

Halina Kwaśnicka

**Programowanie
obiektywne
w Turbo Pascalu**

**Oficyna Wydawnicza
Politechniki Wrocławskiej**





**BIBLIOTEKA GŁÓWNA
POLITECHNIKI WROCŁAWSKIEJ**



Politechnika Wrocławska



Halina Kwaśnicka

Programowanie obiektowe w Turbo Pascalu

Wrocław 1995

Opiniodawcy

Jan KWIATKOWSKI

Wiesław PORĘBSKI

Opracowanie redakcyjne i korekta

Barbara WACHOWSKA

Programy „FigureDemo”, „ListDemo” i moduł „Figures” oraz rysunek 8.2 ze strony 161 „Turbo Pascal 7.0, User's Guide” wykorzystano w pracy za zgodą firmy Borland.



287006 L11

© Copyright by Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław 1995

OFICyna WYDAWNICZA POLITECHNIKI WROCLAWSKIEJ

Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

ISBN 83-7085-157-6

Nakład 500+50 egz. Ark. wyd. 7,25. Ark. druk. 6¹/₄. Papier offset. kl. III, 70 g, B1.

Oddano do druku w lipcu 1995 r. Druk ukończono w sierpniu 1995 r.

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Zam. nr 515/95.

Spis treści

Przedmowa	7
-----------------	---

CZĘŚĆ I. PROGRAMOWANIE OBIEKTOWE W TP6.0 DEFINIOWANIE I PRZETWARZANIE OBIEKTÓW STATYCZNYCH

1. Główne cechy programowania obiektowego	9
2. Na czym polega dziedziczenie?	9
2.1. Realizacja dziedziczenia w Turbo Pascalu bez OOP	10
2.2. Rozwiązanie problemu dziedziczenia cech w programowaniu obiektowym	10
3. Definiowanie typów obiektowych i typów obiektowych potomnych	11
3.1. Deklaracja zmiennych typu obiektowego	12
3.2. Deklarowanie zmiennych z wartością początkową typu obiektowego	13
4. W jaki sposób są dostępne pola obiektu?	13
5. Co to są metody?	15
6. Definiowanie metod	15
7. Zasięg metody	16
8. Przekazywanie typów obiektowych przez moduły	18
9. Definiowanie pól lub metod prywatnych	20
10. Na czym polega rozszerzanie obiektów?	21
11. Co wynika z dziedziczenia metod statycznych?	23
11.1. Sposób wiązania metod statycznych z wywołującymi je zmiennymi	23
11.2. Sposób wywoływania metod dziedziczonych	24
12. Co to są metody wirtualne i polimorfizm?	24
13. Zasady zgodności typów obiektowych	25
14. Korzyści ze stosowania obiektów polimorficznych	27
15. Definiowanie metod wirtualnych	28

16.	<i>Znaczenie słowa constructor przy definiowaniu typu obiektowego</i>	30
17.	<i>Rola konstruktora</i>	31
17.1.	<i>Reprezentacja obiektów w pamięci operacyjnej komputera</i>	31
17.2.	<i>Co zawiera tablica metod wirtualnych VMT?</i>	32
18.	<i>Automatyczna kontrola wywołań metod wirtualnych</i>	34
19.	<i>Podejmowanie decyzji: procedura czy metoda?</i>	40
20.	<i>Budować metodę wirtualną czy statyczną?</i>	41

CZĘŚĆ II. PROGRAMOWANIE OBIEKTOWE W TP6.0 – PRZETWARZANIE OBIEKTÓW DYNAMICZNYCH

21.	<i>Co to są obiekty dynamiczne?</i>	43
22.	<i>Alokacja i inicjowanie obiektów dynamicznych za pomocą procedury New</i>	44
23.	<i>Zwalnianie pamięci przydzielonej obiektom dynamicznym</i>	46
24.	<i>Co to jest destruktor?</i>	47
25.	<i>Mechanizm zwalniania pamięci za pomocą Dispose z konstruktorem jako drugim parametrem wywołania</i>	49
26.	<i>Procedury i funkcje w bibliotece TP6.0 związane z przetwarzaniem obiektów</i>	54

CZĘŚĆ III. ZMIANY W PROGRAMOWANIU OBIEKTOWYM W WERSJI TP7.0

27.	<i>Dyrektywa public</i>	59
28.	<i>Słowo kluczowe inherited</i>	59
29.	<i>Metody dynamiczne i Tablica Metod Dynamicznych (DMT)</i>	61
29.1.	<i>Metody dynamiczne</i>	61
29.2.	<i>Tablica Metod Wirtualnych i Tablica Metod Dynamicznych</i>	62

Dodatek A

	<i>BAZA DANYCH NA OBIEKTACH</i>	65
1.	<i>Wprowadzenie</i>	65

	5
2. <i>Kolekcja</i>	66
3. <i>Typ TCollection</i>	67
4. <i>Strumienie</i>	70
5. <i>Typ TStream</i>	71
6. <i>Typ TBufStream</i>	73
7. <i>Tworzenie kolekcji i strumienia</i>	74
8. <i>Przykład bazy danych</i>	76
Dodatek B	
KRÓTKI PRZEGLĄD RÓŻNIC JĘZYKA TP7.0 W STOSUNKU DO TP6.0	82
1. <i>Dwa zintegrowane środowiska</i>	82
1.1. <i>Nowe cechy środowiska</i>	82
2. <i>Nowe dyrektywy kompilatora</i>	83
3. <i>Parametry procedur lub funkcji deklarowane jako stałe</i>	84
4. <i>Nowe procedury i funkcje w module System</i>	85
5. <i>Rozszerzenia języka dotyczące stosowania parametrów o nie ustalonej liczbie elementów</i>	88
5.1. <i>Tablice o nie ustalonej liczbie elementów</i>	88
5.2. <i>Parametry łańcuchowe o nie określonej długości łańcucha</i>	91
Dodatek C	
PODZIAŁ PAMIĘCI PODCZAS URUCHAMIANIA PROGRAMU NAPISANEGO W JĘZYKU TURBO PASCAL (PLIKU *.EXE) Z POZIOMU SYSTEMU OPERACYJNEGO DOS	93
Indeks	97

Przedmowa

Niniejszy podręcznik powstał z inspiracji studentów Politechniki Wrocławskiej, dla których autorka prowadziła wykłady dotyczące m.in. programowania zorientowanego obiektowo w języku Turbo Pascal. Czytelnik, aby mógł korzystać z tego podręcznika, powinien znać programowanie w języku TP6.0 lub TP7.0 bez przetwarzania obiektów, czyli powinien umieć posługiwać się wszystkimi typami prostymi, typami strukturalnymi (tablice, zbiory, rekordy i pliki), typem wskaźnikowym, korzystać z podstawowych modułów standardowych (Crt, DOS, Graph) oraz tworzyć własne podprogramy (procedury i funkcje) i moduły.

Ze względu na to, że filozofia programowania strukturalnego różni się istotnie od programowania obiektowego, to ostatnie stwarza wielu początkującym problemy. Z doświadczenia autorki wynika, że dużą trudność stanowi zrozumienie istoty „późnego wiązania”, tj. ustalania, która implementacja metody o danym nagłówku ma być w danym momencie wykonana. Rozszerzona kompatybilność typów obiektowych i polimorfizm dają dość duże możliwości programiście, warto je więc poznać, by nie programować „mechanicznie”, opierając się na przykładach. Dla tych, którzy zrozumieli istotę dziedziczenia, programowanie obiektowe nie jest większym problemem niż strukturalne.

Prezentowany podręcznik składa się z trzech części:

W *Części I* wprowadzono podstawowe pojęcia związane z programowaniem obiektowym w TP6.0, omówiono definiowanie i przetwarzanie obiektów statycznych, korzystanie z rozszerzonej kompatybilności typów obiektowych. Wyjaśniono również, na czym polega polimorfizm i jak go stosować, a zatem – rolę konstruktora i tablicy metod wirtualnych.

W *Części II* omówiono problemy związane z przetwarzaniem obiektów dynamicznych w TP6.0. Wyjaśniono też rolę destruktorów w prawidłowej gospodarce pamięcią.

W *Części III* pokazano, jakie zmiany dotyczące definicji i przetwarzania obiektów wprowadzono w wersji TP7.0. Najważniejsza ze zmian to wprowadzenie tzw. metod dynamicznych.

Dodatek A zawiera krótkie omówienie tej części biblioteki *TurboVision*, która jest potrzebna do utworzenia bazy danych dla obiektów z możliwością przechowywania jej w pliku dyskowym. Pokazano również przykład bazy danych z zapisem kolekcji obiektów na dysk i jej odczytem. Zasygnalizowano możliwości przetwarzania takiej bazy danych. Celem tego dodatku jest zachęcenie czytelnika do dalszego studiowania tematu oraz pokazanie, że tworzenie takich baz danych na obiektach nie jest zadaniem trudnym.

W *Dodatkowi B* omówiono krótko podstawowe różnice między językiem TP6.0 a TP7.0 ze szczególnym zwróceniem uwagi na parametry podprogramów typu otwartych tablic i otwartych łańcuchów.

W pracy znajduje się wiele prostych przykładów ilustrujących omawiane zagadnienia. Przez dwie części podręcznika przewija się przykład obiektów graficznych, który jest dostarczany z pakietem TP6.0. Zdaniem autorki jest to przykład dobrze ilustrujący główne cechy i możliwości programowania zorientowanego obiektowo, a fakt, iż jest on w pakiecie TP6.0, ułatwia korzystanie z niego. Autorka uzyskała zgodę firmy BORLAND na

zamieszczenie tego przykładu. Są też inne przykłady, zaproponowane przez autorkę, proste, pisane w ten sposób, by dobrze ilustrowały omawiany problem. Przykłady są numerowane zgodnie z numerami rozdziałów, w których występują; jeśli w którymś rozdziale jest więcej przykładów, to są one oznaczane kolejnymi literami alfabetu. Umieszczane są najczęściej na końcu rozdziału. Słowa kluczowe w języku Turbo Pascal są pisane pismem pogrubionym.

Specjalnym znakiem ¶ oznaczono fragmenty, na które należy zwrócić szczególną uwagę; znakiem < – te paragrafy, które można pominąć.

Składam serdeczne podziękowania mgr Hannie Mazur z Centrum Informatycznego Politechniki Wrocławskiej za wnikliwe przeczytanie „surowej wersji” tej pracy. Uwzględnienie Jej uwag przyczyniło się do tego, że niniejszy tekst jest znacznie przystępniejszy.

Dziękuję również studentom II roku IB (w roku akademickim 1994/1995), za inspirację do napisania tej książki oraz dr. inż. Janowi Kwiatkowskiemu za słowa zachęty i umożliwienie wydania tego podręcznika.

Specjalne podziękowania składam firmie BORLAND za zgodę na wykorzystanie w pracy programów „FigureDemo”, „ListDemo” i modułu „Figures” zawartych w „Borland Pascal 6.0 Users Guide” oraz rysunku 8.2 ze strony 161 „Turbo Pascal 7.0, User’s Guide”

Halina Kwaśnicka

CZĘŚĆ I. PROGRAMOWANIE OBIEKTOWE W TP6.0

DEFINIOWANIE I PRZETWARZANIE OBIEKTÓW STATYCZNYCH

1. Główne cechy programowania obiektowego

Programowanie zorientowane obiektowo (OOP ang. *Object Oriented Programming*) jest bardziej strukturalne, bardziej modularne i abstrakcyjne niż programowanie strukturalne. Wyróżnia się trzy główne cechy języków OOP:

- Hermetyzacja (ang. *Encapsulation*) – połączenie rekordu z procedurami i funkcjami operującymi na polach tego rekordu (nazywanych metodami) w nowy typ danych – obiekt.

- Dziedziczenie (ang. *Inheritance*) – definiuje się obiekt, a następnie używa się go do budowania hierarchii obiektów potomnych. Każdy obiekt potomny może dziedziczyć dostęp do pól (charakterystyk) i metod (sposobów działania) – procedur i funkcji – jego wszystkich przodków.

- Polimorfizm (ang. *Polymorphism*) – dana metoda (procedura lub funkcja) o określonej nazwie jest przekazywana przez dziedziczenie (w dół hierarchii obiektów), przy czym każdy obiekt w hierarchii implementuje metodę w sposób odpowiedni dla siebie.

Obiekt jest pojęciem występującym w realnym świecie. Każdy obiekt może być jednoznacznie opisany przez określenie: (1) zespołu charakterystycznych dla niego cech, (2) funkcji jakie dany obiekt spełnia lub sposobów jego zachowania się.

- Obiekty – postrzegane są łącznie jako zespół ich charakterystyk i sposobów ich zachowania się (postępowania, działania).

2. Na czym polega dziedziczenie?

✂ Dobrą metaforą dla opisu mechanizmu *dziedziczenia w programowaniu obiektowym* jest taksonomia w biologii. Naukowiec próbując zaklasyfikować nowy obiekt (np. zwierzę) zadaje sobie pytanie: *Na ile jest on podobny do innych obiektów z ogólnej klasy? Na ile jest różny?* Każda klasa ma zbiór charakterystyk (cech) i zachowań (działań), które ją definiują. Naukowiec będzie przeprowadzał analizę startując od szczytu *drzewa rodowodu*, schodząc w dół po gałęziach, zadając odpowiednie pytania. Najwyższy poziom jest najbardziej ogólny. Ważny jest fakt, że jeśli jakaś charakterystyka (własność) jest zdefiniowana na pewnym poziomie, wszystkie kategorie (klasy) poniżej tego poziomu

zawierają tę charakterystykę (własność) (np. cechą klasy *ssaki* jest: *żyworodność*; *psy*, *koty*, *słonie*, itp. jako podklasy *ssaków* też mają cechę *żyworodność*).

OOP jest procesem budowania drzewa rodowodu dla struktur danych. Dziedziczenie jest mechanizmem, przez który typy danych dziedziczą charakterystyki (pola danych i metody) z prostszych, bardziej ogólnych typów.

2.1. Realizacja dziedziczenia w Turbo Pascalu bez OOP

≈ W Turbo Pascalu *obiekt* jest bardzo podobny do *rekordu*. W rekordzie są połączone jedną nazwą powiązane ze sobą elementy – tzw. *pola*. W środowisku graficznym rekordem mogą być połączone współrzędne X i Y, określające położenie punktu na ekranie (typ rekordowy *Location*, przykład 2.1.a).

O zmiennej typu *Location* możemy mówić, że jest *wystąpieniem* (ang. *instance*) typu *Location*.

≈ Mając tak zdefiniowany typ *Location* możemy na jego pola *X* i *Y* patrzeć na dwa sposoby: kiedy potrzebujemy oddzielnie współrzędne *X* i *Y*, możemy je traktować oddzielnie jako pole *X* i pole *Y* rekordu. Z drugiej strony, jeśli chcemy myśleć o *X* i *Y* jako o współrzędnych łącznie określających punkt na ekranie monitora, to możemy myśleć o nich łącznie jako o zmiennej typu *Location*.

≈ Przypuśćmy, że typ *Point* powinien określać jasność punktu na ekranie, w miejscu zdefiniowanym przez zmienną rekordową *Location*. Definiując typ *Point* jako rekordowy, można dodać pole logiczne (*Boolean*) wskazujące, czy w danym położeniu jest świecący piksel. Zatem w *Point* będą takie same pola *X* i *Y* jak w typie *Location* (przykład 2.1.b) lub można skorzystać ze zdefiniowanego wcześniej typu *Location* (przykład 2.1.c).

Przykład 2.1.

```
(a) type Location = record
        X, Y: Integer;
      end;

(b) type Point = record
        X, Y: Integer;
        Visible: Boolean;
      end;

(c) type Point = record
        Position: Location;
        Visible: Boolean;
      end;
```

2.2. Rozwiązanie problemu dziedziczenia cech w programowaniu obiektowym

Zadajmy sobie pytanie: „Czym punkt na ekranie różni się od położenia na ekranie?” Odpowiedź brzmi: „Punkt jest położeniem, które jest *zaświecone*”. W definicji *punktu*

występuje wprost *położenie*. Ponieważ wszystkie *punkty* muszą zawierać *położenie*, typ *Point* jest *potomkiem* typu *Location*. *Point* dziedziczy wszystko to, co zawiera *Location* i dodaje wszystko to, co jest nowe dla *Point*. Proces ten (w którym jeden typ dziedziczy charakterystyki innego typu) nazywa się dziedziczeniem (ang. *inheritance*), typ dziedziczony nazywa się potomkiem lub typem potomnym (ang. *descendant type*), typ, od którego potomek dziedziczy, nazywa się przodkiem (ang. *ancestor type*).

3. Definiowanie typów obiektowych i typów obiektowych potomnych

Do definicji typu obiektowego służy słowo kluczowe **object**. Obiekt jest strukturą składającą się ze stałej liczby składników. Składnikiem obiektu może być *pole* – zawiera dane określonego typu lub *metoda* (rozdz. 5) – wykonuje pewne operacje na polach obiektu. Deklaracja *pól* polega na wyspecyfikowaniu identyfikatora pola i jego typu, tak jak przy polach rekordu. Deklaracja *metod* jest podobna do deklaracji procedur i funkcji: zawiera nagłówek *metody*, składający się ze słowa kluczowego **procedure**, **function**, **constructor** (rozdz. 16) lub **destructor** (rozdz. 24), nazwy metody oraz ewentualnej listy parametrów. Po nagłówku metody może wystąpić słowo kluczowe **virtual**, które oznacza, że dana metoda jest wirtualna (rozdz. 15). Zasięg definiowanego typu obiektowego to blok, w którym jest on definiowany lub, jeśli jest definiowany w części **interfejs** modułu – moduł i inne programy lub moduły wykorzystujące dany moduł. Definicję typu obiektowego kończy słowo kluczowe **end**.

