynik z bufora:
Podam Ci date i czas: Mon Oct 11 03:04:31 2008


...dowolny klawisz...
```

Konstruktor klasy `ostrstream` tworzy obiekt składający w buforze informacje przeznaczone do wyprowadzenia. Na uwagę zasługują tu niektóre szczególne elementy:

```
ios::out | ios::app
```

— jest logicznym sumowaniem flag (*OR-ing*) określających tryb pracy. Flagi `out` (*open for output* — otwarta możliwość zapisu) oraz `app` (*append* — dodawanie/dołączanie) powodują, że łańcuchy tekstowe zostają połączone, a na ich końcu dodany zostaje znak *end of string* (`\0`):

```
obiekt << ctime( &czas ) << ends;
```

Metoda `str()` obiektu wyprowadza sumaryczny łańcuch znaków do strumienia `cout`:

```
cout << "Wyprowadzam wynik z bufora: " << endl;
cout << obiekt.str();
```

Jak widać z przykładów, zestaw metod, flag i manipulatorów pozwala w C++ bardzo skutecznie manipulować danymi wejściowymi i wyjściowymi.

5 Operacje dyskowe obiektowo

5.1. Obiektowe operacje na plikach

Obiekty strumieniowe oparte na klasie bazowej `ios` mogą obsługiwać nie tylko strumienie danych przepływających do i od standardowych urządzeń We/Wy. Mogą nam one posłużyć także do wykonywania operacji przesyłania do pliku (ang. *Output File Stream* — `ofstream`) i z pliku (ang. *Input File Stream* — `ifstream`). Rozpoczniemy od przytoczenia interesującego przykładowego programu, który posługując się klasą `ofstream`:

- zakłada w bieżącym katalogu plik dyskowy *DANE.TST*;
- otwiera plik dla zapisu;
- zapisuje do pliku tekst `"to jest zawartosc pliku"`;
- zamyka plik.

Listing D32.CPP

```
#include <fstream.h>
main()
{
  ofstream plik("dane.tst");
  plik << "To jest zawartosc pliku";
  return 0;
}
```

I już. O wszystkie szczegóły techniczne tych (wcale przecież nie najprostszych) operacji zadbał producent w bibliotekach klas wejścia/wyjścia. Konstruktor tworzy obiekt `plik`. Operator << wyprowadza strumień do pliku dyskowego. Jeśli zechcemy do pliku dopisać coś jeszcze, wystarczy dodać:

```
#include <fstream.h>
```

```
void main()
{
  ofstream plik("dane.tst");
  plik << "To jest zawartosc pliku" << " i jeszcze cos.";
}
```

Urzekająca prostota, nieprawdaż? Niedowiarek mógłby w tym momencie zapytać: „A jeśli plik już istnieje, chyba nie jest to takie proste?”. Rzeczywiście, należałoby rozbudować program w C++ do postaci:

```
#include <fstream.h>
main()
{
  ofstream plik("dane.tst", ios::app);
  plik << " Dopiszemy do pliku jeszcze i to...";
  return 0;
}
```

Korzystamy tu dodatkowo z globalnej zmiennej `ios::app`, określającej inny niż typowy tryb dostępu do pliku dyskowego, i w dalszym ciągu z operatora <<. Jeśli chcemy ręcznie obsługiwać szczegółowe operacje plikowe, możemy posługiwać się predefiniowanymi metodami obiektów strumieniowych i flagami zaczerpniętymi z klasy bazowej `ios`, na przykład:

```
ifstream plik("PLIK.DAT", ios::in | ios::binary);       // <-- otwieranie w trybie binarnym
ofstream plik("A:\\plik0.bin", ios::out | ios::binary); // <-- binarnie, do zapisu


ifstream plik("C:\\PLIK.BIN", ios::in | ios::nocreate); // <-- bez zastepowania
if ( !plik )
  cout << "Plik juz istnieje." << endl;


/*  metody, otwarcie i zamkniecie  /*
plik.open("C:\\PLIK.BIN", ios::in | ios::nocreate);
plik.close();
```

Korzystanie z predefiniowanych konstruktorów jest o tyle wygodne, że w chwili gdy obiekt przestaje być widoczny, na przykład gdy wychodzimy poza blok lub funkcję, w obrębie której został utworzony, destruktor automatycznie zamyka strumień plikowy. Konstruktor i destruktor ustawiają także domyślne flagi (domyślny tryb to tryb tekstowy, a nie binarny) i obsługują buforowanie We/Wy. W przykładzie w następnym

podrozdziale, poświęconym obsłudze obiektowej bazy danych, postaram się udowodnić, że nawet podczas przesyłania różnych typów danych domyślne konstruktory i destruktory tych obiektów strumieniowych radzą sobie zupełnie nieźle i często nie ma potrzeby „ręcznej ingerencji”.

5.2. Obiektowe bazy danych

Obsługa baz danych to jeden z najszybciej rozwijających się obszarów zastosowań oprogramowania. Znakomita większość stosowanych współcześnie baz danych i systemów obsługi tychże baz danych to tzw. systemy *RDBMS* i *OODBMS*. Skrótowce te oznaczają odpowiednio:

- RDBMS (ang. *Relational DataBase Management System*) — system zarządzania relacyjną bazą danych,
- OODBMS (ang. *Object Oriented DataBase Management System*) — system zarządzania obiektową bazą danych.

5.2.1. Projekt bazy danych

Aby zaprezentować klarowny i nie nazbyt złożony przykład, liczbę pól rekordu i tym samym liczbę kolumn tabeli ograniczyłem do 3. Aby przykład był zrozumiały intuicyjnie, wyobraźmy sobie, że chcemy w pliku dyskowym zapamiętywać listę dłużników i wierzycieli. Robimy to po to, by wiedzieć, jak wygląda nasz prywatny bilans.

Przyjmiemy, że informację „Adam winien jest mi 100 zł” zapisujemy jako „+100”. Z kolei informację „Ja jestem winien Barbarze 50 zł”, zapiszemy jako „–50”. Teraz możemy przystąpić do projektowania tabeli bazy danych. Gdybyśmy robili to ołówkiem na papierze, zapewne narysowalibyśmy tabelkę z kolumnami (pola) i wierszami (rekordy). Tabela mogłaby wyglądać na przykład tak:

Lp	**IMIĘ**	**SUMA**
1	Adam	+100
2	Barbara	–50,20
3	Krzysztof	+320
4	Mietek	–78,50
...	...	...
`int Lp;`	`char Imie[10];`	`float Sum;`

Tabela ta ma 3 kolumny (to będą pola rekordów) oraz 4 wiersze (to będą rekordy). W C++ pojedynczy wiersz tabeli możemy zdefiniować jako strukturę lub obiekt (tu będzie to obiekt). Nazwijmy tę strukturę `Dlug`:

```
struct Dlug
```


programowanie_strukturalne_DODATEK.indb 185 2010-07-05 11:54:57

```
{
  int Lp;              // = 0x0002
  char Imie[10];       // = "Barbara\0"
  float Sum;           // = -50.200000
};
```

Ale to dopiero pojedynczy wiersz tabeli. Tabela taka jest w istocie listą, czyli ponumerowaną jednowymiarową tablicą składającą się z obiektów tego samego typu — rekordów. Możemy to w C++ zapisać tak:

```
struct Dlug
{
  int Lp;
  char Imie[10];
  float Sum;
};

Dlug Tablica[...];
```

Dla uproszczenia i skrócenia zapisu przyjmijmy, że tabela będzie mieć jakąś założoną w programie maksymalną wielkość `MAX`. Zapiszemy to tak:

```
#define MAX 7

class Dlug
{
public:
  int Lp;
  char Imie[10];
  float Sum;
};

Dlug R[MAX];
```

Pamiętając, że zadeklarowanie tablicy wymaga zastanowienia nad wywoływaniem domyślnego konstruktora obiektów, możemy rozbudować klasę następująco:

```
class Dlug
{
public:
  int Lp;
```

```
    char Imie[10];
    float Sum;
    Dlug(void);          // <-- konstruktor bezparametrowy
    void Print();        // <-- metoda „wydrukuj rekord”
    void Update();       // <-- metoda „zmien zawartość rekordu”
  } R[MAX];

  Dlug::Dlug(void)
  {
    Lp = Sum = 0;        // <-- czyscimy pola obiektu
    Imie[0] = '?';
    Imie[1] = '\0';
  }
```

Konstruktor, inicjując, zeruje numeryczne pola obiektu, a w miejsce łańcucha znakowego zapisuje pytajnik i znak końca tekstu. Są to typowe środki ostrożności. Dzięki nim w razie próby odczytania zawartości pustego obiektu, czyli rekordu, który nie został jeszcze w sposób świadomy i sensowny wprowadzony do bazy danych, otrzymamy klarowne wskazanie.

Aby nasz obiekt nie był „głuchym niemową”, musimy zadbać o obecność metod zapewniających odczytanie zawartości z rekordu (tu: `Print()`) i zapisanie danych do rekordu (`Update()`). „Update” znaczy dosłownie „aktualizuj”, czyli właśnie „wprowadź nową zawartość”. Przesyłać informacje będziemy w dwu kierunkach:

`Dlug` --> na ekran

`Dlug` <-- z klawiatury

W tym przypadku posłużymy się typowymi strumieniami wejścia/wyjścia. Szkielet metod jest oczywisty:

```
  void Dlug::Print()
  {
    cout << Lp + 1;
    cout << Imie;
    cout << Sum;
   }
  void Dlug::Update()
  {
    Print();             // <-- wydrukuj przed aktualizacja
    cout << Lp +1;       // <-- podaj automatycznie numer rekordu
```

```
    cin >> Imie;          // <-- wczytaj nowe dane
    cin >> Sum;
    .....
    Print();              // <-- wydrukuj po aktualizacji
}
```

Aplikacje obsługujące bazę danych tym przede wszystkim różnią się od innych, że muszą obsługiwać pliki dyskowe. Poprzednia wersja tabeli z rekordami powinna zostać najpierw wczytana z dyskowego pliku bazy danych. Następnie po zaktualizowaniu zawartości rekordów należy tabelę ponownie zapisać do pliku dyskowego. Operacje

`Dlug` --> do pliku dyskowego

`Dlug` <-- z pliku dyskowego

— wymagają zastosowania obiektów-strumieni predefiniowanych klas:

```
ifstream plik_we;  ofstream plik_wy;  // <-- deklaracja obiektow
int i = 0;          // <-- numer obiektu - rekordu
....
ifstream plik_we("A:\\AM_DBF.DAT");   // <-- utworzenie obiektow
ofstream plik_wy("A:\\AM_DBF.DAT");
```

Przypomnijmy, że `ifstream` to obiekt-strumień danych z pliku wejściowego, natomiast `ofstream` to obiekt-strumień danych do pliku wyjściowego. Zapis:

```
ifstream plik_we("A:\\AM_DBF.DAT");
```

— stanowi wywołanie konstruktora jednoargumentowego i połączenie strumienia z fizycznym plikiem wyjściowym, tu: *AM_DBF.DAT* na dyskietce *A:*. Plik wejściowy i plik wyjściowy może (i często jest) tym samym fizycznym plikiem dyskowym. Może tak być, jeśli dopilnujemy, by aplikacja nie próbowała wykonywać operacji odczytu lub zapisu w przypadkowych, nieprzemyślanych momentach. Jeśli jest to niezbędne, należy rozważyć możliwość utworzenia własnych buforów wejścia/wyjścia. Jeśli jednak będziemy postępować uważnie, automatyczne buforowanie powinno okazać się wystarczające — i tak jest w tym przykładzie. Aby porządkowanie obiektów przebiegało w sposób klarowny i przejrzysty, numer rekordu jest zmienną globalną `i`. Na początku numer ten zostaje zainicjowany jako `0`.

Dyskowe operacje zapisu/odczytu mogą przebiegać strumieniowo, przy czym nie musimy martwić się o to, że dane są różnych typów. Podobnie jak strumienie obiektów `cin` i `cout`, obiekty `ifstream` i `ofstream` prawidłowo rozpoznają i obsługują różne typy danych. A zatem zapis i odczyt mogą zostać zapisane tak:

```
Dlug R[MAX];
ifstream plik_we("A:\\AM_DBF.DAT");
plik_we >> R[j].Lp >> R[j].Imie >> R[j].Sum;
```


programowanie_strukturalne_DODATEK.indb 188 2010-07-05 11:54:57

```
ofstream plik_wy("A:\\AM_DBF.DAT");
plik_wy << R[j].Lp << endl << R[j].Imie << endl;
plik_wy << R[j].Sum << endl;
```

Przy zapisie danych do pliku aplikacje często stosują dodawanie separatorów, by uniknąć sklejenia pól. W tym przypadku jako znak separatora wykorzystany został `endl` (*END of Line character* — znak końca wiersza, inaczej `'\n'`).

Tak byłoby dla pojedynczego obiektu-rekordu. Skoro jednak mamy całą tabelę, a rekordy mogą zostać ponumerowane, możemy zapisywać tabelę i odczytywać ją w całości, posługując się w tym celu nieskomplikowaną pętlą programową. Funkcje realizujące te operacje zostały nazwane `ReadTable()` (wczytaj tabelę) i `StoreTable()` (zapamiętaj tabelę).

```
void ReadTable(void)
{
  ifstream plik_we("A:\\AM_DBF.DAT");
  for(int j=0; j<MAX; j++)
    plik_we >> R[j].Lp >> R[j].Imie >> R[j].Sum;
}
void StoreTable(void)
{
  ofstream plik_wy("A:\\AM_DBF.DAT");
  for(int j=0; j<MAX; j++)
  {
    plik_wy << R[j].Lp << endl << R[j].Imie << endl;
    plik_wy << R[j].Sum << endl;
  }
}
```

Aby w aplikacji panował porządek, a użytkownik otrzymał wyraźny komunikat, który rekord jest w danej chwili aktywny, czyli dostępny do modyfikacji, najlepiej zastosować flagę-znacznik `int Aktywny` (1 lub 0, Tak lub Nie). Oprócz tego należy zadbać, by rekordy były drukowane na ekranie zawsze w tych samych miejscach. Poza oczywistym uporządkowaniem ekranu uzyskujemy tu także dodatkowo samoczynne nadpisywanie się nowej zawartości na starej. Nie jest to najbardziej eleganckie rozwiązanie, w Windows załatwiają to automatycznie obiekty — pola edycyjne `TEditBox/TEdit`. Tutaj jednakże, by nie zaciemniać obrazu i nie rozbudowywać aplikacji ponad miarę, zostanie użyta funkcja `gotoxy()`, ustawiająca kursor tekstowy w zadanym miejscu ekranu tekstowego. Po przyjęciu takiego uproszczenia kod metod może wyglądać tak jak na poniższym listingu.

Listing D33.CPP

```
class Dlug
{
public:
  int Lp;
  char Imie[10];
  float Sum;
  int Aktywny;
  Dlug(void);
  void Print(int offset);
  void Update(int offset2);
} R[MAX];

Dlug::Dlug(void)
{
  Lp = Sum = Aktywny = 0;
  Imie[0] = '?';
  Imie[1] = '\0';
}

void Dlug::Print(int gdzie)
{
  gotoxy(1, gdzie); cout << Lp + 1;
  gotoxy(5, gdzie); cout << Imie;
  gotoxy(20, gdzie); cout << Sum;
  if (Aktywny == 1) cout << " <-- ";
  else cout << ".     ";
}

void Dlug::Update(int gdzie = 12)
{
  Aktywny = 1;
  Print(4 + Lp);
  gotoxy(1, gdzie); cout << Lp + 1 << ".                                  ";
  gotoxy(5, gdzie); cin >> Imie;
```

```
    gotoxy(20, gdzie); cin >> Sum;
    Aktywny = 0;
    Print(4 + Lp);
  }
```

Dzięki zastosowaniu funkcji `gotoxy()` tabela, rekordy w tabeli, rekord do aktualizacji i znacznik aktywnego rekordu będą zawsze drukowane na ekranie w tych samych miejscach. Strzałka (<--) będzie wskazywać aktywny rekord (jego starą zawartość w tabeli), a kropka (.) — rekordy pozostałe.

Do kompletu potrzebne są jeszcze dwie funkcje. Pierwsza posłuży do wyprowadzenia na ekran całej tabeli (np. świeżo wczytanej z dysku). Nosi ona nazwę `PrintTable()`. Druga sporządzi dla użytkownika raport na podstawie bieżącej zawartości tabeli bazy danych. Funkcja będzie sumować zawartości pól `Sum` wszystkich rekordów tabeli i wyprowadzać na ekran wynik, który otrzymał nazwę `Bilans`. Dodana na końcu zwłoka `delay()` pozwoli użytkownikowi odczytać końcowy wynik.