Definicja typu obiektowego ma postać:

```
type nazwa_typu = object
    lista_deklaracji_pól;                {jak pola rekordu}
    lista_deklaracji_metod; {nagłówki procedur i funkcji}
end;
```

✚ W definicji typu obiektowego, przed kończącym definicję słowem kluczowym **end** musi wystąpić średnik, inaczej będzie błąd kompilacji: *oczekiwany średnik*.

Typ obiektowy może być definiowany jako samodzielny, niezależny typ lub jako typ potomny istniejącego (zdefiniowanego wcześniej) typu obiektowego. Wtedy po słowie kluczowym **object** podaje się nazwę typu przodka ujętą w nawiasy okrągłe.

W module można definiować pewne pola i metody typu obiektowego dostępne tylko wewnątrz tego modułu; służy do tego celu słowo kluczowe **private** (rozdz. 9).

W naszym przykładzie z grafiki, obiekty mogłyby być zdefiniowane jak w przykładzie 3.a; typy zdefiniowane w przykładzie 3.b mogą być użyteczne dla bazy danych o różnych przesyłkach.

✚ Na tym etapie, w definicji typów obiektowych nie uwzględniamy metod. Pojęcie metod, następnie metod wirtualnych, konstruktora i destruktoru będzie stopniowo wprowadzane w dalszej części pracy.

Przykład 3.

```

(a) type Location = object                                {typ nie będący potomkiem}
      X,Y:Integer;                                       {dwa pola typu całkowitego}
    end;
    Point = object(Location) {Point jest potomkiem typu Location}
      Visible:Boolean;
    {dziedziczy pola X i Y oraz ma własne pole Visible typu logicznego}
    end;

(b) type TOpakowanie = (WKopercie,Miekkie,Szttywne);      {typ wyliczeniowy}
    TRodzajKonteneru = (Zwykly,Hermetyczny,Chlodnia);
    TList = object {typ bazowy dla 3 niżej zdefiniowanych typów}
      Nazwisko,Imie,Adres:String[30];
      RodzajPrzesylki:string[10];
      Waga:Real;
    end;
    TPakiet = object(TList) {typ TPakiet jest potomkiem typu TList}
      Opakowanie:TOpakowanie;
    {dziedziczy pola Nazwisko, Imie, Adres, RodzajPrzesylki z typu TList
    oraz ma własne pole Opakowanie}
    end;
    TPaczka = object(TList) {typ TPaczka jest potomkiem TList}
      Szerokosc,Wysokosc,Dlugosc:Real;
    end;
    TKontener = object(TList) {potomek TList}
      RodzajKonteneru:TRodzajKonteneru;
    end;

```

Wszystkie powyższe typy nie zawierają metod. Typ *Location* jest przodkiem, *Point* jest potomkiem. Wszystkie typy dziedziczące z typu *Location* nazywane są typami potomnymi, lecz *Point* jest potomkiem bezpośrednim (ang. *immediate descendants*). Odwrotnie, *Location* jest bezpośrednim przodkiem (ang. *immediate ancestor*) dla typu *Point*. Bezpośredni potomek typu *Point* będzie dziedziczył zarówno pola typu *Location* jak i dodane pola w typie *Point*, jednak dla typu *Location* nie będzie to już bezpośredni, lecz pośredni potomek. Definiowanie typów potomnych może przebiegać praktycznie nieskończenie. Można zdefiniować typ potomny typu *Point*, typ potomny tego typu potomnego, itd. Wiele aplikacji opiera się na budowie hierarchicznej struktury obiektów.

- Obiekty, jak wynika z powyższych przykładów, są bliskie rekordom. Istnieją oczywiście różnice, przykładowo, pola *X* i *Y* z typu *Location* nie występują wprost wewnątrz typu *Point*, lecz *Point* zawiera te pola, przez dziedziczenie. Możemy mówić o wartości *Point.X* tak, jak o wartości *Location.X*.
- Typ obiektowy (podobnie jak podkartoteki w DOSie) może mieć dowolną liczbę bezpośrednich potomków, ale tylko jednego bezpośredniego przodka (patrz przykład 3.b).

3.1. Deklaracja zmiennych typu obiektowego

Zmienne, czyli konkretne egzemplarze (ang. *instance*) typów obiektowych są deklarowane tak, jak inne zmienne, statyczne lub dynamiczne.

Przykład 3.1.

```

type PointPtr = ^Point;
    Point = object
        X,Y:Integer;
        Visible:Boolean;
    end;
var StatPoint:Point;                                {zmienna statyczna }
    DynaPoint:PointPtr;    {zm. dynamiczna, należy jej przydzielić pamięć}

```

3.2. Deklarowanie zmiennych z wartością początkową typu obiektowego

Zmienne z wartością początkową typu obiektowego (ang. *typed constants*) deklaruje się tak, jak dla typu rekordowego. Wartości nadaje się tylko polom danych, składniki będące metodami (patrz rozdz. 6) pomija się.

☛ Zmienne z wartością początkową typów obiektowych zawierających metody wirtualne nie muszą być inicjowane przez wywołanie konstruktora (patrz rozdz. 16). W tym przypadku zmienna inicjowana jest automatycznie przez kompilator (patrz przykład 17.2.b, na str. 34).

Przykład 3.2.

```

type TPoint = object
    X,Y:Integer;                                {współrzędne punktu}
    Visible:Boolean;
end;
TRec = object(TPoint)
    a,b:Integer;                                {długości dwóch boków prostokąta}
end;
{zmienne z wartością początkową (ang. typed constant: )}
const p:TPoint = (X:200;Y:100;Visible:True);
    r:TRec = (X:100;Y:50;a:150;b:200;Visible:True);

```

4. W jaki sposób są dostępne pola obiektu?

Język TP6.0 dopuszcza taki sam dostęp do pól obiektu, jak do pól rekordu, tzn. za pomocą instrukcji wiążącej **with** lub pełnej nazwy kwalifikowanej (przykład 4.a).

☛ Zanim zaczniemy operować polami obiektu, przypomnijmy krótko, w jaki sposób odbywało się inicjowanie zmiennej rekordowej (nadanie wartości początkowych polom rekordu). Załóżmy, że mamy zainicjować zdefiniowany poniżej rekord typu *Location*. Programista użyłby nazwy kwalifikowanej lub instrukcji wiążącej **with** aby przypisać wartości polom X i Y (przykład 4.b). Jest to sposób skuteczny, ale związany z jedną wyspecyfikowaną zmienną rekordową *MyLocation*. Jeśli zajdzie potrzeba inicjowania więcej niż jednej zmiennej typu *Location*, konieczne będzie zastosowanie wielu instrukcji **with** wykonujących dokładnie tę samą czynność. Naturalnym, kolejnym krokiem powinno

być zbudowanie procedury inicjującej, która uogólnia instrukcję **with** tak, by obejmowała każdą zmienną typu *Location*, przekazywaną jako parametr (przykład 4.c).

Dla typu obiektowego mamy identyczne możliwości jak dla rekordu (przykład 4.d, choć, jak będzie to pokazane w następnym rozdziale, takie odwoływanie się do pól obiektu nie jest zalecane). Można zdefiniować procedurę uogólniającą instrukcję **with**, będzie ona specyficzna tylko dla zmiennych jednego typu, np. *Point*. Należy zatem kontrolować, by używać jej tylko do zmiennych tego typu, dla którego była budowana¹. Byłoby wygodniej, gdyby istniał sposób umożliwiający połączenie typu rekordowego i kodu procedury (ang. *code*) w jedną całość.

Przykład 4.

```
(a) type Point = object                                {typ obiektowy}
      X,Y:Integer;
      Visible:Boolean;
    end;
    var MyPoint:Point;                                {zmienna obiektowa}
```

Możliwe są następujące odwołania do pól obiektu:

```
MyPoint.X:=17; MyPoint.Y:=42; MyPoint.Visible:=True;    {pola obiektu}
with MyPoint do                                          {instrukcja wiążąca}
begin
  X:=17; Y:=42; Visible:=True
end;
```

```
(b) type Location = record                             {typ rekordowy}
      X,Y:Integer;
    end;
    var MyLocation:Location;                            {zmienna rekordowa!}
    MyLocation.X:=17; MyLocation.Y:=42;                 {pola rekordu}
    with MyLocation do
    begin
      X:=17; Y:=42
    end;
```

```
(c) procedure InitLocation(var zmienna:Location;NewX,NewY:Integer);
    begin                                              {procedura uogólniająca with dla zm. rekordowych}
    with zmienna do
      begin
        X:=NewX; Y:=NewY
      end
    end;
```

¹ Dopasowanie typów parametrów procedury i danych zastosowanych do niej jest zadaniem programisty. Choć ograniczenia na zgodność typów są tu pomocne, mogą one tylko wskazać na niemożność połączenia procedury z danymi ze względu na nieodpowiedni typ. W niektórych sytuacjach, kompilator nie wykryje błędnego zastosowania danych, np. w graficznych procedurach *MoveTo(x,y)* czy *MoveRel(Dx,Dy)*. Tu programista może mylnie przesunąć położenie pisaka zamiast do wskazanego punktu, do punktu przesuniętego o zadane wartości. Takich przykładów można wskazać więcej.


```
(d) procedure InitP(var zmienna:Point;NewX,NewY:Integer;NVis:Boolean);
begin      {procedura uogólniająca with dla zmiennych obiektowych}
with zmienna do
  begin
    X:=NewX; Y:=NewY; Visible:=NVis
  end
end;
```

- ☛ Należy zwrócić uwagę, że pola dziedziczone są dostępne tak, jak pola definiowane wewnątrz danego obiektu. Mimo, że X i Y są definiowane w typie Location, w zmiennych typu Point można się do nich odwoływać tak, jakby były tam definiowane: MyPoint.X:=17;

5. Co to są metody?

Ostatnie stwierdzenie o potrzebie łączenia razem typu rekordowego i kodu procedury w jedną całość prowadzi wprost do pojęcia *metody*.

- ☛ Metoda jest to procedura lub funkcja, zadeklarowana wewnątrz obiektu i ściśle ograniczona do tego obiektu. Metoda operuje na polach obiektu, jest objęta niewidocznym **with**, powodującym to, że zmienne (ang. *instances*) typu, w którym definiowana jest dana metoda, są dostępne wewnątrz tej metody.

Pomimo że dopuszczalny jest dowolny dostęp do pól obiektu (instrukcja **with**, nazwa kwalifikowana), korzystanie z takiego dostępu nie jest zalecane. W programowaniu obiektowym wymaga się, by pola zmiennych obiektowych były wykorzystywane tylko przez specjalne procedury zdefiniowane wewnątrz danego typu obiektowego, nazywane metodami. Kiedy tylko jest to możliwe, dostęp do pól zmiennych obiektowych uzyskuje się przez wywoływanie *metod typu obiektowego*

- ☛ Pola obiektu są tym, co obiekt wie, metody są tym, co obiekt robi. W programowaniu obiektowym ważne jest, by programista w czasie pisania programu myślał o danych (polach z danymi) i metodach (procedurach i funkcjach przetwarzających te dane) łącznie.

6. Definiowanie metod

Wewnątrz definicji typu obiektowego deklaruje się nagłówek procedury lub funkcji, działającej jako metoda. Pełną definicję metody podaje się poza definicją typu obiektowego. Przy podawaniu pełnej definicji metody stosuje się nazwę kwalifikowaną, tzn. nazwa metody jest poprzedzona nazwą typu. Podobnie jak w części **interface** modułu, deklaracja metod wewnątrz typu obiektowego mówi jedynie, co metoda robi, nie mówi natomiast jak metoda to robi. To, jak metoda działa jest definiowane na zewnątrz definicji typu obiektowego.

Przykład 6.

```

type Location = object
    X,Y:Integer;
    procedure Init(InitX,InitY:Integer);
    function GetX:Integer;
    function GetY:Integer;
end;

procedure Location.Init(InitX,InitY:Integer);
begin
    X:=InitX; Y:=InitY;
end;

function Location.GetX:Integer;
begin
    GetX:=X;
end;

function Location.GetY:Integer;
begin
    GetY:=Y;
end;

```

Mając zdefiniowaną metodę `Location.GetX` można zdefiniować „zwykłą” procedurę o nazwie `GetX`. Ta „zewnętrzna” procedura nie będzie miała powiązania z typem obiektem `Location`. Definicja innej, zewnętrznej procedury o tej samej nazwie co metoda, jakkolwiek poprawna, spowodowałaby mniejszą czytelność programu.

7. Zasięg metody

Należy zauważyć, że w żadnej definicji metody nie było wprost konstrukcji **with** zmienna obiektowa **do ...**. Metody danego typu obiektowego mają swobodny dostęp do pól danych zmiennych tego typu. Pomimo że w źródłowym kodzie programu ciało metody jest oddzielone od pól danych, mają one taki sam zasięg. Z tego właśnie wynika fakt, że metoda typu `Location` może zawierać odwołanie `GetY:=Y` bez kwalifikatora do `Y`. Możliwe jest to dlatego, że `Y` należy do typu obiektowego, którego metoda jest wywoływana. Kiedy zmienna obiektowa wywołuje metodę, wykonywana jest ukryta instrukcja łącząca zmienną i metodę, efekt tego jest taki sam jak wykonania następującej instrukcji:

```
with myself do metoda
```

Ta ukryta instrukcja **with** dokonuje się przez przekazywanie, przy każdorazowym wywołaniu metody, ukrytego parametru metody. Parametr ten nazywa się *Self*. Jest to pełny 32-bitowy wskaźnik do tej zmiennej, która spowodowała wywołanie metody. Metoda `GetY` należąca do typu `Location` jest równoważna zapisowi:

```

function Location.GetY(var Self:Location):Integer;
begin
    GetY:=Self.Y;
end;

```

W tym przykładzie, zapis `(var Self:Location)` nie jest syntaktycznie poprawny, wystąpi

błąd: *Duplicate identifier (Self)*, ponieważ jest wykonywana niejawnie instrukcja wiążąca. Programista może nie być świadomy używania parametru *Self*. Parametr ten jest zawsze przekazywany bez udziału programisty. Użycie wprost parametru *Self* jest dopuszczalne, aczkolwiek należy unikać takiej potrzeby. Wskazaniem do użycia parametru *Self* może być konflikt nazw w metodzie, należy wtedy użyć identyfikatora parametru *Self* do nazwy kwalifikowanej pól należących do zmiennej obiektowej wywołującej metodę (nie ma innego sposobu).

Przykład 7.

```

program ParametrSelf;
uses Crt;
type TT1 = object
    X,Y:Integer;                                {pola typu TT1}
    procedure pisz(znak:char);                  {wypisz znak w miejscu (X,Y)}
    procedure Init(NoweX,NoweY:Integer);        {nadaj wartości polom}
end;
TT2 = Object
    X,Y:Integer; {pola typu TT2, nie mające związku z typem TT1}
    procedure ZmienZnak(var VV1:TT1);          {wypisuje zadany znak w}
end; {miejscu o współrzędnych równych polom zmiennej typu TT1}

procedure TT1.pisz(znak:char);
begin
    GoToXY(X,Y); Writeln(znak);
end;

procedure TT1.Init(NoweX,NoweY:Integer);
begin
    X:=NoweX; Y:=NoweY;
end; {Init }

procedure TT2.ZmienZnak(var VV1:TT1);
begin
    with VV1 do                                {wymagany jest dostęp do pól X,Y zmiennej VV1}
    begin
        Self.X:=X; Self.Y:=Y;                  {nadajemy wartości polom zm. VV2 (typu TT2)}
    end;    {X i Y po prawej stronie instr. przypisania, to pola zm. VV1}
    GoToXY(X,Y);
    Writeln('2');                               {wypisz cyfrę 2}
end; { ZmienZnak }

var V11:TT1;
    V21:TT2;
    a,b:byte;

begin                                {programu głównego}
    ClrScr;
    Write('Podaj dwie liczby - wspolrzedne ekranu: ');
    Readln(a,b);
    ClrScr;
    V11.Init(a,b);                        {ustal wartości pól zmiennej V11 jako wczytane a,b}
    V11.pisz('A');                        {wypisz literkę A w miejscu o wsp. a,b}
    Repeat until Keypressed;              {odczekaj na wciśnięcie klawisza}

```

```
V21.ZmienZnak(V11);    {wypisz 2 w miejscu wskazanym przez pola zm. V11}
Readln;                {odczekaj na wciśnięcie klawisza <ENTER>}
end.
```

- ☛ Jedną z konsekwencji jednakowego zasięgu obiektów i ich metod jest to, że nazwy parametrów formalnych metod nie mogą być identyczne z nazwami pól danych obiektu. Jest to zgodne z zasadą dotyczącą zasięgu identyfikatorów i zakazem dublowania się identyfikatorów w Turbo Pascalu: gdy nazwa parametru formalnego zwykłej procedury jest taka sama jak nazwa parametru aktualnego tej procedury, wystąpi błąd.

8. Przekazywanie typów obiektowych przez moduły

Często stosuje się definiowanie typów obiektowych w modułach. Typy obiektowe są definiowane w części **interface**, zaś metody typów obiektowych („ciała” metod) definiowane są w części **implementation**. Moduły mogą mieć zdefiniowane swoje prywatne typy obiektowe w części **implementation**; dla takich typów są aktualne te same ograniczenia co dla każdego innego typu zdefiniowanego w tej części modułu. Typ obiektowy zdefiniowany w części **interface** modułu może mieć typy potomne w części **implementation**. Jeśli moduł B korzysta z modułu A, moduł B może zawierać definicje typów potomnych dla typów zdefiniowanych w części **interface** modułu A (przekazywanych do modułu B).

Typy i metody opisywane wcześniej mogą być zebrane w module *Points*.

Przykład 8.

(a){Zamieszczony w pakiecie TP 6.0}

```
unit Points;
interface
uses Graph;
type Location = object
    X,Y:Integer;
    procedure Init(InitX,InitY:Integer);
    function GetX:Integer;
    function GetY:Integer;
end;
Point = object(Location)
    Visible:Boolean;
    procedure Init(InitX,InitY:Integer);
    procedure Show;
    procedure Hide;
    function IsVisible:Boolean;
    procedure MoveTo(NewX,NewY:Integer);
end;
implementation
procedure Location.Init(InitX,InitY:Integer);
begin
    X:=InitX; Y:=InitY;
```

```

end;
function Location.GetX:Integer;
begin
  GetX:=X;
end;
function Location.GetY:Integer;
begin
  GetY:=Y;
end;
procedure Point.Init (InitX,InitY:Integer);
begin
  Location.Init (InitX,InitY); Visible:=False;
end;
procedure Point.Show;
begin
  Visible:=True;
  PutPixel (X,Y,GetColor);
end;
procedure Point.Hide;
begin
  Visible:=False;
  PutPixel (X,Y,GetBkColor);
end;
function Point.IsVisible:Boolean;
begin
  IsVisible:=Visible;
end;
procedure Point.MoveTo (NewX,NewY:Integer);
begin
  Hide;
  Location.Init (NewX,NewY);
  Show;
end;
end.

```

(b) Przykładowy program wykorzystujący powyższy moduł:

```

Program MakePoints;
uses Graph,Points;
var APoint:Point;                                {Point - typ zdefiniowany w module}
begin
  ...                                           {w tej części winno być otwarcie grafiki}
  APoint.Init (151,82);                       {wywołanie metody zdefiniowanej w module}
  APoint.Show;                                {wywołanie metody zdefiniowanej w module}
  readln;                                     {zatrzymanie programu, by zobaczyć świecący punkt}
  APoint.MoveTo (163,101);                     {wywołanie metody zdefiniowanej w module}
  Readln;                                     {zatrzymanie programu}
  APoint.Hide;                                {wywołanie metody zdefiniowanej w module}
  ...                                           {zamknięcie grafiki}
end.

```

✂ Dodając metody do modułu *Point* powiększamy jego kod źródłowy. Program łączący

(linker) TP6.0 nie włącza do kodu wynikowego programu kodów tych metod z modułów, które nie są w programie wykorzystywane. Tworząc moduł z metodami można nie obawiać się tego, że zbędnie powiększa się kod wykorzystującego go programu.

9. Definiowanie pól lub metod prywatnych

Niekiedy nie chcemy pewnej części typu obiektowego eksportować poza moduł, np. kilku metod lub pól z danymi. Chcemy umożliwić innemu programiście korzystanie z typu, ale bez możliwości manipulowania wprost pewnymi jego polami. Do tego celu służy dyrektywa **private**.

☛ Prywatne metody i pola są dostępne jedynie wewnątrz modułu, w którym zadeklarowany jest dany obiekt. Prywatne pola i metody są deklarowane bezpośrednio po deklaracji pól i metod ogólnodostępnych (tzw. publicznych).

Pełna syntaktyczna postać deklaracji obiektu jest następująca:

```
type NewObject=object(nazwa_typu_poprzednika)  {nie musi być poprzednik}
    Pola_1;      {pola publiczne}
    Metody_1;    {metody publiczne}
    private      {nie musi wystąpić private}
    Pola_2;      {pola prywatne}
    Metody_2;    {metody prywatne}
end;
```

Przykład 9.

```
program TestPrivate;                                {program główny}
uses Proba;
type Point=object(Location)                        {typ potomny typu definiowanego w module}
    color:byte;
end;
    Circle=object(Point)
        r:byte;
    end;
var TestPt:Location;
    ACircle:Circle;
begin
    writeln(' WYDRUKI Z PROGRAMU GLOWNEGO');
    writeln('zm_mod.x= ',zm_mod.x); {gdyby drukować zm_mod.y byłby błąd...}
    TestPt.x:=10;                      {... bo jest to pole prywatne}
    ACircle.x:=30;
    writeln(ACircle.x);                 {wypisze 30; Acircle.y - byłby błąd}
    writeln(TestPt.x);                  {wypisze 10}
    writeln;writeln('Wcisnij ENTER by zakonczyc');
readln;
end.

unit proba;
interface
uses crt;
```

```

type Location=object
    x:integer;
    private
        y:integer;           {pole prywatne!}
    end;
var zm_mod:Location;
implementation
begin
    clrscr;
    zm_mod.x:=11;
    zm_mod.y:=22;
    writeln('WYDRUKI Z MODULU');
    writeln('zm_mod.x = ',zm_mod.x);           {wypisze 11}
    writeln('zm_mod.y = ',zm_mod.y);           {wypisze 22}
end.

```

Wyniki pojawiające się na ekranie po uruchomieniu powyższego programu:

```

WYDRUKI Z MODULU
zm_mod.x=11
zm_mod.y=22
WYDRUKI Z PROGRAMU GŁÓWNEGO
zm_mod.x=11
30
10
Wcisnij ENTER by zakonczyc

```

10. Na czym polega rozszerzanie obiektów?

Spójrzmy na standardową procedurę *WriteLn*. Umożliwia ona przekazywanie parametrów różnych typów, np.:

```

WriteLn(ZmiennaZnakowa);
WriteLn(ZmiennaCalkowita);
WriteLn(ZmiennaRzeczywista);

```

Turbo Pascal bez OOP nie daje możliwości samodzielnego budowania takich procedur. OOP rozwiązuje ten problem przez dziedziczenie. Kiedy tworzymy typ potomny, metody typu poprzednika są dziedziczone; jeśli jest taka potrzeba mogą one być przesłaniane (ang. *overriden*). Przesłaniając dziedziczoną metodę definiuje się nową z tą samą nazwą co dziedziczona, lecz z różnym „ciałem” (jeśli trzeba) i różnym zbiorem parametrów.

Poniższy przykład (rysowanie koła na ekranie) pokazuje wykorzystanie potomka typu *Point*.

Przykład 10.

{ Typ *Point* zdefiniowany jest w przykładzie 8.a }

```

type Circle = object(Point)           {potomek typu Point}
    Radius:Integer;                   {dodatkowe pole - promień}
    procedure Init(InitX,InitY,InitRadius:Integer);
    procedure Show;
    procedure Hide;

```

```

        procedure Expand(ExpandBy:Integer);
        procedure MoveTo(NewX,NewY:Integer);
        procedure Contract(ContractBy:Integer);
    end;

procedure Circle.Init(InitX,InitY,InitRadius:Integer);
begin
    Point.Init(InitX,InitY);
    Radius:=InitRadius;
end;

procedure Circle.Show;
begin
    Visible:=True;
    Graph.Circle(X,Y,Radius);
end;

procedure Circle.Hide;
var
    TempColor:Word;
begin
    TempColor:=Graph.GetColor;
    Graph.SetColor(GetBkColor);
    Visible:=False;
    Graph.Circle(X,Y,Radius);
    Graph.SetColor(TempColor);
end;

procedure Circle.Expand(ExpandBy:Integer);
begin
    Hide;
    Radius:=Radius+ExpandBy;
    if Radius<0 then Radius:=0;
    Show;
end;

procedure Circle.Contract(ContractBy:Integer);
begin
    Expand(-ContractBy);
end;

procedure Circle.MoveTo(NewX,NewY:Integer);
begin
    Hide;
    X:=NewX;
    Y:=NewY;
    Show;
end;

```

W tym przykładzie koło jest „grubym punktem”. Typ Circle ma wszystkie pola typu Point (położenie X i Y, Visible, bo je dziedziczy) plus dodatkowo pole promień (Radius). Ponieważ Circle ma nowe pole, należy je inicjować, zatem potrzebna jest nowa metoda Init. Można w niej przywołać metodę poprzednika (Point.Init). Wywołując tu metodę poprzednika zapewniamy sobie, że wszelkie ewentualne zmiany w typie przodka będą uwzględnione w typie potomnym, bez powtarzania instrukcji nadającej wartość polom X i Y.

Zamiast procedury PutPixel do pokazania „grubego punktu”, czyli koła, należy użyć procedury rysowania koła z modułu Graph, zatem potrzebna jest nowa metoda Show itd. Odwołanie Graph.Circle informuje kompilator, że dotyczy ono procedury rysowania koła z modułu GRAPH.TPU a nie nazwy typu obiektowego.

11. Co wynika z dziedziczenia metod statycznych?

W powyższym przykładzie łatwo zauważyć, że jedna z metod typu *Point*, a mianowicie *MoveTo* jest przesłonięta w typie *Circle*. Chociaż wygląda identycznie jak w *Point*, nie odwołuje się do dodatkowego pola *Radius*, ani nie korzysta z innych bibliotek poza GRAPH.TPU. Metody *GetX* i *GetY* z typu *Location* są dziedziczone bez przesłaniania. Napisaliśmy zatem drugi raz identyczną metodę, tylko z innym kwalifikatorem metody (była już metoda *Point.MoveTo*, teraz zdefiniowana została *Circle.MoveTo*).

Wszystkie metody, które dotychczas tworzyliśmy, to metody statyczne.

☛ Termin *statyczne* jest używany do nazywania metod nie będących *metodami wirtualnymi*. Metody *wirtualne*, definiowane z użyciem słowa kluczowego **virtual** są wyjaśnione w rozdz. 12.

Zanim przejdziemy do metod wirtualnych i wyjaśnienia w jaki sposób pomogą nam uniknąć ponownego definiowania tej samej metody z innym nagłówkiem, spróbujmy dobrze rozpoznać problem, później łatwiej będzie zrozumieć jego rozwiązanie.

Na czym polega problem: Dziedziczona (czyli skopiowana dokładnie) przez typ *Circle* metoda *MoveTo* z typu *Point* nie będzie pracować prawidłowo, kiedy zostanie wywołana dla zmiennej obiektowej typu *Circle*. Kiedy zmienna typu *Circle* wywoła *MoveTo* zdefiniowaną w typie *Point* (dziedziczną), na ekranie będzie przesuwany punkt a nie koło (metoda ta spowoduje ukrycie punktu przez wywołanie metody *Point.Hide*, przesunięcie punktu – to ma być wykonywane jednakowo dla punktu i koła, wyświetlenie na ekranie punktu zamiast koła przez wywołanie *Point.Show*). Kiedy zmienna typu *Circle* wywoła metodę *MoveTo* zdefiniowaną w obrębie własnego typu, spowoduje ukrycie koła przez wywołanie metody *Circle.Hide*, po czym ponowne narysowanie go w przesuniętym miejscu przez wywołanie metody *Show* zdefiniowanej wewnątrz własnego typu (*Circle.Show*).

11.1. Sposób wiązania metod statycznych z wywołującymi je zmiennymi

Wywołanie metod statycznych jest ustalane przez kompilator, w czasie kompilacji programu. Kompilując metody typu *Point*, kompilator najpierw napotyka *Point.Show* i *Point.Hide*, skompilowane kody obu umieszcza w segmencie kodów. Dalej napotyka na metodę *Point.MoveTo*, która wywołuje obie zdefiniowane wcześniej metody: *Point.Show* i *Point.Hide*. Tak jak zawsze, w miejscu wywoływania procedury, kompilator umieszcza adres kodu wywoływanej procedury (tzn. adres, pod którym w segmencie kodów jest umieszczony kod wywoływanej procedury). Zawsze, kiedy wywoływana jest metoda *Point.MoveTo*, spowoduje to przywołanie kodów *Point.Show* i *Point.Hide*. Zatem, jeśli typ

Circle odziedziczy tę metodę z typu *Point*, będzie mógł stosować tę metodę dokładnie tak, jak została ona skompilowana, czyli z *Point.Show* i *Point.Hide*.

Przykład 11.1.

To, co typ Circle mógłby odziedziczyć z typu Point przedstawiałoby się następująco:

```
procedure Point.MoveTo (NewX, NewY: Integer);
begin
    Hide;                                {Wywołanie Point.Hide}
    X:=NewX;
    Y:=NewY;
    Show;                                {Wywołanie Point.Show}
end;
```

Wewnątrz metody MoveTo będą wywoływane Point.Hide i Point.Show, pomimo iż typ Circle ma zdefiniowane własne metody Hide i Show. Tak długo, dopóki Point.MoveTo będzie wywoływać Point.Hide i Point.Show, nie będzie mogła być dziedziczona przez Circle. Musi być przesłonięta przez identyczną, ale wywołującą Circle.Hide i Circle.Show.

11.2. Sposób wywoływania metod dziedziczonych

Kompilator, napotykając na wywołanie metody przez daną zmienną, szuka definicji tej metody w typie zmiennej wywołującej metodę; jeśli metoda o wywoływanej nazwie nie jest definiowana wewnątrz tego typu, kompilator będzie jej szukał u bezpośredniego poprzednika. Jeśli i tam nie znajdzie, będzie szukał u kolejnego poprzednika, itd., aż dojdzie do typu pierwotnego, który nie ma już poprzednika. Jeśli i tam nie znajdzie, będzie sygnalizował błąd.

- ☛ Należy pamiętać, że kiedy metoda statyczna zostanie znaleziona u poprzednika, stosowana jest zawsze tak, jak została zdefiniowana i skompilowana w typie poprzednika. Jeśli metoda poprzednika wywołuje inne metody, będą to zawsze metody poprzednika, nawet jeśli zostały przesłonięte (zdefiniowane na nowo u potomka).

12. Co to są metody wirtualne i polimorfizm?

Omówione dotychczas metody, to metody statyczne. Są takie, ponieważ kompilator decyduje o wszystkich wywołaniach w czasie kompilacji przez umieszczenie w miejscu wywołania adresu ich kodu. Z tego wynika opisany powyżej problem. Sposobem na jego rozwiązanie może być dynamiczne wywoływanie metod, czyli definiowanie metod jako wirtualnych.

- ☛ *Metody wirtualne* są bardzo ważnym narzędziem dla uogólnienia programowania, nazywanego *polimorfizmem*.

Polimorfizm – z greckiego znaczy „wiele kształtów”. Jest to sposób pozwalający na to, by dana akcja o jednej nazwie, przechodziła w dół wraz z hierarchią obiektów (dziedziczeniem), przy czym każdy obiekt w hierarchii implementuje akcję w sposób

odpowiedni dla niego.

Prosty przykład rysowania i przesuwania po ekranie figur graficznych, opisany uprzednio, dobrze nadaje się do zastosowania polimorfizmu akcji, przez metody wirtualne. Każdy typ obiektowy w naszej hierarchii reprezentuje różne typy figur na ekranie monitora: punkt lub koło. Inaczej mówiąc, mamy możliwość pokazania na ekranie punktu lub koła. Można dalej zdefiniować obiekty reprezentujące inne figury: linię, kwadrat, łuk itp. i można napisać dla nich metody, które pokażą lub przesuną dany obiekt na ekranie. Można powiedzieć, że wszystkie te figury mają coś wspólnego, można je mianowicie wyświetlić na ekranie. To co jest inne dla każdej figury, to sposób ich wyświetlania (*PutPixel*, *Circle*, *Rectangle*, *Line* itp.). Każda z tych figur może być pokazana na ekranie, ale mechanizm jej pokazywania jest dla niej specyficzny.

Różnica między wywoływaniem *statycznych* a *wirtualnych* metod jest różnicą między *decyzją podejmowaną teraz a decyzją późniejszą*. Wywołując metodę statyczną, dajemy kompilatorowi polecenie: „Wiesz czego ja chcę, więc przywołaj to”. Wywołując metodę wirtualną, informujemy kompilator: „Teraz jeszcze nie wiem czego chcę, kiedy będzie czas – zapytaj zmiennej”. Każdy typ figury ma własną implementację metody *Show* i to, która implementacja *Show* ma być wywoływana przez metodę *MoveTo* powinno zależeć od zmiennej wywołującej metodę *MoveTo*. Zatem ustalenie adresu kodu metody *Show* wywoływanej przez metodę *MoveTo* nie może odbywać się w czasie kompilacji: Kiedy *MoveTo* jest kompilowana, żadna decyzja co do wywoływania *Show* nie może być podjęta. Informacja o tym, która implementacja *Show* ma być wywoływana nie jest jeszcze dostępna. Dopiero w czasie wykonywania programu można zapytać, jaka zmienna obiektowa (typu punkt, koło, kwadrat?) wywołuje *MoveTo*. Wiązanie wywołania metody z konkretną metodą (jej implementacją) dokonuje się podczas faktycznego wykonywania wywołanej metody – w czasie uruchamiania programu.

13. Zasady zgodności typów obiektowych

Dziedziczenie zmienia zasady kompatybilności (zgodności) typów. Typ potomny dziedziczy zgodność ze wszystkimi typami poprzednimi. Rozszerzenie zgodności typów rozpatruje się w trzech aspektach:

- między zmiennymi obiektowymi (*object instances*),
- między wskaźnikami na zmienne obiektowe (*pointers to object instances*),
- między parametrami aktualnymi a formalnymi.

☛ We wszystkich trzech aspektach należy zapamiętać, że zgodność jest rozszerzona jedynie od potomków do poprzedników, co oznacza, że każdy potomek może być swobodnie użyty w miejsce typu poprzednika, natomiast nie odwrotnie.

Przykład 13.