```
void PrintTable(void)
{
  for(int j=0; j<MAX; j++)
    R[j].Print(4 + R[j].Lp);
}


void Raport(void)
{
  int Bilans = 0;
  for(int j = 0; j < MAX; j++)
    Bilans += R[j].Sum;
  gotoxy(20, 22);
  cout << "BILANS: " << Bilans;
  delay(500);
  return;
}
```

Zwłoka ta jest dość mała, ale taka decyzja została podjęta w pełni świadomie. Aby zademonstrować pewien efekt uboczny, jest wywoływana z wnętrza destruktora obiektu `Dlug`. Jeśli więc 7 destruktorów wywoła ją kolejno, czas dla użytkownika wyniesie 3,5 sekundy.

Aplikacja jest już właściwie niemal kompletna. Pozostało jeszcze tylko napisać szybko prosty, a w miarę wygodny interfejs użytkownika, bo nie wypada przecież, by obiektowa baza danych działała inaczej niż w sposób zdarzeniowy... Działanie interfejsu i całej aplikacji można ująć tak jak w tabeli D.8.

programowanie_strukturalne_DODATEK.indb 191 2010-07-05 11:54:58

Tabela D.8. Obsługa zdarzeń bazy danych

ZDARZENIE	OPERACJA	UWAGI
Start aplikacji	Start pętli wczytywania znaków (naciśniętych klawiszy)	
[F1]	Drukowanie tabeli rekordów na ekranie	Nawet jeśli rekordy są „puste”
[F2]	Wczytanie poprzedniej zawartości wszystkich rekordów z dysku	Bez drukowania
[F3]	Zapis zawartości wszystkich rekordów do pliku	Po zakończeniu aktualizacji bieżącego rekordu
[F4]	Aktualizacja bieżącego rekordu	Nie można przerwać
[F5]	Zakończenie aplikacji	i wydrukowanie bilansu
[Enter]	Następny rekord do aktualizacji	

Działanie jest typowe na tyle, na ile tak nieskomplikowana aplikacja może zilustrować mechanizmy działania obiektowych baz danych. Interfejs użytkownika umieszczony w ciele funkcji `main()` wygląda następująco:

```
int main(void)
{
  char znak;

  for(int j=0; j<MAX; j++)
    R[j].Lp = j;
  DrukujInfo();

  while(1)
  {
    if (getch() =='\0')
      znak = getch();
    switch(znak)
    {
      case ';': PrintTable(); break;          // [F1]
      case '<': ReadTable(); break;           // [F2]
      case '=': StoreTable(); break;          // [F3]
      case '>': R[i].Update(); break;         // [F4]
      case '?': return (0);                   // [F5]
```

```
    }
  }
}
```

Napisaną w taki sposób obiektową aplikację można łatwo rozbudować o obsługę innych plików dyskowych. Wystarczy użyć parametru przy tworzeniu obiektów `ifstream` i `ofstream`. Równie prosto można zaimplementować zapytanie wybierające z tabeli na przykład tylko naszych dłużników, a pomijające milczeniem wierzycieli.

Dla wygody użytkownika można dodać jeszcze mechanizm *autoscroll*, czyli automatycznego przewijania rekordów, dbając jednocześnie, by numer rekordu nie przekroczył dopuszczalnego zakresu wartości:

```
i = ++i % MAX;
```

Poniżej zamieszczony został pełny kod źródłowy aplikacji *AMdbase.CPP* po dokonaniu wszystkich modyfikacji i uzupełnień.

Listing D34.CPP

```
/*  Object Oriented Data Base Management System – C. A Majczak 2009  */
/*  UWAGA: Baza danych pracuje z DYSKIETKA! Jesli chcesz, zmien  */
/*  nazwe i sciezke do pliku roboczego bazy danych. – np. C:\  */

#define MAX 7
#include <conio.h>
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <dos.h>

void DrukujInfo(void)
{
  clrscr();
      cout  <<  "[F1]  PrTb,    [F2]  Odczyt,  [F3]  Zapis,  [Enter]  -
Nastepny...\n";
   cout << "[F4] - Aktualizacja      [F5] - KONIEC. ";
}

class Dlug
{
public:
```

```
    int Lp;
    char Imie[10];
    float Sum;
    int Aktywny;
    Dlug(void);
    void Print(int offset);
    void Update(int offset2);
    ~Dlug(void);
  } R[MAX];

  Dlug::Dlug(void)
  {
    Lp = Sum = Aktywny = 0;
    Imie[0] = '?';
    Imie[1] = '\0';
  }

  ifstream plik_we; ofstream plik_wy;
  int i = 0;

  void Dlug::Print(int gdzie)
  {
    gotoxy(1, gdzie); cout << Lp + 1;
    gotoxy(5, gdzie); cout << Imie;
    gotoxy(20, gdzie); cout << Sum;
    if (Aktywny == 1) cout << " <-- "; else cout << ".     ";
  }

  void Dlug::Update(int gdzie = 12)
  {
    Aktywny = 1;
    Print(4 + Lp);
    gotoxy(1, gdzie); cout << Lp+1 << ".                           ";
    gotoxy(5, gdzie); cin >> Imie;
    gotoxy(20, gdzie); cin >> Sum;
```

```
  Aktywny = 0;
  Print(4 + Lp);
  i = ++i % MAX;
}

void PrintTable(void)
{
  for(int j=0; j<MAX; j++)
    R[j].Print(4 + R[j].Lp);
}

void ReadTable(void)
{                                             // jesli pendrive to E:, to
  ifstream plik_we("A:\\AM_DBF.DAT");         // "E:\\AM_DBF.DAT"
  for(int j=0; j<MAX; j++)
    plik_we >> R[j].Lp >> R[j].Imie >> R[j].Sum;
}

void StoreTable(void)
{
  ofstream plik_wy("A:\\AM_DBF.DAT");
  for(int j=0; j<MAX; j++)
  {
    plik_wy << R[j].Lp << endl << R[j].Imie << endl;
    plik_wy << R[j].Sum << endl;
  }
}

void Raport(void)
{
  int Bilans = 0;
  for(int j = 0; j < MAX; j++) Bilans += R[j].Sum;
  gotoxy(20, 22);
  cout << "BILANS: " << Bilans;
  delay(500);
```

```
  return;
}

Dlug::~Dlug(void)
{
  Raport();
}

int main(void)
{
  char znak;

  for(int j=0; j<MAX; j++)
    R[j].Lp = j;
  DrukujInfo();

  while(1)
  {
    if (getch() =='\0')
      znak = getch();
    switch(znak)
    {
      case ';': PrintTable(); break;          // [F1]
      case '<': ReadTable(); break;           // [F2]
      case '=': StoreTable(); break;          // [F3]
      case '>': R[i].Update(); break;         // [F4]
      case '?': return(0);                    // [F5]
    }
  }
}
```

Dzięki działaniu konstruktora aplikacja działa poprawnie, nawet jeśli liczba rekordów w tabeli jest mniejsza niż `MAX`, czyli tabela nie jest całkowicie wypełniona. Działanie konstruktora można sprawdzić, poprzedzając pętlę inicjującą znakami komentarza:

```
// for(int j=0; j<MAX; j++)
//   R[j].Lp = j;
```

Wydrukowanie tabeli jeszcze przed jej załadowaniem z dysku i przed aktualizacją spowoduje wyprowadzenie na ekran wszystkich rekordów z zainicjowanym zerowym numerem rekordu. Aplikacja nie będzie oczywiście działać wtedy poprawnie.

W poprzednich opisach pominięta została implementacja destruktora `~Dlug(void)` oraz funkcji `void DrukujInfo(void)`, jednakże ich działanie wydaje się dość oczywiste. Można łatwo przekonać się, że zwiększenie liczby obiektów-rekordów nie spowoduje ani zakłóceń, ani nawet wyraźnego spowolnienia pracy aplikacji.

5.3. Przykład przeciążenia operatora <<

Operator << (*insertor*) dobrze radzi sobie z wyprowadzaniem na ekran danych typów prostych (zarówno znakowych, jak i numerycznych), a nawet tablic znakowych. Naszym zadaniem będzie nauczenie go, jak wyprowadzać na ekran zawartość pól struktury. Do eksperymentów użyjemy własnego typu strukturalnego danych nazwanych tu `Dane`.

```
struct Dane
{
  char nazwisko[20];
  int wiek;
  float wzrost;
};
```

Ten typ strukturalny zawiera trzy pola typów prostych. Każde z nich moglibyśmy ręcznie wyprowadzić na ekran na przykład tak:

```
struct Dane D;
...
cout << D.nazwisko;
cout << D.wiek;
cout << D.wzrost;
```

— lub tak:

```
cout << D.nazwisko  << D.wiek << D.wzrost;
```

Pierwszy krok polega na dodaniu do programu nowej definicji funkcji operatorowej:

```
operator << ()
```

Konstrukcja nazwy takiej funkcji składa się w C++ z tych samych elementów, co normalnej funkcji (tj. typ wartości zwracanej, nazwa funkcji, lista argumentów w nawiasie), jednak nazwa jest zbudowana w sposób szczególny. Składa się mianowicie ze słowa kluczowego `operator`, spacji i znaku operatora (tu: <<).