Przyjmijmy deklaracje:

```
type LocationPtr = ^Location;
```

```

    PointPtr = ^Point;
    CirclePtr = ^Circle;
var ALocation:Location;
    APoint:Point;
    ACircle:Circle;
    PLocation:LocationPtr;
    PPoint:PointPtr;
    PCircle:CirclePtr;

```

Poniższe instrukcje przypisania są poprawne:

```

ALocation:=APoint;
APoint:=ACircle;
ALocation:=ACircle;

```

Odwrotne przypisania nie są dopuszczalne.

Klasa Location jest podzbiorem klasy Point, a ta jest podzbiorem klasy Circle: $\text{Location} \subseteq \text{Point} \subseteq \text{Circle}$. Typ potomny, przez dziedziczenie zawiera wszystko to, co zawiera typ poprzedni. Dlatego też typ potomny ma rozmiar równy lub zwykle większy niż poprzednik, nigdy mniejszy. Przypisując typ poprzedni typowi potomnemu spowodowalibyśmy, że nie wszystkie pola typu potomnego byłyby ustalone, co jest niebezpieczne i zwykle niedozwolone.

☛ W czasie instrukcji przypisania zmiennych obiektowych, tylko te pola, które są wspólne dla obu typów są kopiowane z obiektu źródłowego do docelowego.

W instrukcji

```
ALocation:=ACircle;
```

przepisane zostaną jedynie pola X i Y , bo one są wspólne dla tych typów.

Podobnie jak dla typów obiektowych, rozszerzona kompatybilność zachodzi również między wskaźnikami na typy obiektowe.

Dla definicji jak w przykładzie 13. dopuszczalne są przypisania:

```

PPoint:=PCircle;
PLocation:=PPoint;
PLocation:=PCircle;

```

Przypisania przeciwne (po lewej stronie znaku przypisania potomek, po prawej – poprzednik) nie są dopuszczalne.

Jako parametry aktualne procedur i funkcji (przy przekazywaniu przez wartość jak i przez zmienną) mogą być zastosowane obiekty tego samego typu co parametry formalne oraz każdego typu potomnego. Mając nagłówek metody:

```
procedure DragIt (Parametr:Point);
```

parametrem aktualnym (parametrem wywołania) może być zarówno typ *Point*, jak i *Circle*, ale nie *Location*. *Parametr* może być parametrem przekazywanym przez zmienną (ze słowem **var**); zachodzi wtedy ta sama reguła zgodności.

☛ Należy pamiętać o istotnej różnicy między przekazywaniem parametrów przez wartość i przez zmienną. Przy przekazywaniu parametru przez zmienną, przekazywany jest adres parametru aktualnego, podczas gdy przez wartość – tworzy się zmienna lokalna, do której kopiowany jest parametr aktualny. Kopia ta obejmuje tylko te pola i metody, które

występują w parametrze formalnym.

Podobnie, kiedy parametrem formalnym jest wskaźnik na typ obiektowy, parametrem aktualnym może być wskaźnik na typ potomny w stosunku do typu wskazywanego przez parametr formalny:

```
procedure Figure.Add(NewFigure:PointPtr);
```

aktualnym parametrem może być typ *PointPtr* lub *CirclePtr*, lecz nie *LocationPtr*.

14. Korzyści ze stosowania obiektów polimorficznych

Rozszerzona kompatybilność typów obiektowych może nasuwać pytanie: jeśli każdy obiekt będący potomkiem typu parametru formalnego procedury (funkcji) może być parametrem aktualnym (wykonania), skąd użytkownik parametru (procedura lub funkcja) wie, jaki obiekt będzie przetwarzać? Rzeczywiście, nie jest to znane w czasie kompilacji, ponieważ może to być każdy potomek; dlatego obiekt taki nazywa się *obiektem polimorficznym*.

■ Obiekty polimorficzne umożliwiają przetwarzanie obiektów, których typ w momencie kompilacji nie jest jeszcze znany.

Żałujemy, że napisaliśmy program będący narzędziem umożliwiającym rysowanie figur na ekranie: punktów, kół, kwadratów, prostokątów, krzywych itp. Jako część tego narzędzia chcemy napisać metodę, która przesuwa figurę po ekranie monitora (kursorem lub wskaźnikiem myszy).

☞ Znany wcześniej sposób, to napisanie oddzielnie procedur przesuwających poszczególne figury: *DragCircle*, *DragSquare*, *DragRectangle*, itd. Koło ma pole danych – promień, prostokąt – dwa boki itd. Sprawny programista, ale nie posługujący się obiektami, poradzi sobie w ten sposób, że każda figura będzie rekordem mającym swoje pole identyfikacyjne *FigureIDTag*, napisze procedurę *DragIt* z parametrem będącym wskaźnikiem tego pola. Wewnątrz *DragIt* sprawdzi jaka figura ma być przetwarzana i w zależności od tego wywoła odpowiednią *DragFigure* stosując *case*:

Przykład 14.

Fragment procedury DragIt:

```
case FfigureIDTag of
  Point :DragPoint;
  Circle:DragCircle;
  Square:DragSquare;
  . . . .
end;
```

W rzeczywistości podejście takie stwarza trochę problemów, np. co będzie jeśli w pewnym momencie dodamy jedną lub kilka figur do programu? Nowa figura nie znajduje się w liście etykiet *case* i nie zostanie rozpoznana. Dodatkowo, budując taki moduł w celu sprzedaży jego wersji skompilowanej, napotykamy problem: Taka procedura może pracować jedynie z obiektami „znanymi” w czasie pisania i kompilacji tego modułu. Użytkownik modułu nie może poszerzyć jego możliwości o figurę nie przewidzianą przez twórcę modułu.

Rozszerzona kompatybilność typów obiektowych, stosowanie obiektów polimorficznych i metod wirtualnych umożliwia inne podejście do budowy metody przesuwania obiektów. Jeśli procedura *DragIt* zdefiniowana w module pracuje z obiektami polimorficznymi, będzie również pracować z każdym obiektem potomnym zdefiniowanym wewnątrz modułu oraz z każdym obiektem zdefiniowanym jako potomny przez użytkownika modułu, na zewnątrz modułu. Jeśli obiekty w module mają metody wirtualne, metody i obiekty z modułu mogą pracować z metodami użytkownika, zdefiniowanymi na zewnątrz modułu. Metody wirtualne, które definiujemy dzisiaj są dostępne dla modułu, który był kompilowany, np. rok temu. Kluczem dla tych możliwości OOP są metody wirtualne.

15. Definiowanie metod wirtualnych

Metody wirtualne tworzy się przez napisanie po nagłówku metody, wewnątrz definiowanego obiektu, słowa kluczowego **virtual**.

- Jeśli w danym obiekcie zdefiniujemy metodę wirtualną o określonej nazwie, to wszystkie metody o tej nazwie definiowane w typach potomnych muszą być wirtualne, inaczej kompilator wskaże błąd. Nagłówki przesłanianych metod wirtualnych muszą odpowiadać sobie pod względem liczby i typów parametrów formalnych oraz typu wyniku, jeśli metodą jest funkcja.

Przykład 15.

(a) Przykłady zdefiniowanych wcześniej typów z metodami wirtualnymi:

```
type Location = object
    X, Y: Integer;
    procedure Init(InitX, InitY: Integer);
    function GetX: Integer;
    function GetY: Integer;
end;

Point = object(Location)
    Visible: Boolean;
    constructor Init(InitX, InitY: Integer);
    procedure Show; virtual;           {metoda wirtualna}
    procedure Hide; virtual;          {metoda wirtualna}
    function IsVisible: Boolean;
    procedure MoveTo(NewX, NewY: Integer);
end;

Circle = object(Point)
    Radius: Integer;
    constructor Init(InitX, InitY: Integer; InitRadius: Integer);
    {metody wirtualne (Show i Hide przesłaniają metody dziedziczone):}
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure Expand(ExpandBy: Integer); virtual;
    procedure Contract(ContractBy: Integer); virtual;
end;
```

Porównując powyższy przykład z przykładem 10 (str. 21) można zauważyć, że w definicji typu Circle nie ma metody MoveTo, która była przy stosowaniu tylko metod statycznych. Teraz typ Circle nie potrzebuje przesłaniać metody MoveTo dziedziczonej z typu Point. Typ Circle może stosować metodę MoveTo dziedziczoną z Point, z serią odwołań do własnych metod (typu Circle), a nie Point jak to było przy statycznych metodach Show i Hide. Program wykorzystujący powyższe typy jest pokazany w przykładzie 18., str. 35.

(b) Program wykorzystujący wirtualną metodę Suma. W programie jest wypisywana suma wartości pól obiektu, przy czym suma ta jest liczona w zdefiniowanej – jako metoda wirtualna – funkcji o nazwie Suma. Metoda ta jest przesłonięta w typie potomnym. Wartość sumy pól jest wypisywana przez wywołanie metody – procedury Wypisz, która jest zdefiniowana tylko w jednym typie. Typ potomny dziedziczy ją w niezmienionej postaci. Przez wywołanie specjalnej metody – konstruktora o nazwie Init (patrz. rozdz. 16) są nadawane wartości polom obiektów.

```

program MetodyWirtualne;
uses Crt;
type Bazowy=object
    X,Y:Integer;
    constructor Init (AX,AY:Integer);
    function Suma:Integer; virtual;
    procedure Wypisz;
end;
Potomny=object(Bazowy)           {dziedziczy wszystko z typu Bazowy}
    A,B:Real;                     {ma 2 dodatkowe pola rzeczywiste}
    constructor Init (AX,AY:Integer;AA,AB:Real);
    function Suma:Integer; virtual;
                                   {przesłonięcie metody Suma}
end;
constructor Bazowy.Init (AX,AY:Integer);
begin
    {   nadanie wartości, przekazanej jako parametry, polom obiektu:   }
    X:=AX; Y:=AY
end;
function Bazowy.Suma:Integer;
    {   liczenie sumy dwóch pól obiektu typu Bazowy   }
begin
    Suma:=X+Y;
end;
procedure Bazowy.Wypisz;
    {   wypisanie sumy pól obiektu, obliczonej w metodzie Suma   }
begin
    writeln('Suma pol obiektu wynosi: ',Suma)           {wywołanie metody Suma}
end;
constructor Potomny.Init (AX,AY:Integer;AA,AB:Real);
begin
    {   nadanie wartości polom dziedziczonym:   }
    Bazowy.Init (AX,AY);                             {można też: X:=AX; Y:=AY}
    A:=AA; B:=AB                                       {nadanie wartości polom własnym}
end;

```

```

end;
function Potomny.Suma:Integer; {liczenie sumy 4 pól obiektu typu Potomny}
begin
    Suma:=Round(X+Y+A+B)          {wynik funkcji musi być typu całkowitego}
end;
var b:Bazowy; p:Potomny;      {deklaracja zmiennych typu Bazowy i Potomny}
begin
    (program główny)
    ClrScr;
    b.Init(6,4);                {nadanie polom zm. b wartości - odpowiednio 6 i 4}
    writeln('Procedura zdefiniowana w typie Bazowy, ',
    'wywołana ze zmiennej typu Bazowy');
    writeln(' powinna wypisac sume liczb 6 i 4',#13#10,#13#10);
    b.Wypisz;                    {wypisanie sumy pól - wartości 10}
    {w powyższym wywołaniu metoda Wypisz wywoła metodę Suma zdefiniowaną w
    typie Bazowy, bo wywołała ją zmienna typu Bazowy}
    p.Init(12,8,6.4,3.3);        {nadanie wartości polom zmiennej p}
    writeln(#13#10,'Procedura zdefiniowana w typie Bazowy, ',
    'wywołana ze zmiennej typu Potomny,');
    writeln('powinna wypisac zaokrągloną sumę liczb 12, 8, 6.4 i ',
    '3.3',#13#10);
    p.Wypisz;                    {wypisanie sumy czterech pól zm. p; wypisze wartość 30}
    {w powyższym wywołaniu metoda Wypisz wywoła metodę Suma zdefiniowaną w
    typie Potomny, bo metodę Wypisz wywołała zmienna typu Potomny}
    readln;
end.

```

16. Znaczenie słowa **constructor** przy definiowaniu typu obiektowego

W powyższych przykładach pojawiło się nowe słowo kluczowe **constructor** zamiast słowa **procedure** przy metodach *Point.Init*, *Circle.Init*, *Bazowy.Init* i *Potomny.Init*. Konstruktor jest specjalnym rodzajem procedury, która umożliwia właściwe wywoływanie metod wirtualnych.

Choć nie jest to formalny wymóg języka TP6.0, zwyczajowo stosuje się dla konstruktorów nazwę *Init*.

☛ Każdy typ obiektowy zawierający metodę wirtualną musi mieć zdefiniowany konstruktor. Zanim, dla danej zmiennej obiektowej, zostanie wywołana jakakolwiek metoda wirtualna, musi zostać wywołany konstruktor. Jeden typ obiektowy może zawierać kilka konstruktorów.

Wywołanie metody wirtualnej bez wcześniejszego wywołania konstruktora może spowodować zawieszenie się systemu, ponieważ kompilator nie ma możliwości sprawdzenia, które implementacje metod mają być aktualnie wywołane.

☛ Każda pojedyncza zmienna obiektowa musi być inicjowana przez oddzielne wywołanie konstruktora.

☛ Nie jest dopuszczalne zainicjowanie jednej zmiennej i przypisanie tej zmiennej drugiej zmiennej tego samego typu obiektowego. Spowoduje to ustalenie wartości pól drugiej zmiennej, ale jest ona nadal niezainicjowana i przy próbie wywołania jakiejkolwiek jej

metody wirtualnej system się zawiesi (przykład 16.a).

- ☛ Każda zmienna obiektowa będąca elementem struktury danych (np. tablicy) musi być niezależnie inicjowana przez wywołanie konstruktora (przykład 16.b).

Przykład 16.

```
(a) var QCircle,RCircle:Circle;           {definiowane są dwie zmienne}
    begin
        QCircle.Init(600,100,30);         {wyw. konstruktor dla QCircle}
        RCircle:=QCircle;                 {RCircle nie jest prawidłowo zainicjowana!}
    end;

(b) var TabPunkt:array[1..3] of TPoint; i:byte;
    begin
        for i:=1 to 3 do TabPunkt[i].Init(i*100,i*50);
    end;
```

17. Rola konstruktora

Spróbujmy odpowiedzieć na pytanie: czym różni się konstruktor od każdej innej procedury, tzn. co robi poza wykonywaniem poleceń zawartych w jego bloku wykonawczym. Dla każdego typu obiektowego system TP6.0 definiuje tzw. *Tablicę Metod Wirtualnych* – *VMT* (ang. *Virtual Method Table*). Tablica *VMT* zawiera rozmiar typu obiektowego oraz, dla każdej metody wirtualnej, wskaźnik do kodu implementacji tej metody. Zadaniem konstruktora jest ustalenie połączenia między zmienną typu obiektowego wywołującą konstruktor, a *VMT* typu tej zmiennej. Sposób realizacji tego połączenia jest omówiony w następnym podrozdziale.

- ☛ Jest tylko jedna tablica metod wirtualnych *VMT* dla każdego typu obiektowego (jedna *VMT* na typ obiektowy, nie na zmienną). Nigdy dwa typy obiektowe nie mają tej samej *VMT*, niezależnie od tego, jak są bardzo do siebie podobne.

17.1. Reprezentacja obiektów w pamięci operacyjnej komputera

Wewnętrzny format danych typu obiektowego przypomina format danych typu rekordowego. Pola obiektu są pamiętane zgodnie z kolejnością ich definiowania, jako nierozłączny ciąg zmiennych. Pola dziedziczone są pamiętane przed nowymi polami, definiowanymi w typie będącym potomkiem.

Jeśli w typie obiektowym są zdefiniowane metody wirtualne, konstruktor lub destruktory (patrz rozdz. 24), kompilator przydzielając pamięć zmiennej obiektowej dodaje jedno pole, nazywane *polem tablicy metod wirtualnych* (*pVMT*) dla tego typu. Jest to 16-bitowe pole, które jest używane do pamiętania adresu względnego w segmencie (tzw. offsetu) tablicy metod wirtualnych (*VMT*) typu zmiennej. Inaczej mówiąc, jest to wskaźnik do tablicy *VMT*. *Pole VMT* (*pVMT*) występuje bezpośrednio po polach danych obiektu. Zmienna typu obiektowego dziedziczącego od przodków metody wirtualne, konstruktor lub destruktory,

dziedziczy również *pVMT*, które jest umieszczone przed polami własnymi. Dla takiej zmiennej nie jest tworzone dodatkowe pole *pVMT* – patrz rys. 1, który ilustruje wewnętrzną reprezentację zmiennych obiektowych typów *Location*, *Point*, *Circle* zdefiniowanych w przykładzie w rozdz. 15.

Location	Point	Circle
X	X	X
Y	Y	Y
	Visible	Visible
	pVMT	pVMT
		Radius

Rys. 1. Reprezentacja zmiennych obiektowych w pamięci komputera

Warto zauważyć, że metody, mimo iż są deklarowane razem z polami, nie pojawiają się w wewnętrznej reprezentacji obiektów. Wynika to stąd, że każda zmienna typu obiektowego musi posiadać swoją prywatną strukturę danych – miejsce w pamięci operacyjnej komputera. Metody dla wszystkich zmiennych tego samego typu obiektowego są natomiast jednakowe, podobnie jak metody dziedziczone bez przesłaniania są wspólne dla wszystkich zmiennych typów poprzednika i potomka.

Typ *Point* jest pierwszym typem mającym pole *pVMT*, dlatego pole to jest po polu *Visible* (po wszystkich polach z danymi). W typie *Circle*, *pVMT* jest przed polem własnym *Radius*, ponieważ jest ono dziedziczone.

☛ Inicjowanie pola *pVMT* zmiennej obiektowej, czyli nadanie wartości temu polu jest wykonywane jedynie przez wywołanie konstruktora (wyjątek stanowi zmienna z wartością początkową typu obiektowego (*typed constant*), patrz rozdz. 3.2.) Nigdy nie można operować tym polem bezpośrednio w programie.

17.2. Co zawiera tablica metod wirtualnych *VMT* ?

Tablica *VMT* każdego typu obiektowego, który ma zdefiniowane lub dziedziczone metody wirtualne, konstruktor lub destruktor, jest ulokowana w początkowej części segmentu danych. Tablice *VMT* są budowane automatycznie przez kompilator, nigdy nie można nimi manipulować bezpośrednio przez program, podobnie jak wskaźnikiem do tablicy *VMT* (czyli polem *pVMT* zmiennej obiektowej).

Pierwsze słowo tablicy *VMT* zawiera rozmiar zmiennej typu obiektowego, dla którego utworzona jest dana *VMT*. Informacja ta jest wykorzystywana przez konstruktory i destruktory do określania, ile bajtów pamięci należy przydzielić lub zwolnić przy użyciu rozszerzonych syntaktycznie standardowych procedur *New* i *Dispose* (rozdz. 22 i 23).

Drugie słowo *VMT* zawiera zanegowany rozmiar zmiennej danego typu obiektowego. Ta informacja jest używana podczas wywołania metody wirtualnej do sprawdzenia, czy dana zmienna była zainicjowana (tzn. czy był wywołany jej konstruktor). Kontrola ta polega na sprawdzeniu, czy zawartość pierwszego słowa *VMT* jest różna od zera oraz czy suma pierwszego i drugiego słowa *VMT* jest równa zero.

Dalsza część tablicy *VMT* to lista 32-bitowych wskaźników do metod, jednego dla każdej metody wirtualnej danego typu, wg. kolejności ich definiowania w typie. Każde 32-bitowe pole zawiera adres początku kodu odpowiadającej mu metody wirtualnej.

Zawartość tablic wirtualnych dla zdefiniowanych poprzednio typów *Point* i *Circle* jest pokazana na rysunku 2. Założono, że w definicji typu *Point* metoda *MoveTo* jest zdefiniowana jako wirtualna i dodatkowo jest zdefiniowany wirtualny **destructor** *Done* (rozdz. 24). Typ *Location* nie posiada tablicy *VMT* (nie posiada metod wirtualnych, konstruktora ani destruktora).

Point VMT	Circle VMT
\$0008	\$000A
\$FFFF	\$FFFF6
@Point.Done	@Point.Done
@Point.Show	@Circle.Show
@Point.Hide	@Circle.Hide
@Point.MoveTo	@Point.MoveTo

Rys. 2. Tablice metod wirtualnych dla typów *Point* i *Circle*

Konstruktor typu obiektowego zawiera specjalny kod, który powoduje zapamiętanie „offsetu” adresu tablicy *VMT* typu obiektowego w polu *pVMT* zmiennej obiektowej, w momencie jej inicjowania, tzn. w momencie „wejścia” do **begin** konstruktora.

☛ Konstruktor typu obiektowego zawiera specjalny kod, który powoduje zapamiętanie „offsetu” adresu tablicy *VMT* typu obiektowego w polu *pVMT* zmiennej obiektowej, w momencie jej inicjowania, tzn. w momencie „wejścia” do **begin** konstruktora.

Przykład 17.2.

(a) Jeśli mamy zmienną *P* typu *Point*, wywołanie *P.Init* spowoduje automatycznie zapamiętanie w polu *pVMT* zmiennej *P* adresu tablicy *VMT* typu *Point*.

☛ Automatyczne inicjowanie jest częścią początku kodu konstruktora. Kiedy następuje **begin** konstruktora, pole *pVMT* jest już ustalone, parametr *Self* jest ustawiony. Jeśli jest taka potrzeba, konstruktor może już wywoływać metody wirtualne.

Konstruktor może zatem, po nagłówku, zawierać tylko **begin end**; będzie wtedy spełniał swoje zadanie – wypełni pole *pVMT* zmiennej właściwym adresem tablicy *VMT*.

(b) Poniższy program pokazuje, że nie ma konieczności wywoływania konstruktora, jeśli operujemy na zmiennych z wartością początkową (ang. *typed constant*), nawet jeśli w typie są zdefiniowane metody wirtualne. W tym przykładzie nawet nie zdefiniowano konstruktora, a mimo to program działa poprawnie. Nie oznacza to jednak, że należy tak programować – należy do pól obiektu mieć dostęp tylko przez zdefiniowane

metody (co nie jest stosowane w tym programie) oraz stosować konstruktor, jeśli są zdefiniowane metody wirtualne. Gdyby w tym programie zadeklarować p i r jako zmienne, przy pierwszej próbie wywołania p.show wystąpiłby błąd przerywający program: obiekt nie jest zainicjowany.

```

program ObiektyZWartosciaPoczatkowa;
uses crt, graph;
type Tpoint = object
    x,y:integer;
    procedure show;virtual;
end;
TRec = object(Tpoint)
    a,b:integer;
    procedure show;virtual;
end;
const p:Tpoint=(X:200;Y:100);           {zmienna z wartością początkową}
      r:TRec=(X:100;Y:50;a:150;b:200);   {zmienna z wartością początkową}
procedure TPoint.Show;                   {metoda rysowania punktu}
begin
    PutPixel(x,y,yellow);
end;
procedure TRec.Show;                     {metoda rysowania prostokąta}
begin
    Rectangle(x,y,x+a,y+b);
end;
var tryb,karta:Integer;  znak:Char;
begin
    {program główny}
    Detectgraph(karta,tryb);
    Initgraph(karta,tryb,'c:\jezyki\tp\bgi');
    p.Show;                               {wywołanie metody rysowania punktu}
    repeat until keypressed;
    r.show;                               {wywołanie metody rysowania prostokąta}
    while keypressed do znak:=ReadKey;
    repeat until keypressed;
    r.X:=300; r.Y:=300;                   {nie zaleca się tak operować na polach obiektu!}
    r.a:=200; r.b:=50;                   {należy korzystać z metod!}
    r.show;                               {ponowne wywołanie metody rys. prostokąta}
    while keypressed do znak:=ReadKey;
    repeat until keypressed;
    CloseGraph;
end.

```

18. Automatyczna kontrola wywołań metod wirtualnych

Jeśli opcja badania zakresów kompilatora jest włączona (tzn. jest {SR+}), to każde wywołanie metody wirtualnej powoduje sprawdzanie czy zmienna, która wywołuje tę metodę jest zainicjowana. Opcja ta zwalnia pracę komputera, jednakże na etapie

uruchamiania i testowania programu zaleca się ją włączać. Później, dla przyspieszenia pracy programu należy ją wyłączyć.

- Zmienna typu obiektowego ma tylko połączenie z tablicą *VMT* typu, nie ma swojej tablicy *VMT*. Konstruktor ustala wartość pola *pVMT*, które łączy zmienną z tablicą *VMT*, dlatego niemożliwe jest wywołanie metody wirtualnej przed wywołaniem konstruktora.

Przykład 18.

Program pokazujący zastosowanie „późnego wiązania”.

{ Moduł Figures (dostarczony w pakiecie TP6.0) zawiera definicje typów obiektowych (wraz z metodami wirtualnymi) określających punkt i koło, w tej postaci został skompilowany. W prezentowanym programie rozszerzono o łuk (program z pakietu TP6.0) i o prostokąt (własny). Metody tworzone w module nie dla tych figur, doskonale pracują z nimi, bez poprawiania ich i nawet bez ponownej kompilacji modułu – jest to zaleta polimorfizmu. Programy z pakietu TP6.0 wykorzystano za zgodą firmy Borland }

[illegible]

```

begin
    Visible:=True;
    Graph.Arc(X,Y,StartAngle,EndAngle,Radius);
end;
procedure Arc.Hide;                                {metoda 'ukrycia' łuku}
var TempColor:Word;
begin
    TempColor:=Graph.GetColor;
    Graph.SetColor(GetBkColor);
    Visible:=False;
    Graph.Arc(X,Y,StartAngle,EndAngle,Radius);      {Rysowanie łuku
                                                    kolorem tła}
    SetColor(TempColor);
end;
                                                    { poczatek - metody dla dodanego typu Rec}
constructor Rec.Init(InitX1,InitY1,InitDX,InitDY:Integer);
begin
    Point.Init(InitX1,InitY1);
    DX:=InitDX-X;
    DY:=InitDY-Y;
end;
procedure Rec.Show;                                {metoda rysowania prostokąta}
begin
    Visible:=True;
    Graph.Rectangle(X,Y,X+Dx,Y+Dy);
end;
procedure Rec.Hide;                                {metoda 'ukrycia' prostokąta}
var TempColor:Word;
begin
    TempColor:=Graph.GetColor;
    Graph.SetColor(GetBkColor);
    Visible:=False;
    Graph.Rectangle(X,Y,X+DX,Y+DY);                {Rysuj prostokąt kolorem tła,
                                                    aby go 'ukryć'}
    SetColor(TempColor);
end;                                                {koniec dla typu Rec}
begin
    { Program glowny:
    GraphDriver:=Detect;
    DetectGraph(GraphDriver,GraphMode);
    InitGraph(GraphDriver,GraphMode,PathToDrivers);
    if GraphResult<>GrOK then
    begin
        Writeln(GraphErrorMsg(GraphDriver));
        if GraphDriver=grFileNotFound then
        begin
            Writeln('w ',PathToDrivers,'. Zmodyfikuj w programie',
, '„PathToDrivers”');
            Writeln('tak, by stala zawierala aktualna sciezke ',
'dostepu do grafiki.');
```

```

Readln;
Halt(1);
end;
{ Wszystkie obiekty typów potomnych typu Point zawierają metody
wirtualne, a więc muszą być inicjowane poprzez wywołanie konstruktora zanim
będą użyte}
ACircle.Init(151,82,50);
                {Początkowe wartości - X,Y: 151,82, promień:50 pikseli}
AnArc.Init(151,82,25,0,90);
                {X,Y: 151,82, promień: 25, kąt początkowy:0°, końcowy: 90°}
ACircle.Drag(5);                {Parametr określa o ile pikseli przesuwane}
ACircle.Hide;                {Ukrycie koła po zakończeniu przenoszenia}
AnArc.Drag(5);                {Wywołanie metody przenoszenia dla łuku!}
AnArc.Hide;
ARec.Init(100,100,300,300);                {Inicjowanie prostokąta}
ARec.Drag(5);                {Przesuwanie prostokąta}
CloseGraph;
end.

```

{Moduł *Figures* z pakietu TP6.0. Destruktory i obiekty dynamiczne definiowane w tym module, w przedstawionym wyżej programie nie są wykorzystywane. Będą wykorzystane w dalszej części pracy, przy omawianiu przetwarzania obiektów dynamicznych}

```

unit Figures;
interface
uses Graph, Crt;
type Location = object                {definicja typu Położenie}
    X,Y:Integer;
    procedure Init(InitX, InitY: Integer);
    function GetX:Integer;
    function GetY:Integer;
end;

PointPtr = ^Point;
Point = object(Location)                {definicja typu Punkt}
    Visible:Boolean;
    constructor Init(InitX,InitY:Integer);
    destructor Done; virtual;
    procedure Show; virtual;
    procedure Hide; virtual;
    function IsVisible:Boolean;
    procedure MoveTo(NewX,NewY:Integer);
    procedure Drag(DragBy:Integer); virtual;
end;

CirclePtr = ^Circle;
Circle = object(Point)                {definicja typu Koło}
    Radius:Integer;
    constructor Init(InitX,InitY,InitRadius:Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure Expand(ExpandBy:Integer); virtual;
    procedure Contract(ContractBy:Integer); virtual;
end;

```

```

implementation
{
    Metody dla typu Location:
}
procedure Location.Init(InitX, InitY: Integer);
begin
    X:=InitX;
    Y:=InitY;
end;
function Location.GetX: Integer;
begin
    GetX:=X;
end;
function Location.GetY: Integer;
begin
    GetY:=Y;
end;
{
    Metody dla typu Point:
}
constructor Point.Init(InitX, InitY: Integer);
begin
    Location.Init(InitX, InitY);
    Visible:=False;
end;
destructor Point.Done; {zbędny w tym programie, patrz obiekty dynamiczne}
begin
    Hide;
end;
procedure Point.Show; {metoda rysowania punktu}
begin
    Visible:=True;
    PutPixel(X, Y, GetColor);
end;
procedure Point.Hide; {metoda 'ukrycia' punktu}
begin
    Visible:=False;
    PutPixel(X, Y, GetBkColor);
end;
function Point.IsVisible: Boolean;
begin {metoda informująca, czy obiekt jest 'widoczny'}
    IsVisible:=Visible;
end;
procedure Point.MoveTo(NewX, NewY: Integer); {metoda wyświetlenia punktu}
begin {... w nowym miejscu}
    Hide;
    X:=NewX;
    Y:=NewY;
    Show;
end;
function GetDelta(var DeltaX, DeltaY: Integer): Boolean;
var KeyChar: Char; {funkcja mówi o ile pikseli należy przesunąć obiekt}
    Quit: Boolean;
begin
    DeltaX:=0; DeltaY:=0; {0 oznacza, że nie ma zmiany pozycji}

```



```

GetDelta:=True;
repeat
  KeyChar:=ReadKey;                                {Wczytajmy wciśnięty klawisz}
  Quit:=True;                                       {Założmy, że jest wciśnięty właściwy}
  case Ord(KeyChar) of
    0: begin {0 oznacza, że jest to klawisz o podwójnym kodzie}
      KeyChar:=ReadKey;                            {Czytamy drugi bajt kodu klawisza}
      case Ord(KeyChar) of
        72: DeltaY:=-1;                            {Strzałka w górę; zmniejszenie Y}
        80: DeltaY:=1;                              {W dół; zwiększenie Y}
        75: DeltaX:=-1;                            {W lewo; zmniejszenie X}
        77: DeltaX:=1;                              {W prawo; zwiększenie X}
        else Quit:=False;                          {Ignoruj każdy inny kod}
      end; {case}
    end;
    13: GetDelta:=False;                            {CR wciśnięte - brak przesunięcia}
    else Quit:=False;                              {Ignoruj każdy inny kod}
  end; {case}
until Quit;
end;

procedure Point.Drag(DragBy:Integer);                {metoda przesuwania punktu}
var DeltaX,DeltaY:Integer;
    FigureX,FigureY:Integer;
begin
  Show;                                             {Wyświetl figurę, która ma być przesuwana}
  FigureX:=GetX;                                  {Pobierz początkową pozycję figury}
  FigureY:=GetY;
  while GetDelta(DeltaX,DeltaY) do                {To jest pętla przesuwania:}
  begin
    FigureX:=FigureX+(DeltaX*DragBy);              {zmień współrzędną X figury }
    FigureY:=FigureY+(DeltaY*DragBy);              {zmień współrzędną Y figury }
    MoveTo(FigureX,FigureY);                       {wyświetl figurę w nowym punkcie}
  end;
end;

{ Metody dla typu Circle: }

constructor Circle.Init(InitX, InitY: Integer; InitRadius: Integer);
begin
  Point.Init(InitX, InitY);
  Radius:=InitRadius;
end;

procedure Circle.Hide;                             {metoda 'ukrycia' koła}
var TempColor:Word;
begin
  TempColor:=Graph.GetColor;
  Graph.SetColor(GetBkColor);
  Visible:=False;
  Graph.Circle(X,Y,Radius);
  Graph.SetColor(TempColor);
end;

procedure Circle.Expand(ExpandBy:Integer);
{metoda zwiększania rozmiaru koła}
begin

```

```

    Hide;
    Radius:=Radius+ExpandBy;
    if Radius <0 then Radius:=0;
    Show;
end;
procedure Circle.Contract (ContractBy:Integer);
                                                    {metoda zmniejszania koła}
begin
    Expand(-ContractBy);
end;
end.           { koniec modułu }

```

19. Podejmowanie decyzji: procedura czy metoda?

Istnieją co najmniej dwa sposoby pozwalające na korzystanie z narzędzia zdefiniowanego w module, by przesuwac dowolną figurę na ekranie. Jedną z możliwości jest zbudowanie zwykłej procedury tak, żeby była dostępna poza modulem (może być to procedura *DragIt* z poniższego przykładu), drugą – metoda.

✂ Pokazana niżej procedura *DragIt* może być wywoływana zarówno ze zmienną obiektową typu *Point*, jak i każdego typu potomnego. Procedura *DragIt* odwołuje się tylko do identyfikatorów zdefiniowanych w typie *Point*, a te są dziedziczone przez każdego potomka. Metody *GetX*, *GetY* i *MoveTo* są statyczne, procedura *DragIt* zna ich adresy już w momencie kompilacji. *Show* jest metodą wirtualną, zatem jej wywołanie odbywa się przez tablicę metod wirtualnych *VMT* typu parametru aktualnego *AnyFigure*. Decyzja, która implementacja *Show* będzie wywołana podejmowana jest dopiero wtedy, kiedy *DragIt* musi ją wywołać. Taka procedura *DragIt* pracuje całkiem dobrze. Zadajmy sobie pytanie: Jeśli każda figura może być za jej pomocą przesuwana, to dlaczego obiekt, który ma być przesuwany, musi być przekazywany jako parametr, a nie może przesuwać sam siebie? Byłoby tak, gdyby *DragIt* była metodą. Wobec powyższego procedura ta powinna stać się metodą.