```
ostream& operator << (ostream& str_out, struct Dane& D)
{
```


programowanie_strukturalne_DODATEK.indb 197 2010-07-05 11:54:58

```
  str_out << D.nazwisko << '\t';
  str_out << D.wiek << '\t';
  str_out << D.wzrost << '\n';
  return (str_out);
};
```

Pierwsze trzy wiersze powodują przesłanie pól struktury `D` typu `Dane` do jakiegoś obiektu `str_out` (to predefiniowany strumień wyjściowy):

```
  str_out << D.nazwisko << '\t';
  str_out << D.wiek << '\t';
  str_out << D.wzrost << '\n';
```

To, co robimy w tym miejscu, to zapis danych do pewnego specjalnego obszaru pamięci, nazywanego **buforem**. Dla przykładu załóżmy, że zmienna `D` zawiera następujące dane:

```
D.nazwisko="Maryna";
D.wiek=33;
D.wzrost=175;
```

Gdy funkcja zadziała, dane zostaną zapisane do bufora strumienia wyjściowego. Ostatni wiersz kodu definicji jest następujący:

```
return (str_out);
```

Zawartość bufora zostaje przesłana do strumienia wyjściowego, a następnie na ekran. I już. Można tu dodać, że strumień wyjściowy może być skierowany nie tylko na ekran, ale także np. do pliku dyskowego.

Listing D35.CPP

```
#include <iostream.h>

struct Dane
{
  char nazwisko[20];
  int wiek;
  float wzrost;
};

ostream& operator << (ostream& str_out, Dane& D)
{
  str_out << D.nazwisko << '\t';
```


programowanie_strukturalne_DODATEK.indb 198 2010-07-05 11:54:58

```
  str_out << D.wiek << '\t';
  str_out << D.wzrost << " metra." << '\n';
  return(str_out);
};

main()
{
  struct Dane osoba1 = {"Maly Kazio", 21, 0.855};
  struct Dane osoba2 = {"Duzy Kazimierz", 42, 1.85};
  struct Dane osoba3 = {"Ronald", 55, 1.56};
  cout << osoba1;
  cout << osoba2;
  cout << osoba3;
  return(0);
}
```

Jeśli nie chcemy oglądać wydruku wyjściowego w oknie *User screen*, możemy dodać wywołanie funkcji `getch()`.

Jeśli operujemy znacznymi ilościami danych, które powinny być na wyjściu formatowane w identyczny sposób, przeprowadzenie rozbudowy operatora jest zapewne najwygodniejszą metodą rozwiązania takiego problemu.

Zapis:

```
ostream& operator << (ostream& str_out, Dane& D)
```

— oznacza, że funkcja pobiera jako argumenty wejściowe:

- `Dane& D` — referencję do struktury `D` typu `Dane`,
- `ostream& str_out` — referencję do strumienia wyjściowego `str_out` typu `ostream` (*ostream* to skrót od *Output Stream*, czyli strumień wyjściowy).

Funkcja zwraca w miejsce operatora referencję do strumienia wyjściowego `ostream`. To ostatnie działanie jest ważne, ponieważ pozwala łączyć przeciążone operatory w łańcuch.

programowanie_strukturalne_DODATEK.indb 199 2010-07-05 11:54:58

Część VI

O metodologii metodologii programowania

Ta część „Dodatku" przeznaczona jest głównie dla nauczycieli, ale jeśli przeczytają ją uczniowie lub słuchacze studium, nie grozi to bynajmniej zdradzeniem jakiejś tajemnicy zawodowej. Tę część dedykuję moim Koleżankom i Kolegom po fachu, bowiem jej zrozumienie wymaga nieco więcej wiedzy zawodowej, ale może być pomocne przy wyjaśnianiu tych zagadnień uczniom.

Poprawne opanowanie metodyki programowania wymaga w istocie umiejętnego połączenia kilku zakresów wiedzy. Metodyka programowania to w istocie:

1. Zrozumienie w pewnym zakresie teorii algorytmów i opanowanie umiejętności ich implementacji:
   a) umiejętność stworzenia i zweryfikowania algorytmu zmierzającego do rozwiązania problemu pewnej kategorii,

   b) znajomość języków programowania strukturalnego i obiektowego, by móc **zaimplementować** algorytm w wybranym języku programowania,

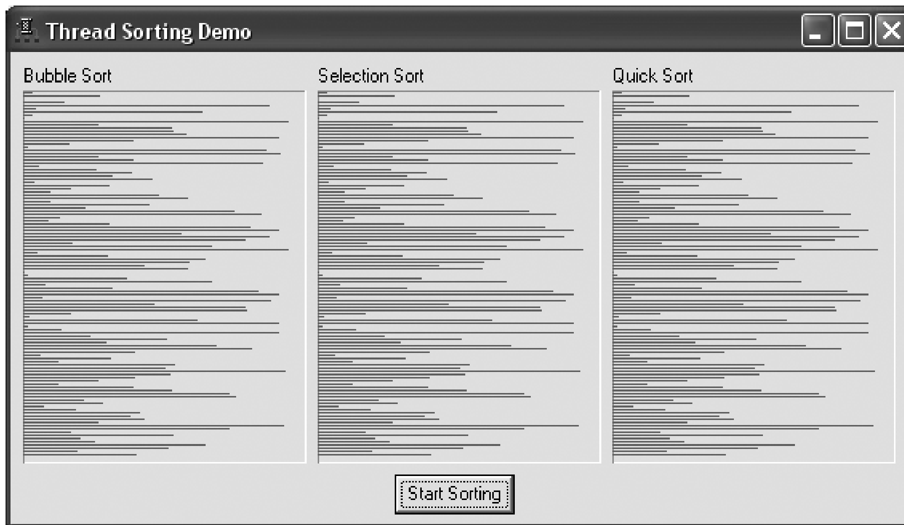
   c) umiejętność posługiwania się konkretnymi narzędziami, np. Turbo Pascal, Delphi, Visual C++, C++ Builder, itp.
2. Zapoznanie się z dostępnymi w typowym środowisku operacyjnym narzędziami i możliwościami pracy administratora i programisty.
3. Zrozumienie podstawowych kategorii myślowych pozwalających na poprawny dobór narzędzi i wybór poprawnej metody rozwiązania problemu.

Punkt (1) jest zdecydowanie najtrudniejszy. Wymaga choćby intuicyjnego wyjaśnienia pojęć „zbieżności" i „efektywności" algorytmów. Algorytmy mogą bowiem angażować różne zasoby systemu (pamięć, miejsce na dysku, czas procesora itp.). Wykorzystując programy przykładowe z książki i z „Dodatku" można łatwo stworzyć programy, które

będą mierzyć czas rzeczywisty potrzebny do wykonywania zadań i pozwolą pokazać uczniom / słuchaczom różne rozwiązania o różnej prędkości działania.

UWAGA

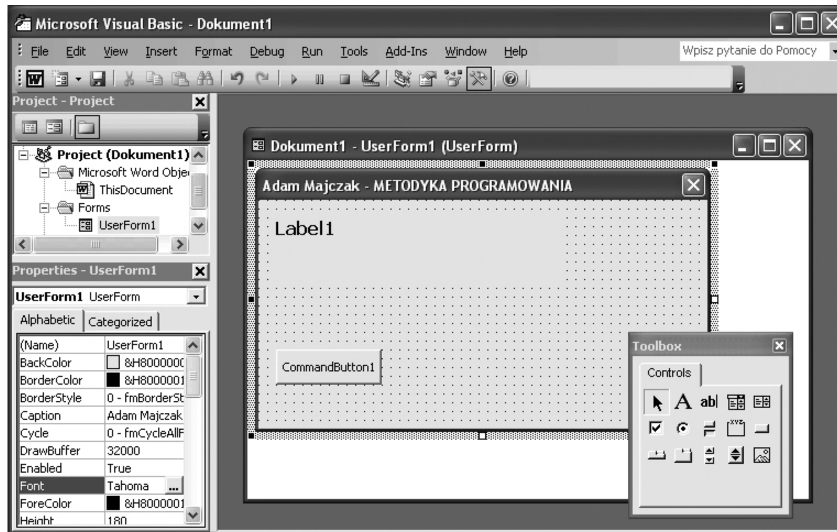
Dobrym praktycznym przykładem jest program THREADS (w Delphi i w C++ Builderze nazywa się tak samo i jest w przykładach — EXAMPLES). Ten przykład porównuje i demonstruje graficznie różnice w prędkości sortowania trzema popularnymi metodami.



Rysunek D30. Sortowanie współbieżne (wielowątkowe) w C++ Builderze — THREADS

Warto pokazać uczniom (wielu użytkowników PC z Windows latami nie zdaje sobie z tego sprawy), że w edytorze MS Word, Excelu, Power Poincie itp. jest wbudowany Visual Basic for Applications z całkiem szerokimi możliwościami tworzenia aplikacji dla Windows.

programowanie_strukturalne_DODATEK.indb 202 2010-07-05 11:54:58

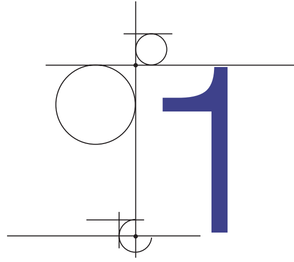


Rys. D31. Środowisko VBA (Visual Basic for Applications) pod dokumentem Worda

Prócz tego i w Open Office, i w MS Office mamy dostępny zawsze SQL (np. MSQuery).

„Dodatek" jest przeznaczony do powtórzenia i utrwalenia wiadomości. Zawiera przykłady zapisu zgodnie z zasadami programowania sekwencyjnego, strukturalnego, obiektowego i zdarzeniowego.

„Dodatek", jak to dodatek, przeznaczony jest do lektury i wykorzystania w zasadzie po opanowaniu materiału podstawowego zawartego w wersji drukowanej podręcznika.

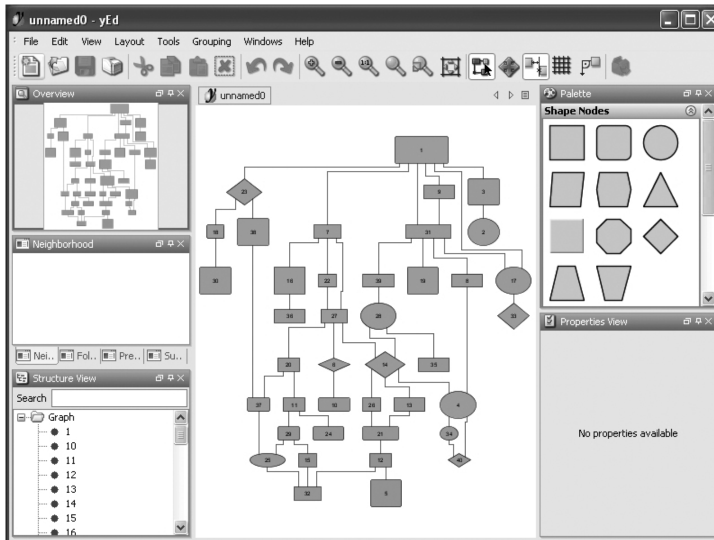


O algorytmach i stylach programowania

W Internecie istnieje wiele możliwości „zaopatrzenia się” w przydatne narzędzia. Edytorów graficznych pozwalających na konstruowanie i wykorzystanie schematów blokowych jest wiele. Oto link do jednego spośród takich edytorów dostępnych w sieci bezpłatnie.

http://www.yworks.com/en/products_yed_about.html?gclid=COrnvNyg2Z8CFc0SzAodIhvmGw

Posługując się dowolną wyszukiwarką (najlepiej używając hasła „Flowchart Editor”), można wyszukać i wybrać dowolny inny.



Rys. D32. Przykładowe okno edytora schematów blokowych yEd

1.1. Zdefiniowanie problemu

Pracownik otrzymuje w formie wypłat pewne sumy pieniędzy (nie zawsze dokładnie równe cenom towarów) $+x_i$ i ponosi różne wydatki $-y_j$. Po upływie pewnego czasu powinien mieć

$$Suma = \sum x_i + \sum -y_j$$

Przychody i wydatki wpisuje do komputera.

Pytanie: Jak powinien działać program zliczający poprawnie przychody, rozchody i bilans?

Załóżmy, że wprowadzenie zera powoduje przerwanie działania programu.

Ponieważ nie zawsze wiadomo, ile będzie tych przychodów i wydatków (może tylko po kilka pozycji, a może kilkanaście czy kilkadziesiąt), napiszemy nasz algorytm (a w efekcie nasze programy) w taki sposób, by można było wprowadzić zmienną ilość wpływów i wydatków.

1.2. Rozwiązanie 1 — prosty program sekwencyjny

Najprostsze rozwiązanie problemu w postaci kodu sekwencyjnego może wyglądać np. tak jak na poniższym listingu. Program działa w pętli programowej (pętla *do-while*), którą przerywa wpisanie zera. To nawet dość logiczne, bo zerowy przychód lub zerowy rozchód w życiu także może oznaczać „przerwę".

Listing D2-01.C

```
#include <stdio.h>
#include <conio.h>
main()
{
  float suma = 0, przychody = 0, wydatki = 0;
  float kwota;
  kwota = 0.0001;          //  zmienna sterujaca musi byc rozna od ZERA
  clrscr();
  do
  {
    puts("\nWczytuje przychody (+) i wydatki (-). Wpisz kwote: ");
    scanf("%f", &kwota);
    suma = suma + kwota;
```


programowanie_strukturalne_DODATEK.indb 205 2010-07-05 11:54:59

```
      if (kwota > 0) przychody += kwota;
      else wydatki += kwota;
      printf("Suma: %4.2f, Przychody: %4.2f ", suma, przychody);
      printf("Wydatki: %4.2f \n", wydatki);
    } while (kwota != 0.0);
    putch('\n');
    puts("Koniec liczenia.");
    getch();
    return 0;
  }
```

Pętlę programową można by przerwać również inaczej, np. stosując instrukcję `break`. Ale to szczegóły techniczne. Program można łatwo podzielić na funkcje i procedury w taki sposób, by kod stał się strukturalny.

1.3. Rozwiązanie 2 — kod strukturalny

Należy tylko pamiętać o dwóch ważnych zasadach. Zarówno wszystkie zmienne, jak i wszystkie funkcje muszą być zadeklarowane ***przed ich pierwszym użyciem***. Definicje funkcji mogą być umieszczone po funkcji `main()` pod warunkiem, że przed nią będą umieszczone prototypy (deklaracje) wszystkich funkcji. W tym przykładzie wszystkie funkcje są typu `void`, czyli są to odpowiedniki procedur.

Listing D2-02.C

```
#include <stdio.h>
#include <conio.h>
float suma = 0, przychody = 0, wydatki = 0, kwota;
void WczytajDane(void)
{
  puts("\nWczytuje przychody (+) i wydatki (-). Wpisz kwote: ");
  scanf("%f", &kwota);
}
void Obliczenia(void)
{
  suma = suma + kwota;
  if (kwota > 0) przychody += kwota;
  else wydatki += kwota;
```

```
}
void DrukujWynik(void)
{
  printf("Suma: %4.2f, Przychody: %4.2f ", suma, przychody);
  printf("Wydatki: %4.2f \n", wydatki);
}
void NapisyKoncowe(void)
{
  putch('\n');
  puts("Koniec liczenia.");
  getch();
}

main()
{
  kwota = 0.0001;
  clrscr();
  do
  {
    WczytajDane();
    Obliczenia();
    DrukujWynik();
  } while (kwota != 0.0);
  NapisyKoncowe();
  return 0;
}
```

Działanie programu będzie identyczne. Zwróćmy uwagę, że plik zawierający kod źródłowy ma rozszerzenie *.C, zatem nie korzystamy z obiektowych możliwości C++. Jeśli chcemy zapisać ten sam program w sposób obiektowy, musimy pamiętać o zmianie domyślnego rozszerzenia i o zastosowaniu kompilatora C++, a nie C. Użyte tu funkcje (a właściwie procedury) w sposób naturalny stają się metodami należącymi do klasy.

1.4. Rozwiązanie 3 — kod obiektowy

Modyfikując nasz kod tak, by stał się zgodny z zasadami programowania obiektowego (ang. *OOP = Object-oriented programming* — programowanie obiektowe), możemy pozostawić dane jako prywatne. To status domyślny, ale póki z danych korzystają tylko własne metody, należące do tej samej klasy (a więc i obiektu), dane mogą pozostawać prywatne. Natomiast metody należy zadeklarować jako publiczne, by można je było wywołać z programu głównego. Oto przykładowe rozwiązanie obiektowe.

Listing D2-03.CPP

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>

class Licznik
{
  float suma, przychody, wydatki;        // tu nie mozna inicjowac
public:                                  // zmienne zainicjuje konstruktor
  float kwota;                           // dostepne z zewnatrz
  Licznik(void);                         // konstruktor bezparametrowy
  void WczytajDane(void);                // deklaracje metod
  void Obliczenia();                     // slowo void mozna pominac.
  void DrukujWynik();
};

Licznik::Licznik(void)                   // definicja konstruktora
{
  suma = 0;
  przychody = 0;
  wydatki = 0;
  kwota = 0.0001;                        // cokolwiek, ale nie zero
}

void Licznik::WczytajDane(void)
{
```

```
  puts("\nWczytuje przychody (+) i wydatki (-). Wpisz kwote: ");
  cin >> kwota;
}
void Licznik::Obliczenia(void)
{
  suma = suma + kwota;
  if (kwota > 0) przychody += kwota;
  else wydatki += kwota;
}
void Licznik::DrukujWynik(void)
{
  printf("Suma: %4.2f, Przychody: %4.2f ", suma, przychody);
  printf("Wydatki: %4.2f \n", wydatki);
}
void NapisyKoncowe(void)                    //  ta funkcja nie jest metoda
{
  putch('\n');
  puts("Koniec liczenia.");
  getch();
}

main()
{
  Licznik Obiekt;                       //  Konstruktor wykona „kwota = 0.0001;”
  clrscr();
  do
  {
    Obiekt.WczytajDane();
    Obiekt.Obliczenia();
    Obiekt.DrukujWynik();
  } while (Obiekt.kwota != 0.0);
  NapisyKoncowe();
```

```
    return 0;
}
```

Procedura `NapisyKoncowe()` może, ale nie musi stać się metodą. Może pozostawać poza obiektem.

1.5. Rozwiązanie 4 — kod zdarzeniowy

Ten problem nie wymaga, by program reagował na jakieś szczególne zdarzenia, które następują podczas jego działania w jego środowisku systemowym. Bądźmy jednak konsekwentni. To na tyle prosty przykład, że sposób organizacji kodu powinien być aż nadto jasny. Najpierw wyjaśnijmy podstawowe pojęcia:

Zdarzenie (ang. *Event*) — wydarzenie, które następuje w środowisku. System Windows rozróżnia ponad 500 różnych zdarzeń. Typowe zdarzenie to np. kliknięcie myszką, naciśnięcie klawisza na klawiaturze itp.