Teraz dodatkowy problem: Jeśli ma to być metoda w istniejącej hierarchii obiektów (ze względu na dziedziczenie), to w którym miejscu w tej hierarchii powinna być umieszczona? Przesuwanie figury obejmuje zmianę położenia figury (*MoveTo*) w zależności od danych dostarczonych przez użytkownika (*GetDelta*). Metaforycznie można myśleć o metodzie *Drag* jako o *MoveTo* dla dowolnego obiektu z możliwością skorzystania z danych zewnętrznych. Jeśli chodzi o dziedziczenie – powinna znaleźć się tuż za *MoveTo*. Każdy obiekt, który dziedziczy *MoveTo* powinien również dziedziczyć *Drag*, zatem powinna ona być zdefiniowana dla typu *Point*.

Przykład 19.

```

procedure DragIt (var AnyFigure:Point; DragBy:Integer);
var DeltaX, DeltaY, FigureX, FigureY:Integer;
begin
    AnyFigure.Show;

```

```

FigureX:=AnyFigure.GetX;
FigureY:=AnyFigure.GetY;
while GetDelta(DeltaX,DeltaY) do                {Pętla przesuwania}
begin
    FigureX:=FigureX+(DeltaX*DragBy);
    FigureY:=FigureY+(DeltaY*DragBy);
    MoveTo (FigureX, FigureY);
end;
end;

```

Procedura DragIt przywołuje dodatkową procedurę GetDelta, która określa zmiany X i Y ustalone przez użytkownika np. za pomocą myszki czy klawiatury. Dla uproszczenia w programie przyjęto, że zmiany są wprowadzane klawiszami ze strzałkami z klawiatury.

20. Budować metodę wirtualną czy statyczną ?

„Papierkiem lakmusowym” wskazującym, czy metoda powinna być wirtualna jest ocena, czy spodziewana jest zmiana działania metody dla któregoś obiektu w hierarchii. *DragIt* jest raczej ustalona, operuje tylko pozycją X, Y figury, i nie wydaje się, by potrzebne było coś innego. Prawdopodobnie nie ma potrzeby robić jej wirtualną.

Musimy jednak zdawać sobie sprawę, że możemy nie być w stanie przewidzieć potrzeb przyszłego użytkownika naszego modułu. Pozostawienie tej metody jako statycznej może okazać się utrudnieniem przy dostosowywaniu modułu do własnych potrzeb użytkownika. Zatem należy jednak zdefiniować tę metodę jako wirtualną.

Na ogół powinniśmy budować metody wirtualne. Statyczne metody powinniśmy tworzyć tylko wtedy, gdy chcemy przede wszystkim optymalizować szybkość pracy programu i zajętość pamięci. Ogranicza to możliwości łatwej rozszerzalności programu.

☛ Ogólna reguła postępowania jest następująca: deklarujemy obiekt *Poprzednik* i wewnątrz niego metodę *Action*. Jak zdecydować, czy metoda *Action* ma być statyczna czy wirtualna? Metodę *Action* należy zdefiniować jako wirtualną, jeśli można przewidzieć, że jakiegokolwiek potomek *Poprzednika* może stosować tę metodę w sposób właściwy dla niego.

Zastosujmy tę regułę do obiektów graficznych. W tym przypadku *Point* jest *Poprzednikiem* i należy zdecydować, czy jego metody zrobić wirtualnymi czy statycznymi. Są to metody *Show*, *Hide* i *MoveTo*. Ponieważ każda figura graficzna ma swój własny sposób wyświetlania i kasowania, *Show* i *Hide* są przesłaniane przez każdą potomną figurę. Przesuwanie figury graficznej wydaje się jednakowe dla każdej figury: Wywołaj *Hide* żeby usunąć figurę, przesun współrzędne X,Y i wtedy wywołaj *Show*, żeby ponownie wyświetlić figurę. Ponieważ algorytm *MoveTo* może być wykorzystywany przez każdą figurę, którą można „zakotwiczyć” w pojedynczym punkcie X,Y – dlatego właściwe będzie zdefiniowanie *Point.MoveTo* jako metody statycznej, dziedziczonej przez wszystkich potomków. Metody *Hide* i *Show* muszą być przesłaniane przez własne realizacje poszczególnych obiektów i muszą być wirtualne, aby metoda *Point.MoveTo* wywoływała właściwe dla typów potomnych implementacje metod *Show* i *Hide*.

Przypomnijmy, że jeśli jakiś obiekt ma metody wirtualne, tworzona jest *VMT* tego obiektu w segmencie danych i każdy egzemplarz tego obiektu ma połączenie z tą *VMT*.

Każde wywołanie metody wirtualnej musi przechodzić przez *VMT*, podczas gdy statycznej nie. Oznacza to, że stosowanie metod wirtualnych jest bardziej elastyczne, ale mniej korzystne ze względu na zajętość pamięci i czas wykonywania programu.

CZĘŚĆ II. PROGRAMOWANIE OBIEKTOWE W TP6.0 – PRZETWARZANIE OBIEKTÓW DYNAMICZNYCH

21. Co to są obiekty dynamiczne?

Wszystkie zmienne obiektowe w dotychczasowych przykładach były obiektami statycznymi, kompilator umieszczał je w segmencie danych (ang. *Data Segment*) pamięci komputera.

☛ Słowo „statyczne” ma tu inne znaczenie niż w odniesieniu do metod. W odniesieniu do zmiennych obiektowych, *statyczne* znaczy „nie dynamiczne”, chodzi o moment przydzielania i zwalniania pamięci tym zmiennym. Metody „statyczne” – znaczą „nie wirtualne”.

Zmiennej obiektowej, jak każdej innej zmiennej dynamicznej, można przydzielić pamięć na *stercie* (część pamięci przeznaczoną na zmienne dynamiczne) w dowolnym miejscu programu, jak również można zwolnić zajmowaną przez nią pamięć. Aktualna jest zasada mówiąca że jeśli typ dynamicznej zmiennej obiektowej zawiera metody wirtualne, zmienna ta musi być zainicjowana przez wywołanie konstruktora, zanim wystąpi jakiegokolwiek wywołanie jej metod wirtualnych.

Wywoływanie metod dla dynamicznej zmiennej obiektowej dokonuje się przez nazwę kwalifikowaną, pisząc nazwę wskaźnika ze znakiem ^ (w miejscu nazwy zmiennej przy obiektach statycznych).

Przykład 21.

Można zadeklarować zmienną statyczną:

```
(a) var ACircle:Circle;
```

lub dynamiczną, której możemy przydzielić pamięć (jak rekordowi) procedurą New:

```
var PCircle:^Circle; {typ Circle zdefiniowany w rozdz. 10}
```

begin

```
    New(PCircle); {Procedura ta spowoduje przydzielenie pamięci  
wystarczającej do zapamiętania zmiennej typu bazowego i zwraca adres  
przydzielonej pamięci}
```

```
(b) PCircle^.Init(600,100,30); {Wywołanie konstruktora obiektu dynamicznego}
```

☛ (c) *Dla przypomnienia na czym polega polimorfizm obiektów dobrze będzie prześledzić poniższy program:*

```
program PrzekazywanieObiektow;  
uses Crt;  
type Pierwotny = object
```

```

        X:Byte;
        procedure Ktory; Virtual;
        constructor Init;
    end;
    Pochodny = object(Pierwotny)
        Y:Word;
        procedure Ktory;Virtual;{konstruktor będzie dziedziczony}
    end;

var Pierw:Pierwotny;
    Poch:Pochodny;
    PierwPtr:^Pierwotny;

procedure Pierwotny.Ktory;
begin
    Writeln(' Pierwotny ');
end;

constructor Pierwotny.Init;           {konstruktor nic nie wykonuje, ustala }
begin {połączenie z VMT, wyjątkowo może być nie przesłaniany w Pochodny}
end;

procedure Pochodny.Ktory;
begin
    Writeln(' Pochodny ');
end;

procedure Wartosc(A:Pierwotny);       {parametr przekazywany przez wartość}
begin
    A.Ktory;                           {wywołanie metody Ktory}
end;

Procedure Zmienna(var A:Pierwotny); {parametr przekazywany przez zmienną}
begin
    A.Ktory;
end;

begin { program główny }
    ClrScr;
    Pierw.Init;
    Poch.Init;
    Pierw.Ktory;                        {wypisuje napis: ' Pierwotny '}
    Pierw:=Poch;
    Pierw.Ktory;                        {wypisuje napis: ' Pierwotny '}
    PierwPtr:=@Poch;
    PierwPtr^.Ktory;                   {wypisuje napis: ' Pochodny '}
    Wartosc(Poch);                     {wypisuje napis: ' Pierwotny '}
    Zmienna(Poch);                     {wypisuje napis: ' Pochodny '}
end.

```

22. Alokacja i inicjowanie obiektów dynamicznych za pomocą procedury New

Standardowa procedura *New*, służąca do alokacji zmiennych dynamicznych w pamięci, została rozszerzona tak, by umożliwiała wygodniejszy sposób przydzielania pamięci obiektom dynamicznym z jednoczesnym ich inicjowaniem, tzn. wywoływaniem ich

konstruktora. *New* może być wywoływana z dwoma parametrami: pierwszy – nazwa wskaźnika obiektu, drugi – nazwa konstruktora (nie ma tu potrzeby stosowania nazwy kwalifikowanej) wraz z jego parametrami.

☛ Działanie procedury *New* wywoływanej z dwoma parametrami jest następujące: Najpierw przydzielana jest odpowiednia ilość miejsca na *stercie*. Następnie, jeśli ta czynność nie zakończy się błędem, wywoływany jest konstruktor z podanymi parametrami. Uzyskana w ten sposób dynamiczna zmienna obiektowa ma zainicjowane pole *pVMT*, a więc jest utworzone połączenie zmiennej z właściwą tablicą metod wirtualnych.

New może być stosowana jako funkcja, zwracająca wartość wskaźnika na nowo utworzoną zmienną dynamiczną (niekoniecznie musi to być obiekt). W tym przypadku parametrem wywołania *New* powinien być identyfikator typu wskaźnikowego, a nie nazwa zmiennej, której chcemy przydzielić pamięć (przykład 22b). Również w takim wywołaniu *New* – jako funkcji – można podać jako drugi parametr identyfikator konstruktora wraz z jego parametrami. Podobnie jak przy stosowaniu *New* jako procedury, tak i tutaj wywołanie konstruktora nastąpi dopiero wtedy, gdy przydzielenie żadanego obszaru pamięci na *stercie* odbyło się bez błędu.

Przykład 22.

(a) *Stosowanie New jako procedury (z konstruktorem i bez)*:

```
type ObDynamiczny = Object
                    X:Byte;
                    constructor Init(Wartosc:Byte);
                end;

{Dla tego typu powstanie tablica metod wirtualnych (VMT), mimo iż typ nie
ma metod wirtualnych, ponieważ w typie tym jest zdefiniowany konstruktor!}
var A,B: ^ObDynamiczny;    {nie ma zdefiniowanej nazwy typu wskaźnikowego}
constructor ObDynamiczny.Init(Wartosc:byte);
begin
    X:=Wartosc;
end;

{sposoby stosowania procedury New - z konstruktorem i bez}
begin
    .....
    New(A);                {pierwszy sposób, bez konstruktora, zmienna A}
    A^.Init(11);           {wywołanie konstruktora}
    New(B, Init(22));       {drugi sposób, z konstruktorem, zmienna B}
    Writeln(A^.X);          {wypisze 11, pole obiektu wskazywanego przez A}
    Writeln(B^.X);          {wypisze 22, pole obiektu wskazywanego przez B}
    .....
end.
```

(b) *Stosowanie New jako funkcji (zarówno z jednoczesnym wywołaniem konstruktora, jak i bez). Do tego celu należy zdefiniować nazwę typu wskaźnikowego:*

```
type ObDynamicznyPtr = ^ObDynamiczny; {definicja nazwy typu wskaźnikowego}
var A,B: ObDynamicznyPtr;
```

sposoby stosowania funkcji *New* (z konstruktorem i bez):

```
begin
.....
A:=New(ObDynamicznyPtr);           {zmienna A, parametrem jest nazwa typu}
A^.Init(11);
  B:=New(ObDynamicznyPtr, Init(22)); {zm. B, param: nazwa typu i konstruktor}
Writeln(A^.X);                     {wypisze 11, odwołanie do pola obiektu wskaz. przez A}
Writeln(B^.X);                     {wypisze 22, odwołanie do pola obiektu wskaz. przez B}
.....
end.
```

(c) Przy definicji: `var PCircle: ^Circle; wywołanie procedury New może mieć postać:`
`New(PCircle, Init(600, 100, 30));`

23. Zwalnianie pamięci przydzielonej obiektom dynamicznym

Podobnie jak w przypadku procedury *New*, tak i przy zwalnianiu pamięci można procedurę *Dispose* wywołać na dwa sposoby. Tradycyjny sposób zwalniania pamięci na *stercie* polega na wywołaniu procedury *Dispose* z jednym parametrem, którym jest nazwa (identyfikator) zmiennej wskazującej (np. *Dispose(PCircle)*). W tym wypadku ilość zwalnianej pamięci określa kompilator na podstawie typu identyfikatora będącego parametrem wywołania.

Obiekt może zawierać składniki będące wskaźnikami do dynamicznych struktur lub obiektów, które powinny być we właściwy sposób i we właściwej kolejności usuwane ze *sterty*. Ze względu na rozszerzoną kompatybilność typów obiektowych, w miejsce przodka może być zastosowany potomek, należy zatem dbać o prawidłową ilość zwalnianej pamięci. Wszystkie wymagania dotyczące usunięcia danego obiektu powinny być ujęte razem, w jednej metodzie, której wywołanie spowoduje właściwe zwalnianie pamięci na *stercie*. Proponowana (choć niekonieczna) nazwa takiej metody to *Done* (rozdz. 24).

Jest dopuszczalne i użyteczne, aby typ obiektowy posiadał kilka takich metod, jako że sposób „czyszczenia” pamięci może zależeć od stanu, w jakim jest dany obiekt lub od sposobu przydzielenia mu pamięci.

❏ Procedura *Dispose* może (i powinna) być stosowana z dwoma parametrami: pierwszy z nich musi być zmienną wskazującą na obiekt, a drugi wspomnianą wcześniej metodą „czyszczącą” wraz z jej parametrami. Metoda „czyszcząca” nazywana jest *destruktor*em i ma ona znaczenie, o którym będzie mowa w kolejnych rozdziałach.

Przykład 23.

Sposoby zwalniania pamięci przydzielonej obiektom dynamicznym.

```
type TObiekt = object
    x, y: Integer;
    constructor Init;
    destructor Done;           {definicja destruktor}
end;
ObiektPtr = ^TObiekt;
```



```

constructor TObiekt.Init;      {konstruktor nadaje wartości polom obiektu}
begin
    x:=0; y:=0;
end;

destructor TObiekt.Done;      {destruktor wypisuje napis}
begin
    WriteLn('DO ZOBACZENIA !');
end;

type IntPtr = ^Integer;      {typ wskazujący na zmienne całkowite}

var p1:IntPtr;
    p2,p3:ObiektPtr;

begin
    . . .
    p1:=New(IntPtr);          {Alternatywna postać New}
    New(p2, Init);             {Przydzielenie pamięci i wywołanie konstruktora}
    Dispose(p2, Done);         {Zwolnienie pamięci i wywołanie destruktor}
    {Alternatywna i zalecana postać przydzielania i zwalniania pamięci:}
    p2:=New(ObiektPtr, Init); {Przydzielenie pamięci i wywołanie konstruktora}
    Dispose(p2, Done);         {Zwolnienie pamięci i wywołanie destruktor}
end.

```

24. Co to jest destruktor?

Destruktor jest deklarowany jak inne metody obiektu, przy czym słowo **procedure** zastąpione jest słowem kluczowym **destructor**. Służy on głównie do zwalniania z pamięci obiektów dynamicznych. Może on łączyć zwolnienie pamięci z innym, dowolnym zadaniem określonym w jego bloku wykonawczym.

Destruktry mogą być dziedziczone i mogą być zarówno statyczne, jak i wirtualne.

- ☛ Ponieważ zwykle różne zadania na obiekcie należy wykonywać dla różnych typów po zakończeniu pracy z danym obiektem, zatem dobrym zwyczajem jest programowanie destruktorów zawsze jako wirtualnych; wtedy jest zapewnione wykonanie właściwego destruktor dla typu obiektowego.
- ☛ Zwróćmy uwagę, że nie zawsze jest wymagane definiowanie destruktor (za pomocą słowa kluczowego **destructor**), nawet gdy obiekty zawierają metody wirtualne. Destruktry są w zasadzie wykorzystywane przy pracy na obiektach dynamicznych.
- ☛ Podczas usuwania obiektów dynamicznych z pamięci destruktry gwarantują, że zwolniona zostanie właściwa wielkość pamięci. Jest to istotne, gdy typ obiektu jest różny od typu identyfikatora związanego z tym obiektem. Problem ten dobrze ilustruje poniższy przykład.

Przykład 24.