Pętla pobierania komunikatów o zdarzeniach (ang. *Message Loop*) — pętla programowa uruchamiana zaraz po starcie programu i działająca aż do końca pracy programu, pobierająca komunikaty o zdarzeniach od systemu operacyjnego (ang. Getting Messages, Pumping Messages). Komunikaty o zdarzeniach umieszczane są w kolejce (ang. Message Queue), a komunikaty, które nie są przeznaczone dla danego programu (i nie są przez niego obsługiwane, np. kliknięcie przez użytkownika w oknie innego programu), są „odprawiane" do obsługi przez inny program (ang. Dispatching Messages).

Event Handler — funkcja (najczęściej procedura) obsługi danego zdarzenia.

Jako się rzekło, program zdarzeniowy musi mieć pętlę pobierania komunikatów o zdarzeniach i handlery obsługi tych zdarzeń. Dla uproszczenia rozpatrzymy tylko dwa zdarzenia:

Zdarzenie 1 — Użytkownik nacisnął klawisz [R] i wydał polecenie „Wykonaj".

Zdarzenie 2 — Użytkownik nacisnął klawisz [Q] i wydał polecenie „Koniec".

Cała reszta kodu programu to w istocie kod obsługi zdarzenia „Wykonaj". A zapisać to można np. tak:

Listing D2-04.CPP

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>                         //  prototyp funkcji exit()

class Licznik
{
```

```
  float suma, przychody, wydatki;        // tu nie mozna inicjowac
public:                                  // zmienne zainicjuje konstruktor
  float kwota;                           // dostepne z zewnatrz
  Licznik(void);                         // konstruktor bezparametrowy
  void WczytajDane(void);                // deklaracje metod
  void Obliczenia();                     // slowo void mozna pominac
  void DrukujWynik();
};

Licznik::Licznik(void)                   // definicja konstruktora
{
  suma = 0;
  przychody = 0;
  wydatki = 0;
  kwota = 0.0001;                        // cokolwiek, byle nie zero
}

void Licznik::WczytajDane(void)
{
  puts("\nWczytuje przychody (+) i wydatki (-). Wpisz kwote: ");
  cin >> kwota;
}
void Licznik::Obliczenia(void)
{
  suma = suma + kwota;
  if (kwota > 0) przychody += kwota;
  else wydatki += kwota;
}
void Licznik::DrukujWynik(void)
{
  printf("Suma: %4.2f, Przychody: %4.2f ", suma, przychody);
  printf("Wydatki: %4.2f \n", wydatki);
```


programowanie_strukturalne_DODATEK.indb 211 2010-07-05 11:54:59

```
}
void NapisyKoncowe(void)                         // ta funkcja nie jest metoda
{
  putch('\n');
  puts("Koniec liczenia.");
  getch();
}

Licznik Obiekt;

void Wykonaj(void)
{
  do
  {
    Obiekt.WczytajDane();
    Obiekt.Obliczenia();
    Obiekt.DrukujWynik();
  } while (Obiekt.kwota != 0.0);
  return;
}

void Koniec(void)
{
  exit(1);
}

void PetlaKomunikatow(void)
{
  while(1)
  {
    if(kbhit())                                  // jesli uzytkownik nacisnal klawisz
    {
```


programowanie_strukturalne_DODATEK.indb 212 2010-07-05 11:54:59

```
        char K = getch();                    // wczytujemy klawisz
        if(K=='R' || K=='r') Wykonaj();
        if(K=='Q' || K=='q') Koniec();
      }
    }
}

main()
{
  clrscr();
  PetlaKomunikatow();
  NapisyKoncowe();
  return 0;
}
```

Zwróćmy uwagę, że teraz program nie ruszy bez naciśnięcia klawisza *[R]* i nie zakończy się bez naciśnięcia klawisza *[Q]*. Poza tym działanie programu pozostaje bez zmian. Teraz po wpisaniu zera możemy ponownie wrócić do programu, nacisnąwszy klawisz *[R]*.

Zauważmy jeszcze, że w miarę ewolucji stylu programowania kod funkcji *main()* staje się coraz krótszy i coraz prostszy. W praktyce programowania to, co znajduje się u nas poza funkcją *main()*, bardzo często znajduje się w standardowych plikach nagłówkowych dołączanych dyrektywą *#include <...>*. Pliki nagłówkowe C++ właśnie po to są, by zawierać deklaracje gotowych klas i gotowych funkcji. Podobnie pętla pobierania od systemu operacyjnego Windows komunikatów o zdarzeniach jest zazwyczaj ukryta i nie musimy „zawracać sobie głowy" szczegółami technicznymi. Jedynie handlery tych zdarzeń, które ma obsługiwać nasz program, zawsze musimy skonstruować samodzielnie.

programowanie_strukturalne_DODATEK.indb 213 2010-07-05 11:54:59

2

Visual C++, szablony i obsługa wyjątków

Podobnie jak w przypadku bibliotek klas (MFC, OWL itp.) i bibliotek komponentów wizualnych (np. VCL) producenci kompilatorów C++ często dodają do IDE biblioteki szablonów (templates). W Visual C++ jest to biblioteka ATL, popularną i szeroko dostępną wersją jest STL (Standard Templates Library). Te dość złożone zagadnienia wymagają przed przystąpieniem do ćwiczeń krótkiego wyjaśnienia i wprowadzenia. Wykorzystanie szablonów i obsługa sytuacji wyjątkowych (EH = Exception Handling) to kolejne dwa elementy, które silnie wpływają na styl programowania i praktyczny sposób konstruowania aplikacji C++ czy Javy.

W C wprowadzono strukturalną obsługę wyjątków, natomiast C++ obsługuje sytuacje wyjątkowe w sposób obiektowy. W C++ wyjątek lub sytuacja wyjątkowa to w istocie obiekt, który jest przekazywany (przenosząc przy okazji pewne informacje) z tego obszaru w kodzie, gdzie wystąpił problem, do tego obszaru w kodzie, który zajmuje się „pokojowym" rozwiązaniem tego problemu. Określenie „pokojowe rozwiązanie" oznacza tu przemyślaną obsługę sytuacji konfliktowej typu brak pamięci operacyjnej, nie istnieje potrzebny plik dyskowy, próba dzielenia przez zero itp., która w normalnych warunkach mogłaby spowodować przerwanie działania programu i, co groźniejsze, często utratę danych trudnych do odzyskania i odtworzenia.

Rodzaj wyrażenia (obiektu-wyjątku) może decydować o tym, która część kodu podejmie próbę rozwiązania konfliktowej sytuacji. Zawartość tak zgłaszanego obiektu-wyjątku (ang. thrown object) może decydować o sposobie powrotu do dalszej pracy aplikacji i o sposobie informowania użytkownika o wystąpieniu i obsłudze sytuacji wyjątkowej. Podstawowe zasady logiki obsługi wyjątków sprowadzają się do następujących działań:

1. Zidentyfikowanie tych fragmentów kodu, które potencjalnie mogą spowodować wystąpienie sytuacji wyjątkowej i ujęcie ich w bloki typu try (dosł.: try = spróbuj).

```
try
{
   // Kod zawierajacy ryzykowne operacje
   ...
}
```

2. Utworzenie bloków obsługi (dosł.: catch = przechwyć) przewidywanych, możliwych sytuacji wyjątkowych.

```
catch( OutOfMemory )              // braklo pamieci
{
   // cos robimy, by uratowac aplikacje...
   ...
}
catch( FileNotFound )             // brak pliku
{
   // robimy cos innego...
   ...
}
// ... itd. ...
```

Z technicznego punktu widzenia bloki `catch` powinny w kodzie występować bezpośrednio po bloku `try`. Sterowanie zostanie przekazane do bloku `catch` tylko wtedy, gdy przewidywana sytuacja wyjątkowa rzeczywiście wystąpi.

Przykładowy kod poniżej ilustruje praktyczne zastosowanie najprostszego, „pustego" obiektu-wyjątku przeznaczonego do obsługi typowej sytuacji konfliktowej — próby zapisania do tablicy elementu, dla którego nie ma przewidzianego miejsca (spoza dopuszczalnego zakresu indeksu), co może w praktyce zagrażać przerwaniem pracy aplikacji i utratą danych.

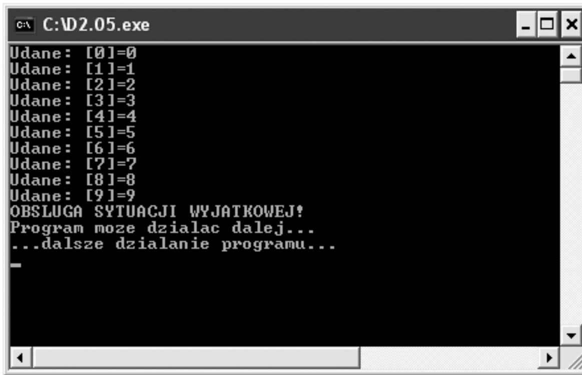
Listing D2.05.CPP

```
#include <iostream.h>
const int DefaultSize = 5;
class ZaDuzo {};                          // definicja klasy dla obiektu - wyjatku
class Array
{
private:
  int *Wskaznik;                          // dane prywatne
  int  rozmiar;                           // wskaznik i rozmiar tablicy
```

```
public:
  Array(int rozmiar = DefaultSize);      // dwa konstruktory
  Array(const Array &Tablica);
  ~Array() { delete [] Wskaznik;}
  Array& operator=(const Array&);        // przeciazone operatory
  int& operator[](int offSet);
  const int& operator[](int offSet) const;
  int PodajRozmiar() const { return rozmiar; }   // zwykla metoda
  friend ostream& operator<< (ostream&, const Array&);
};
Array::Array(int size) : rozmiar(size)
{
  Wskaznik = new int[size];
  for (int i = 0; i<size; i++) Wskaznik[i] = 0;
}
Array& Array::operator=(const Array &Tablica)
{
  if (this == &Tablica)
    return *this;
  delete [] Wskaznik;
  rozmiar = Tablica.PodajRozmiar();
  Wskaznik = new int[rozmiar];
  for (int i=0; i<rozmiar; i++)
    Wskaznik[i] = Tablica[i];
  return *this;
}
Array::Array(const Array &Tablica)
{
  rozmiar = Tablica.PodajRozmiar();
  Wskaznik = new int[rozmiar];
  for (int i = 0; i<rozmiar; i++)
    Wskaznik[i] = Tablica[i];
}
int& Array::operator[](int offSet)
{
```

```
  int size = PodajRozmiar();
  if (offSet >= 0 && offSet < PodajRozmiar())
    return Wskaznik[offSet];
  else
  {
    throw ZaDuzo();                          // zglaszany obiekt - wyjatek
    return Wskaznik[offSet];
  }
}
const int& Array::operator[](int offSet) const
{
  int mysize = PodajRozmiar();
  if (offSet >= 0 && offSet < PodajRozmiar())
    return Wskaznik[offSet];
  throw ZaDuzo();
  return Wskaznik[offSet];
}
ostream& operator<< (ostream& output, const Array& theArray)
{
  for (int i = 0; i<theArray.PodajRozmiar(); i++)
    output << "[" << i << "] " << theArray[i] << endl;
  return output;
}

int main()
{
  Array intArray(10);
  try
  {
    for (int j = 0; j< 100; j++)
    {
      intArray[j] = j;
      cout << "Udane: [" << j << "]=" << j << endl;
    }
  }
```

```
    catch (ZaDuzo)                          // przechwycenie obiektu - wyjatku
    {
      cout << "OBSLUGA SYTUACJI WYJATKOWEJ!\n";
    }
    cout << "Program moze dzialac dalej..." << endl;
    cout << "...dalsze dzialanie programu..." << endl;
    getchar();
    return 0;
}
```



Rysunek D33.
Obsługa sytuacji wyjątkowej w działaniu. Aplikacja konsoli Visual C++

Powyższy przykładowy prosty kod „przy okazji" demonstruje przeciążanie operatorów przy użyciu metod i przy zastosowaniu funkcji zewnętrznej kategorii `friend`.

2.1. Konstruowanie i stosowanie szablonów

Szablony wymagają zastosowania słowa kluczowego C++ `template`. Szablony można stosować w odniesieniu do funkcji. Oto prosty przykład parametryzacji algorytmu sortowania bąbelkowego przy użyciu szablonu

```
template <class V>
void VectorBubbleSort(V wektor[], int rozmiar)
```

dla jednowymiarowej tablicy (wektora) stanowiącej argument funkcji sortującej.

Listing Template1.CPP

```
// TemplateDemo – szablon uzyty na liscie argumentow funkcji
#include <iostream.h>
template <class V>
void VectorBubbleSort(V wektor[], int rozmiar)
{
  for(int i=0; i<(rozmiar-1); i++)
```

```
    for(int j=(rozmiar-1); i<j; j--)
      if( wektor[j-1] > wektor[j] )
        {
          V tymczasowa = wektor[j];
          wektor[j] = wektor[j-1];
          wektor[j-1] = tymczasowa;
        }
}

main()
{
  int Numerki[7] = {1, 3, 5, 4, 2, 0, 6};
  char Literki[5] = {'d', 'A', 't', 'Z', 'a'};

  VectorBubbleSort(Numerki, sizeof(Numerki)/sizeof(Numerki[0]));
  VectorBubbleSort(Literki, sizeof(Literki)/sizeof(Literki[0]));
  for(int i=0; i<sizeof(Numerki)/sizeof(Numerki[0]); i++)
    cout << "Num[" << i << "]=" << Numerki[i] << endl;
  for(int i=0; i<sizeof(Literki)/sizeof(Literki[0]); i++)
    cout << "Lit[" << i << "]=" << Literki[i] << endl;
  getchar();
  return 0;
}
```

Szablony zastosowane w odniesieniu do klas powodują utworzenie rodziny klas. Kompilator może następnie wygenerować samodzielnie nową klasę w oparciu o zadany szablon. A oto prosty przykład kolejkowania z zastosowaniem szablonu wobec własnej klasy BUFOR.

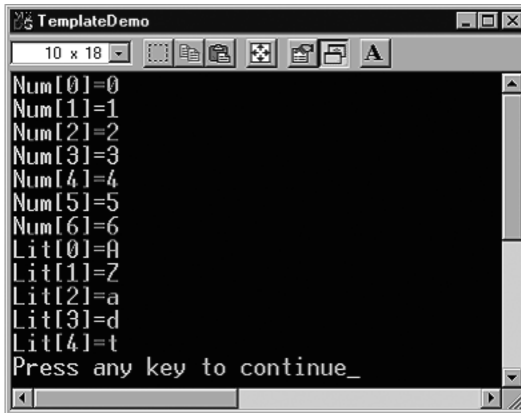
Listing Template2.CPP

```
#include <iostream.h>
const int DefaultSize = 5;
template <class K>
class BUFOR
{
public:
  BUFOR(int Rozmiar = DefaultSize)
  {
```

```
    Licznik = 0; Wskaznik = new K[Rozmiar];
  }
  ~BUFOR()
  {
    delete [] Wskaznik;
  }
  void DoBufora( K zmienna);
  K ZBufora();
protected:
  int Licznik;
  K *Wskaznik;
};

template <class K>
void BUFOR<K>::DoBufora(K szablon)
{
  Wskaznik[Licznik++] = szablon;
}

template <class K>
K BUFOR<K>::ZBufora(void)
{
  return Wskaznik[--Licznik];
}

main()
{
  BUFOR<int> B(5);
  B.DoBufora(1);
  B.DoBufora(2);
  B.DoBufora(3);
  cout << B.ZBufora() << ' ' << B.ZBufora() << ' '
    << B.ZBufora() << endl;
  BUFOR<double> D(4);
  D.DoBufora(1.11);
```

```
    D.DoBufora(2.12);
    D.DoBufora(3.13);
    cout << D.ZBufora() << ' ' << D.ZBufora() << ' '
      << D.ZBufora() << endl;
    getchar();
    return 0;
  }
```

Przy sposobności przykład ilustruje zastosowanie operatorów `new` i `delete`. Bufory tworzone z użyciem szablonu `K` mogą mieć różną wielkość i zawierać elementy różnych typów. Ta „typologiczna” elastyczność powoduje, że szablony są często stosowane w praktyce do tworzenia klas i obiektów-kontenerów „na różne różności” (*container classes*).



Rysunek D34. Sortowanie z zastosowaniem szablonu `<class V>`

Rysunek D35. Bufor z zastosowaniem szablonów w działaniu

Wszystkie przykładowe programy w wersji źródłowej znajdują się na nośniku elektronicznym w katalogu \PROG. Katalog ten został podzielony na podkatalogi (podfoldery) dla poszczególnych języków programowania. Poszczególne przykłady można również rozpoznać po rozszerzeniach: *.BAS, *.PAS, *.C, *.CPP. Każdy projekt Delphi i C++ Buildera znajduje się w odrębnym folderze.

3

Można zastosować JavaScript

Algorytmy wyszukiwania minimum, maksimum, zadanego elementu w zbiorze lub sortowania zbioru danych mogą stać się dobrym punktem wyjścia do rozważań o skuteczności różnych narzędzi i różnych metod programowania. Oto przykład (okrojony, ale działający) algorytmu sortującego napisanego w JavaScript. Skopiowanie poniższego tekstu i utworzenie pliku HTML wystarczy, by prześledzić działanie programu.

Listing Sort.HTML

```
<html>
  <body>
    <input onclick="sortuj()" type="button" value="Sortowanie"
name="B1">
    <script language=javascript>
      // Sortowanie przez wybór
      var MAX = 30;                  // wielkość tablicy do posortowania
      function sortuj()
      {
        var A = new Array(MAX);
        var i, j, x, t, pmin;        // deklarujemy zmienne
        // najpierw wypełniamy tablicę A[] liczbami pseudolosowymi
        for(i = 0; i < MAX; i++)
          A[i] = Math.floor(Math.random() * 1000);
        t = "Przed:<BR><BR>";        // akumulujemy tekst w buforze
        for(i = 0; i < MAX; i++)
          t = t + A[i] + " ";
```