{ W poniższym przykładzie nie zdefiniowano konstruktora i destruktor w typie potomnym, są one dziedziczone z typu poprzednika. Gdy rozpatrujemy cel przykładu, w którym nie operuje się na polach obiektu Pochodny, interesuje nas tylko wielkość zajmowanej i zwalnianej pamięci – jest to dopuszczalne }

```

program Destruktor;
uses Crt;

type Pierwotny = object
    X:Byte;
    constructor Init(Wartosc:Byte);
    destructor Done;
end;

    Pochodny = object(Pierwotny)
        Y:array[1..100] of Word;
    end;

constructor Pierwotny.Init(Wartosc:Byte);
begin
    X:=Wartosc;                                {ustala wartość pola X}
end;

destructor Pierwotny.Done;
begin
    Writeln('X = ',X);                          {wypisuje wartość pola X}
end;

var Pierw:^Pierwotny;
    Poch:^Pochodny;
    WielkoscStery:LongInt;

begin
    ClrScr;
    WielkoscStery:=MemAvail;

    {                część 1                }
    New(Pierw,Init(11));
    Dispose(Pierw);
    Writeln('Czy MemAvail=WielkoscStery? odp.: ',MemAvail=WielkoscStery);
    { Wypisuje TRUE } {                część 2                }
    New(Poch,Init(22));
    Dispose(Poch);
    Writeln('Czy MemAvail=WielkoscStery? odp.: ',MemAvail=WielkoscStery);
    { Wypisuje TRUE } {                część 3                }
    New(Poch,Init(22));
    Pierw:=Poch;
    Dispose(Pierw,Done);
    Writeln('Czy MemAvail=WielkoscStery? odp.: ',MemAvail=WielkoscStery);
    { Wypisuje TRUE } {                część 4                }
    New(Poch,Init(22));
    Pierw:=Poch;
    Dispose(Pierw);
    Writeln('Czy MemAvail=WielkoscStery? odp.: ',MemAvail=WielkoscStery);
    { Wypisuje F A L S E }
    Writeln('Roznica WielkoscStery - MemAvail wynosi:',
    WielkoscStery-MemAvail:4,' bajtow');
end.

```

W powyższym przykładzie otrzymamy następujący wydruk na ekranie:

Czy MemAvail=WielkoscStery? odp.: TRUE

Czy MemAvail=WielkoscStery? odp.: TRUE

X = 22

Czy MemAvail=WielkoscStery? odp.: TRUE

Czy MemAvail=WieloscStery? odp.: FALSE

Różnica WieloscStery - MemAvail wynosi: 200 bajtów

W części 3. i 4. programu przydzielono pamięć dla zmiennej obiektowej typu Pochodny, po czym wskaźnik tego obiektu przypisano zmiennej wskaźnikowej związanej z typem Pierwotny. Przypisanie takie było możliwe, ponieważ typ Pochodny jest potomkiem typu Pierwotny. Zwolnienie pamięci przez zastosowanie Dispose(Pierw,Done) spowodowało zwolnienie właściwej liczby komórek pamięci, tzn. tyle, ile przydzielono dla zmiennej typu Pochodny. Zwolnienie natomiast pamięci w części 4. programu poprzez użycie Dispose(Pierw) spowodowało zwolnienie tylu komórek pamięci, ile zajmowałaby zmienna typu Pierwotny, mimo iż w tym momencie wskaźnik Pierw wskazywał na zmienną typu Pochodny.

Destruktor decyduje o tym, jak wielki obszar sterty ma być zwolniony, należy zatem definiować go w każdym typie obiektowym. Do zwalniania pamięci należy używać procedury Dispose z nazwą destruktora jako drugim argumentem jej wywołania.

25. Mechanizm zwalniania pamięci za pomocą Dispose z konstruktorem jako drugim parametrem wywołania

Wróćmy do przykładu z obiektami graficznymi. Zmienna typu *Circle* może być przypisana zmiennej typu *Point*. Przypisania takie mogą być wykonywane zarówno dla zmiennych obiektowych, jak i wskaźników wskazujących na te obiekty. *Polimorfizm* obiektów polega na tym, że w czasie kompilacji nie wiadomo jaki obiekt będzie w poszczególnych procedurach używany, wiadomo jedynie, że będzie to któryś z obiektów w *hierarchii* zdefiniowanych obiektów, a każdy z nich zajmuje inną wielkość pamięci.

Destruktor rozwiązuje problem zwalniania właściwego rozmiaru pamięci przez odwołanie się do tego miejsca, gdzie informacja o wielkości pamięci zajmowanej przez dany obiekt jest zapamiętana, czyli do odpowiedniej tablicy *VMT*. W każdej tablicy *VMT* – jak pamiętamy – jest umieszczony rozmiar (w bajtach) danego typu obiektowego. Właściwa *VMT* jest dostępna dla każdej metody w momencie jej wywoływania przez niewidoczny parametr *Self*, przekazywany do metody. Destruktor jest metodą specjalnego rodzaju, która też ma parametr *Self*.

- ☛ Jako ostatnie działanie destruktora sprawdza w tablicy *VMT* rozmiar typu zmiennej, która go wywołuje i przekazuje ten rozmiar do procedury *Dispose*. *Dispose* zwalnia wtedy odpowiedni obszar na *stercie*. Destruktor może być metodą pustą (tzn. zawierać tylko **begin end**;) i też wypełni swoje zadanie; przekaże właściwy rozmiar pamięci do *Dispose*. Zadanie to destruktora wykonuje bez udziału programisty. Dlatego właśnie, aby prawidłowo zwalniać pamięć przy „późnym wiązaniu”, należy wywoływać procedurę *Dispose* z destruktorem jako drugim parametrem.
- ☛ Wywołanie destruktora spoza procedury *Dispose* nie powoduje automatycznego zwalniania pamięci.

Przykład 25. (alokacja obiektów dynamicznych):

{Przykład pokazuje użycie obiektów dynamicznych i destruktora do ich usuwania z pamięci. Tworzona jest dynamiczna lista obiektów graficznych, która jest następnie usuwana z pamięci przez wywołanie destruktora. Program jest dostarczany z pakietem TP6.0 i jest wykorzystany za zgodą firmy Borland.

Budowanie listy wymaga, by każdy utworzony obiekt zawierał wskaźnik do następnego obiektu w tej liście. Typ Point nie zawiera pola będącego takim wskaźnikiem. Ponieważ chcemy w dalszym ciągu korzystać z gotowego modułu, który możemy mieć np. w postaci skompilowanej, nie będziemy dołączać do typu Point jednego dodatkowego pola będącego wskaźnikiem na następny obiekt. Każdy potomny obiekt dziedziczyłby wtedy to pole. Wybierzmy rozwiązanie nie wymagające zmian w naszych typach obiektowych, stanowiących hierarchię obiektów potomnych. Rozwiązaniem tym jest utworzenie prostego, nowego typu obiektowego, nie będącego potomkiem typu Point. Utwórzmy typ List, który umożliwi tworzenie listy obiektów typu Point. Ponieważ Point nie ma wskaźnika na następny obiekt, utworzymy prosty typ rekordowy Node, który będzie miał jedno pole wskazujące na obiekt typu Point i drugie wskazujące na następny obiekt na liście:}

```
type Node = record
    Item:PointPtr;
    Next:NodePtr;
end;
List = object
    Nodes:NodePtr;
    constructor Init;
    destructor Done; virtual;
    procedure Add(Item: PointPtr);
    procedure Report;
end;
```

Typ List będzie miał metody umożliwiające dodawanie nowych obiektów do listy, zatem będzie miał pole Nodes wskazujące na rekord Node. Metoda Add dodaje nie tyle sam obiekt (Point) do listy, lecz wskaźnik tego obiektu. Jest to więc lista wskaźników na obiekty. Ze względu na rozszerzoną kompatybilność typów wskaźnikowych, lista może zawierać wskaźniki zarówno na typ Point jak i na każdy typ potomny typu Point.

W programie ListDemo jest zadeklarowana jedna zmienna statyczna AList typu List i jest zbudowana lista z trzech węzłów. Każdy węzeł wskazuje na różną figurę graficzną, która jest typem Point lub jego typem potomnym. Przed utworzeniem jakiegokolwiek obiektu dynamicznego jest określana liczba bajtów dostępnej pamięci na stercie. Następnie jest sprawdzana liczba dostępnych bajtów na stercie po utworzeniu listy, jak i po usunięciu listy z pamięci za pomocą destruktora. Cała struktura, obejmująca trzy rekordy Node i trzy obiekty Point, jest usuwana przez wywołanie pojedynczego destruktora wywoływanego ze statycznego obiektu typu List – zmiennej Alist. Rysunek 3. pokazuje struktury danych stosowane w poniższym programie.

```
program ListDemo;
uses Crt, Graph, Figures;
```

```

const PathToDrivers = 'C:\TP6\BGI';
type ArcPtr = ^Arc;
    Arc = object(Circle)
        StartAngle, EndAngle: Integer;
        constructor Init (InitX, InitY, InitRadius, InitStartAngle,
            InitEndAngle: Integer);
        procedure Show; virtual;
        procedure Hide; virtual;
    end;

NodePtr = ^Node;
Node = record
    Item: PointPtr;
    Next: NodePtr;
end;

ListPtr = ^List;
List = object
    Nodes: NodePtr;
    constructor Init;
    destructor Done; virtual;
    procedure Add (Item: PointPtr); {dodaje do listy wsk. na Point}
    procedure Report;
end;

var GraphDriver, GraphMode: Integer;
    Temp: String;
    AList: List; {AList jest zmienną statyczną, choć może być dynamiczną}
    PArc: ArcPtr;
    PCircle: CirclePtr;
    RootNode: NodePtr;
    {          Procedury nie będące metodami          }
procedure OutTextLn (TheText: String);
begin
    OutText (TheText);
    MoveTo (0, GetY+12);
end;

procedure HeapStatus (StatusMessage: String);
begin
    Str (MemAvail: 6, Temp);
    OutTextLn (StatusMessage+Temp);
end;

{          Metody dla typu Arc, zdefiniowanego w module Figures          }
constructor Arc.Init (InitX, InitY: Integer; InitRadius: Integer;
    InitStartAngle, InitEndAngle: Integer);
begin
    Circle.Init (InitX, InitY, InitRadius);
    StartAngle := InitStartAngle;
    EndAngle := InitEndAngle;
end;

procedure Arc.Show;
begin
    Visible := True;
    Graph.Arc (X, Y, StartAngle, EndAngle, Radius);
end;

```

```

procedure Arc.Hide;
var TempColor:Word;
begin
    TempColor:=Graph.GetColor;
    Graph.SetColor(GetBkColor);
    Visible:=False;
    Graph.Arc(X,Y,StartAngle,EndAngle,Radius);
    SetColor(TempColor);
end;

    {          Metody dla typu List          }
constructor List.Init;
begin
    Nodes:=nil;
end;

destructor List.Done;
var N:NodePtr;
begin
    while Nodes<>nil do
        begin
            N:=Nodes;
            Nodes:=N^.Next;
            Dispose(N^.Item,Done);
            Dispose(N);
        end;
    end;

procedure List.Add(Item:PointPtr);
var N:NodePtr;
begin
    New(N);
    N^.Item:=Item;
    N^.Next:=Nodes;
    Nodes:=N;
end;

procedure List.Report;
var Current:NodePtr;
begin
    Current:=Nodes;
    while Current<>nil do
        begin
            Str(Current^.Item^.GetX:3,Temp);
            OutTextLn('X = '+Temp);
            Str(Current^.Item^.GetY:3,Temp);
            OutTextLn('Y = '+Temp);
            { Current^.Item^.Show;
              Delay(1000); } {można wyświetlić poszczególny obiekt}
            Current:=Current^.Next;
        end;
    end;

begin    { Program główny          }
    DetectGraph(GraphDriver,GraphMode);
    InitGraph(GraphDriver,GraphMode,PathToDrivers);
    if GraphResult<>GrOK then

```

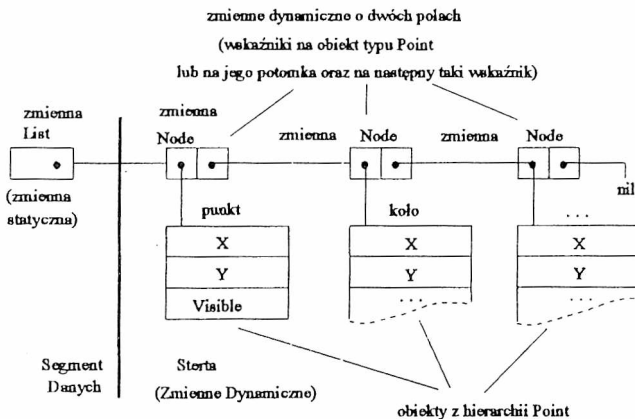
```

begin
  Writeln(GraphErrorMsg(GraphDriver));
  if GraphDriver=grFileNotFound then
  begin
    Writeln('w ',PathToDrivers,
      ' . Zmodyfikuj "PathToDrivers" w programie');
    Writeln('by wskazywała lokalacje zbiorów obsługi grafiki.');
```

```

    Writeln;
  end;
  Writeln('Wcisnij <Enter>...');
  Readln;
  Halt(1);
end;
HeapStatus('Przestrzen na sterpie przed alokacja listy: ');
AList.Init; { Tworzenie listy }
{ Tworzenie i dodanie kilku figur do listy (w pojedynczych operacjach) }
AList.Add(New(ArcPtr, Init(151, 82, 25, 200, 330)));
AList.Add(New(CirclePtr, Init(400, 100, 40)));
AList.Add(New(CirclePtr, Init(305, 136, 5)));
{ Przegląd listy i wyświetlenie wsp. X, Y figur }
AList.Report;
HeapStatus('Heap space after list is allocated ');
AList.Done; { Usunięcie całej listy poprzez jedno wywołanie destruktoru }
HeapStatus('Przestrzen na sterpie po usunięciu listy: ');
OutText('Wcisnij <Enter> by zakonczyc program: ');
Readln;
CloseGraph;
end.
```

Wygląd struktur danych w programie *Listdemo* (wykorzystano tu – za zgodą Borlanda rysunek 8.2 ze strony 161 „Turbo Pascal 7.0, User’s Guide”, z dołączonym opisem i napisami w języku polskim) :



Rys. 3. Schemat struktury danych w programie *ListDemo*

Przyjrzyjmy się destruktorowi *List.Done* z powyższego przykładu. Usunięcie obiektu *List*

obejmuje zwolnienie trzech różnych rodzajów struktur: polimorficznych obiektów figur graficznych znajdujących się na liście, rekordów typu *Node*, które łączą elementy listy, oraz (jeśli jest umieszczony na sterckie) obiektu *List*, który jest głową listy. Cały ten proces obejmuje zwykle wywołanie destruktoru zmiennej *AList*.

Warto zwrócić uwagę na sposób zwalniania pamięci obiektów typu *Point* w destruktorze *Done*:

```
Dispose (N^.Item, Done) ;
```

W przykładzie tym, *N^.Item* jest pierwszym obiektem typu *Point* na liście, a wywoływana metoda *Done* jest jego destruktor. Przypomnijmy, że aktualnym obiektem *N^.Item* niekoniecznie jest obiekt typu *Point*, może to być potomny typ typu *Point*. Usuwany obiekt z pamięci jest obiektem polimorficznym, więc destruktor *Done* musi sprawdzać w tablicy *VMT* właściwego typu rozmiar aktualnego obiektu.

W powyższym przykładzie, *AList* jest zmienną statyczną, ale może być również zmienną dynamiczną:

```
var PList:ListPtr;  
.....  
Dispose (PList, Done) ;
```

Tutaj *Dispose* wywołuje destruktor *Done* aby zwolnił pamięć struktury na sterckie, następnie *Dispose* usuwa z pamięci wskaźnik na *PList*, usuwając głowę listy ze sterckie.

26. Procedury i funkcje w bibliotece TP6.0 związane z przetwarzaniem obiektów

Znana z wcześniejszych wersji języka Turbo Pascal funkcja *SizeOf* ma nieco rozszerzone działanie, ponadto wprowadzona jest dodatkowa funkcja do rozpoznawania typu i procedura do zabezpieczenia się przed przydzieleniem zmiennej obiektowej pamięci, jeśli wystąpi błąd w jej inicjacji.

■ Standardowa funkcja **SizeOf**:

```
function SizeOf (X):Word;
```

gdzie *X* może być nazwą zmiennej lub typu. Funkcja *SizeOf* użyta ze zmienną obiektową mającą pole *pVMT* pobiera rozmiar obiektu zawarty w tablicy *VMT* typu i zwraca go jako wynik. Nie zwraca ona zatem wielkości obiektu jaka powinna wynikać z deklaracji danego identyfikatora, lecz zawsze wielkość wynikającą z aktualnego typu obiektu (przykład 26a). Jeśli parametr *X* jest nazwą typu obiektowego mającego *VMT*, zwracany jest rozmiar typu odczytywany z właściwej *VMT*.

■ Standardowa funkcja **TypeOf** (patrz niżej przykład 26.b):

```
function TypeOf (X):Pointer;
```

gdzie *X* jest identyfikatorem typu obiektowego lub zmiennej typu obiektowego. Funkcja ta najczęściej służy do rozpoznawania aktualnego typu obiektu. Wynikiem (typu *Pointer*) jest

wskazanie na *VMT* danego typu obiektowego. Parametrem wywołania tej funkcji może być tylko zmienna obiektowa mająca pole *pVMT* lub identyfikator typu obiektowego mającego tablicę *VMT*.

■ Standardowa procedura **Fail**:

```
procedure Fail;
```

Procedura ta jest dostępna jedynie wewnątrz konstruktora. Jeśli w procedurze lub funkcji *New* automatycznie jest wykonywany konstruktor, wówczas użycie procedury *Fail* spowoduje zwolnienie uprzednio przydzielonego obiektowi obszaru sterty. Zatem zmiennej wskazującej przypisywana jest wartość *Nil*.

Procedury *Fail* można użyć również wewnątrz konstruktora obiektu nie tworzonego na sterzie. Wówczas konstruktor można wywołać tak, jakby był funkcją zwracającą wartość typu *Boolean*. Jeżeli procedura *Fail* była wywołana, wówczas konstruktor zwróci wartość *False*, w przeciwnym przypadku – *True* (przykład 26.c).

Przykład 26.