```
            t += "<BR><BR>";
            // sortujemy
            for(j = 0; j < MAX - 1; j++)
            {
              pmin = j;
              for(i = j + 1; i < MAX; i++)
                if (A[i] < A[pmin])
                  pmin = i;
                x = A[pmin];
                A[pmin] = A[j];
                A[j] = x;
            }
                                              //  wyświetlamy wynik sortowania
            t += "Po sortowaniu:<BR><BR>";
            for(i = 0; i < MAX; i++) t += A[i] + " ";
              document.write(t);
          }
        </script>
      </body>
    </html>
```

Przytaczam tu powyższy kod tylko w celu zasygnalizowania, że w razie kłopotów z dostępem do kompilatora C++ wiele zagadnień można wyjaśnić i zilustrować, posługując się Notatnikiem Windows do edycji tekstu, a następnie jakąkolwiek przeglądarką, np. Internet Explorerem. JavaScript ma oczywiście istotne ograniczenia, ale składnia Javy i JavaScriptu jest na tyle zbliżona do C i C++, że zapisanie przykładowych programów w JavaScript nie powinno być szczególnie trudne. Zachęcam do samodzielnych eksperymentów, bowiem tylko wprawa czyni programistę.

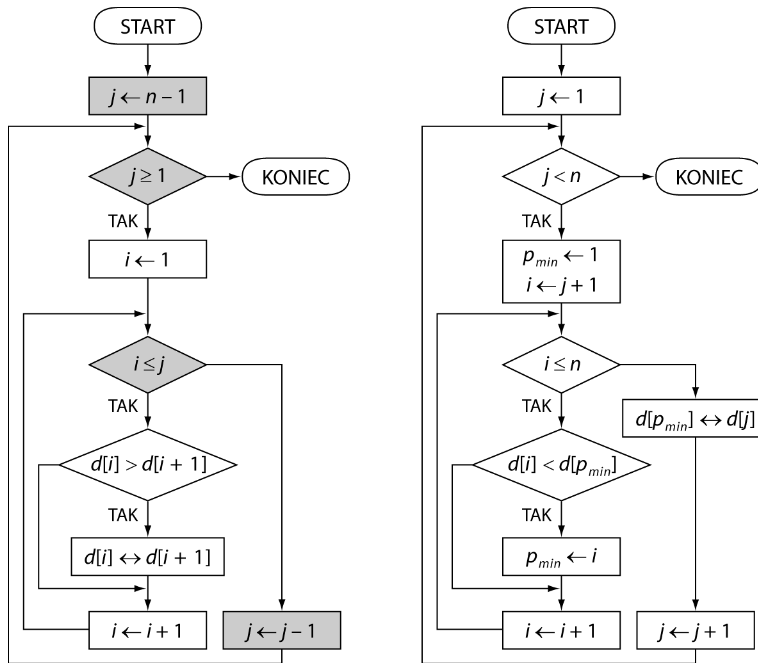
Chcąc korzystać z Javy, musimy skopiować ze strony internetowej Sun Microsystems pakiety:

JRE — Java Runtime Environment (tu jest interpreter JAVA.EXE)

JDK — Java Development Kit (tu jest kompilator JAVAC.EXE)

Obydwa te pakiety muszą być dobrane stosownie do naszej wersji środowiska operacyjnego (np. do Windows XP lub dla telefonu komórkowego, powiedzmy, Windows Mobile).

programowanie_strukturalne_DODATEK.indb 223 2010-07-05 11:55:00



Rysunek D36. Przykładowe algorytmy sortowania

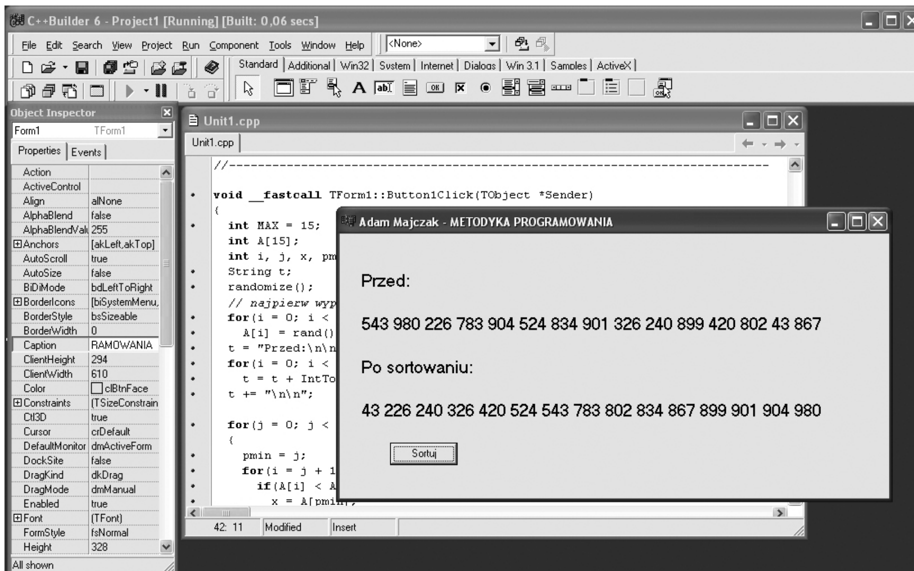
Dla porównania ten sam kod przeniesiony do C++ Buildera i wstawiony jako handler obsługi zdarzenia „ButtonClick", czyli kliknięcie przycisku, wygląda i działa, jak widać poniżej.

Listing Sort.CPP

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  int MAX = 15;                    // 15 a nie 30, żeby się mieściło na rysunku
  int A[15];
  int i, j, x, pmin;               //  deklarujemy zmienne
  String t;
  randomize();
  //  najpierw wypełniamy tablicę A[] liczbami pseudolosowymi
  for(i = 0; i < MAX; i++)
    A[i] = rand() % 1000;
  t = "Przed:\n\n";                //  akumulujemy tekst w buforze
  for(i = 0; i < MAX; i++)
    t = t + IntToStr(A[i]) + " ";        //  konwersja int -> String
```

```
    t += "\n\n";
                                        // sortujemy
    for(j = 0; j < MAX - 1; j++)
    {
      pmin = j;
      for(i = j + 1; i < MAX; i++)
        if(A[i] < A[pmin]) pmin = i;
          x = A[pmin];
      A[pmin] = A[j];
      A[j] = x;
    }
                                        // wyświetlamy wynik sortowania
    t += "Po sortowaniu:\n\n";
    for(i = 0; i < MAX; i++) t += IntToStr(A[i]) + " ";
      Label1->Caption = t;
  }
```

Na rysunku poniżej widać kod sortujący w działaniu w środowisku C++ Buildera.

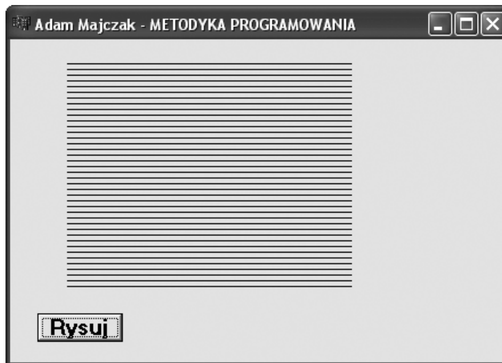


Rysunek D37. Sortowanie w C++ Builderze

programowanie_strukturalne_DODATEK.indb 225 2010-07-05 11:55:00

C++ Builder jest zorganizowany bardzo podobnie do Delphi, jednakże przy uruchamianiu programów przykładowych należy pamiętać o notacji wskaźnikowej stosowanej przez C++ Builder. Oto przykład prostego kodu i rezultat jego działania.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  for(int i=1; i<200; i+=5)
  {
    Form1->Canvas->MoveTo(50, 20+i);
    Form1->Canvas->LineTo(300, 20+i);
  }
}
```



Rysunek D38. Składnia stosowana w C++ Builderze i w Delphi jest podobna. Wyniki działania programu są zgodne z oczekiwaniami

WAŻNA UWAGA KOŃCOWA

Choć Basic (zarówno Visual Basic, jak i Visual Basic for Applications) jest — niesłusznie — lekceważony, składnia Pascala, C++, Basica i Javy staje się w pewnym stopniu coraz bardziej podobna. Znajomość, choćby pobieżna, VBA i JavaScript wydaje się przydatna, więc posługując się przykładami z niniejszej książki („Dodatku"), można i warto zaznajomić się również z tymi coraz popularniejszymi narzędziami. Ich podstawową zaletą jest powszechna dostępność i wykorzystanie standardowych obiektów (dokument, formularz itp.).

Ponieważ na kilkuset stronach nie da się zawrzeć wszystkiego, czytelników zainteresowanych doświadczeniami autora i wymianą spostrzeżeń (oraz dodatkowych materiałów i przykładów) zapraszam do skontaktowania się pocztą elektroniczną: *a_majczak@o2.pl*.

programowanie_strukturalne_DODATEK.indb 226 2010-07-05 11:55:00