(a) Program RozmiarObiektu;

```
uses Crt;
type Maly = Object
    X:Byte;                               {ma jedno pole typu Byte}
    constructor Init;
end;
Wiekszy = Object(Maly)
    Y:Byte;                               {konstruktor jest dziedziczony}
    end;                                   {ma dodatkowe pole typu Byte}
var A:^Maly;    {wskazuje na obiekt zawierający pole Byte i pole VMT}
    B:^Wiekszy; {wskazuje na obiekt z dwoma polami Byte i polem VMT}
constructor Maly.Init;
begin
end;
begin    { program główny }
    ClrScr;
    New(A);
    A^.Init;                               {A wskazuje na obiekt typu Maly}
    Writeln(SizeOf(A^));                   {wypisuje liczbę 3}
    New(B);
    B^.Init;                               {B wskazuje na obiekt typu Wiekszy}
    Writeln(SizeOf(B^));                   {wypisuje liczbę 4}
    A:=B;                                  {przypisanie wskaźników, A wskazuje na Wiekszy}
    Writeln(SizeOf(A^)); {wypisuje liczbę 4, właściwą dla typu zm. B^}
end.
```

(b) Program RozmiarTypu;

```
uses Crt;
type Pierwszy = Object
    X:Byte;
    constructor Init;
end;
Drugi = Object {jest to różny typ, mimo iż ma taki sam rozmiar}
```

```

        Y:Byte;
        constructor Init;
    end;

var A,B:Pierwszy;           {zmienne A i B są tego samego typu}
    C:Drugi;                 {zmienna C jest innego typu}

constructor Pierwszy.Init;
begin
end;

constructor Drugi.Init;
begin
end;

begin
    ClrScr;
    A.Init;                   {inicjowanie zmiennych A, B i C}
    B.Init;
    C.Init;
    Writeln(KindOf(A)=KindOf(B)); {ten sam typ zmiennych, wypisuje: TRUE}
    Writeln(KindOf(B)=KindOf(C)); {różne typy zmiennych, wypisuje: FALSE}
end.

```

(c) {Program pokazujący wykorzystanie procedury Fail, wzorowany jest na przykładzie zamieszczonym w Help wersji TP7.0}

```

program PrzykladProceduryFail;
uses Crt;
const m=50;

type PTablica=^TTablica;           {Typ wskaźnikowy na tablicę}
     TTablica=array[1..m] of byte;
     PBaza = ^TBaza;                {wskaźnik na typ TBaza}
     TBaza = object
         pl:string;                 {pole - wskaźnik na napis}
         constructor Init(FailMe:Boolean;l:string);
         destructor Done;
     end;

     TPochodny = ^TPochodny;         {wskaźnik na typ TPochodny, potomka TBaza}
     TPochodny = object(TBaza)
         p2:PTablica;                {pole - wskaźnik na tablicę}
         constructor Init(FailMe:Boolean);
         destructor Done;
     end;

constructor TBaza.Init(FailMe:Boolean;l:string);
begin
    if FailMe then begin            {jeśli param. FailMe jest True wykonaj}
        writeln('Bedzie wywoływana Fail w TBaza.Init');
        writeln('Pamiec przed Fail w TBaza.Init: ',MaxAvail);
        Fail;
        writeln('Pamiec po wywołaniu Fail w TBaza.Init: ',
            MaxAvail);
    end

    else begin                      {w przeciwnym przypadku wykonaj}
        New(pl);                    {przydziel pamięć dla pola}
        pl^:=l;                     {zapisz napis l w pamięci}
    end
end

```

```

        writeln('Zapisany lancuch w pamieci to: ',p1^);
        writeln('Dostepna pamiec po przydzieleniu lancuchowi: ',MaxAvail)
    end
end;

destructor TBaza.Done;
begin
    Dispose(p1);                {zwolnij pamieć przydzieloną polu p1}
end;

constructor TPochodny.Init(FailMe:Boolean);
begin
    if not TBaza.Init(FailMe,'Napis, ktoremu przydzielil pamiec TBaza.Init')
    then begin
        {jeśli nie jest poprawnie wykonany konstruktor poprzednika - typu TBaza,
        w tym konstruktorze, tzn. dla typu TPochodny, też powinna być wywołana
        procedura Fail. Konstruktor jest wywołany tak, jakby był funkcją zwracającą
        wartość typu Boolean}
        writeln('Bład w TBaza.Init, pamiec przed wywołaniem Fail w ',
            'TPochodny.Init: ',MaxAvail);

        Fail;                    {wywołanie procedury Fail}
        writeln('Bład w TBaza.Init, pamiec po wywołaniu Fail w TPochodny.Init:',
            MaxAvail);
    end else
        {Konstruktor poprzednika wykonany poprawnie, konstruktor typu TPochodny
        kontynuuje pracę - przydziela pamięć dla pola p2:}
        New(p2)
    end;

destructor TPochodny.Done;
begin
    TBaza.Done;                {zwalnia pamięć przydzieloną napisowi - pole p1}
    dispose(p2);                {zwalnia pamięć przydzieloną tablicy - pole p2}
end;

var P:TPochodny;
{ze względu na cel przykładu, deklarujemy tylko zmienną typu potomnego}
X:Boolean;                    {zmienna użyta jako parametr FailMe}

begin
    ClrScr;
    writeln('Pamiec na poczatku: ',MaxAvail);
    for X:=False to True do    {wykonaj dla FailMe równego False i True}
    begin
        writeln(#13#10,'FailMe wynosi ',x);
        P:=New(TPochodny,Init(X));
        if P<>nil then {P=nil, jeśli w P.Init wywołana będzie procedura Fail}
        begin
            writeln('Obiekt zainicjowany, pamiec dla p2 przydzielona ',
                'pozostalo:',MaxAvail);

            Dispose(P,Done);
            writeln('Obiekt zwolniony, pamiec: ',MaxAvail);
        end else
            writeln('Obiekt nie jest poprawnie zainicjowany, ',
                'wskaznik na obiekt (P) wskazuje nil');
            writeln('Pamiec w tej sytuacji, po TPochodny.Init: ',MaxAvail);
        end;
    end;
end;

```

```
    readln;  
end.
```

Po wykonaniu powyższego programu na ekranie monitora pojawi się następujący napis:

Pamięć na początku: 327360

FailMe wynosi FALSE

Zapisany lancuch w pamięci to: Napis, któremu przydzielił pamięć TBaza.Init

Dostępna pamięć po przydzieleniu lancuchowi: 327088

Obiekt zainicjowany, pamięć dla p2 przydzielona, pozostało: 327032

Obiekt zwolniony, pamięć: 327360

Pamięć w tej sytuacji, po PPochodny.Init: 327360

FailMe wynosi TRUE

Bedzie wywoływana Fail w TBaza.Init

Pamięć przed Fail w TBaza.Init: 327344

Błąd w TBaza.Init, pamięć przed wywołaniem Fail w TPochodny.Init: 327344

Obiekt nie jest poprawnie zainicjowany, wskaźnik na obiekt (P) wskazuje nil

Pamięć w tej sytuacji, po PPochodny.Init: 327360

Gdyby w TPochodny.Init nie była wywołana procedura Fail wtedy, gdy TBaza.Init nie przydzielił pomyślnie pamięci, to zamiast 327 360 bajtów wolnej pamięci na sterce, byłoby tylko 327 344 bajtów.

CZĘŚĆ III. ZMIANY W PROGRAMOWANIU OBIEKTOWYM W WERSJI TP7.0

27. Dyrektywa *public*

W TP6.0 można definiować prywatne pola obiektów, tzn. takie pola, które nie mogą być dostępne poza modulem (rozdz. 9). Do tego celu służy dyrektywa **private**. Deklaracja tych pól może wystąpić bezpośrednio po deklaracji pól o swobodnym (publicznym) dostępie. W wersji TP7.0 wprowadzono dodatkowe słowo kluczowe – **public**. Jest to dyrektywa oznaczająca, że deklaracje występujące po niej nie mają specjalnych ograniczeń odnośnie ich zasięgu – są to deklaracje pól i metod publicznych. Można zatem w typie obiektowym deklarować na przemian pola i metody prywatne (o ograniczonym zasięgu) i publiczne – dostępne zarówno w module, jak i w programach i modułach wykorzystujących ten moduł.

Przykład 27.

```
Type TNapis = object
    private                                {definicje pól prywatnych}
        X,Y,Dlugosc:Integer;
        Nazwa:String;
    public                                {definicje pól publicznych}
        constructor Init(x1,y1,d:Integer; Nz:String);
        procedure Edytuj; virtual;
    private                                {definicja pól prywatnych}
        procedure WyswietlNapis(x2,y2:Integer; S:String);
end;
```

⌘ Konstruktor zawsze powinien być definiowany jako metoda publiczna.

28. Słowo kluczowe *inherited*

Dodane w wersji TP7.0 słowo kluczowe **inherited** wskazuje, że występujący po niej identyfikator metody dotyczy metody zdefiniowanej w typie poprzednika. Konstrukcja ta może być użyta tylko wewnątrz definiowania ciała metody typu potomnego (brak istnienia poprzednika byłby błędem). Na ogół takie odwołanie do metody (podobnie jak odwołanie przez nazwę kwalifikowaną) bywa stosowane do wywołania metody zdefiniowanej u poprzednika i przesłoniętej u potomka.

Przykład 28.

```

program ZastosowanieInherited;
uses Crt, Graph;

type TPoint = object
    x,y:integer;
    constructor Init(a,b:Integer);
    procedure show;virtual;
end;

TRec = object(TPoint)           {potomek typu TPoint}
    da,db:integer;
    constructor Init(a,b,al,b1:Integer);
    procedure show;virtual;
end;

var p:Tpoint; r:TRec;
    tryb,karta:Integer; znak:Char;

constructor TPoint.Init;
begin
    x:=a; y:=b;
end;

constructor TRec.Init(a,b,al,b1:Integer);
begin
    inherited Init(a,b);         {wywoływany jest konstruktor z typu TPoint}
    da:=al;db:=b1;               {nadanie wartości dwóm pozostałym polom}
end;

procedure TPoint.Show;
begin
    PutPixel(x,y,yellow);        {lub OutTextXY(x,y,'TU'), tekst lepiej widać}
end;

procedure TRec.Show;
begin
    inherited Show; {metoda Show z TPoint, widoczniejsze, gdy pisze tekst}
    rectangle(x,y,x+da,y+db);    {rysowanie prostokąta}
end;

begin
    Detectgraph(karta,tryb);
    initgraph(karta,tryb,'c:\jezyki\tp\bgi');
    p.Init(250,125);              {utworzenie punktu, nadanie wartości jego polom}
    p.Show;                       {„narysowanie” punktu}
    repeat until keypressed;      {czekanie}
    r.Init(100,50,100,150); {inicjowanie prostokąta - ustalenie wartości pól}
    r.show;                       {narysowanie prostokąta}
    while keypressed do znak:=ReadKey; {czekanie}
    repeat until keypressed;
    r.Init(300,300,200,50); {inicjowanie prostokąta w innym miejscu ekranu}
    r.show;                       {narysowanie prostokąta}
    while keypressed do znak:=ReadKey; {czekanie}
    repeat until keypressed;
    closegraph;
end.

```

29. Metody dynamiczne i Tablica Metod Dynamicznych (DMT)

W wersji TP7.0, oprócz metod wirtualnych można stosować metody dynamiczne. Zapewniają one inny sposób realizacji „późnego wiązania” metod. W związku z tym niewielkiej zmianie uległa struktura *Tablicy Metod Wirtualnych (VMT)*. Dla metod dynamicznych inny jest, niż dla metod wirtualnych, sposób ustalania, która implementacja metody ma być wykonana. Jeśli typ obiektowy zawiera metody dynamiczne, to dla takiego typu budowana jest *Tablica Metod Dynamicznych (DMT)*, która zazwyczaj zmniejsza zapotrzebowanie na pamięć przy programowaniu obiektowym.

29.1. Metody dynamiczne

Deklaracji metod dynamicznych dokonuje się w taki sam sposób jak metod wirtualnych z jedną różnicą: należy dodatkowo określić *indeks* metody dynamicznej (ang. *dynamic method index*) po prawej stronie słowa kluczowego **virtual** (przykład 29.1). *Indeks* metody dynamicznej musi być:

- wartością całkowitą z przedziału 1..65535,
- unikalny dla danej metody – nie może powtarzać się dla żadnej metody dynamicznej tego typu obiektowego lub któregośkolwiek z jego poprzedników.

☛ Jeśli w danym obiekcie jest zdefiniowana metoda dynamiczna o danej nazwie i indeksie, to wszystkie metody o tej nazwie definiowane w typach potomnych muszą być dynamiczne, nagłówki tych metod muszą odpowiadać sobie pod względem liczby i typów parametrów, numeru indeksu oraz typu wyniku, jeśli metodą jest funkcja.

☛ Zanim dla zmiennej zostanie wywołana jakakolwiek metoda dynamiczna, zmienna musi być zainicjowana, tzn. musi być wywołany konstruktor tej zmiennej.

Każda zmienna obiektowa będąca elementem struktury danych musi być niezależnie inicjowana przez wywołanie konstruktora, np. dla zmiennej:

```
var TabPunkt:array[1..3] of TPoint;
```

może być:

```
for i:=1 to 3 do TabPunkt[i].Init(i*100,i*50);
```

☛ Metody dynamiczne wywołuje się w ten sam sposób jak każde inne metody. Swym działaniem nie różnią się one od metod wirtualnych.

☛ Można definiować zmienne, zmienne z wartością początkową lub wskaźniki na obiekty zawierające metody dynamiczne, podobnie jak dla innych typów.

Do przydzielania pamięci zmiennym obiektowym posiadającym metody dynamiczne stosuje się procedurę *New* z drugim parametrem – konstruktorem; do usuwania obiektów z pamięci używa się procedury *Dispose* z drugim parametrem – destruktor.

Przykład 29.1.

Można napisać przykład podobny do 28, ale stosując metody dynamiczne, a nie wirtualne.

W przykładzie dodano jeszcze jedną metodę dynamiczną, powodującą wypisywanie na ekranie graficznym aktualnych współrzędnych punktu dla typu TPoint i współrzędnych narożników dla typu TRec, w miejscu ekranu o współrzędnych będących parametrami tej metody:

```
type TPoint = object
    x,y:integer;
    constructor Init(a,b:Integer);
    procedure Show; virtual 10;           {metoda dynamiczna}
    procedure Pisz_Wsp(c,d:integer); virtual 20; {met. dynam.}
end;
TRec = object(TPoint)
    da,db:integer;
    constructor Init(a,b,al,bl:Integer);
    procedure Show; virtual 10;           {metoda dynamiczna}
    procedure Pisz_Wsp(c,d:Integer); virtual 20; {met. dynamiczna}
end;

...
{występujące w programie identyfikatory sx,sy,sdx,sdy - to dodatkowo
zdefiniowane zmienne globalne typu string[3], poniżej podane są ciała
dodanych metod dynamicznych:}
procedure TPoint.Pisz_Wsp(c,d:integer);
begin
    OutTextXY(c,d,'wspolrzedne punktu: ('+sx+', ('+sy+')');
end;
procedure TRec.Pisz_Wsp(c,d:integer);
begin
    OutTextXY(c,d,'wspolrzedne prostokata: ('+sx+', '+sy+',
        ('+sdx+', '+sdy+')');
end;
begin {program główny}
    ...
    p.Init(250,125);           {inicjowanie punktu}
    p.Show;                   {„rysowanie” punktu}
    r.Init(100,50,100,150);    {inicjowanie prostokąta}
    r.show;                   {rysowanie prostokąta}
    Str(p.x,sx); Str(p.y,sy);
    p.Pisz_Wsp(50,350);
    Str(r.x,sx); Str(r.y,sy); Str(r.x+r.da,sdx); Str(r.y+r.db,sdy);
    r.Pisz_Wsp(50,380);        {wypisanie współrzędnych prostokąta}
    ...
end.
```

29.2. Tablica Metod Wirtualnych i Tablica Metod Dynamicznych

W związku z możliwością definiowania w TP7.0 metod dynamicznych nieco inaczej niż w wersji TP6.0 wygląda *Tablica Metod Wirtualnych (VMT)*. Dwa pierwsze słowa zawierają, podobnie jak poprzednio, *rozmiar* i *zanegowany rozmiar*, służące do zwalniania właściwego rozmiaru pamięci dla obiektów dynamicznych oraz, przy dyrektywie **\$R+**, do

kontroli, czy dana zmienna obiektowa jest zainicjowana. Trzecie słowo *VMT* zawiera adres *Tablicy Metod Dynamicznych (DMT)* danego typu obiektowego lub 0, jeśli typ nie zawiera metod dynamicznych i wobec tego nie jest tworzona dla niego *DMT*. Czwarte słowo *VMT* jest zarezerwowane i zawiera zawsze 0. Następnie jest lista 32-bitowych wskaźników na metody wirtualne zdefiniowane w danym typie, podobnie jak w *VMT* w wersji TP6.0.

Biorąc pod uwagę efektywność wykorzystania pamięci – *VMT* nie są najlepszym rozwiązaniem: jeśli typy potomne dziedziczą daną metodę od swoich poprzedników bez przesłaniania jej, to i tak *VMT* każdego z nich zawiera 4-bajtowy (32-bitowy) wskaźnik na tę samą metodę. Inne jest rozwiązanie dla metod dynamicznych. *Tablica Metod Dynamicznych* zawiera wskaźniki tylko na metody dynamiczne przesłaniane lub zdefiniowane w danym typie. Kiedy potomek przesłania tylko niewielką część dziedziczonych metod dynamicznych, to mniej pamięci będzie zużyte na pamiętanie wskaźników do metod dynamicznych niż przy stosowaniu metod wirtualnych.

■ *Tablica Metod Dynamicznych (DMT)* powstaje tylko dla tych typów obiektowych, które mają (zdefiniowaną bądź dziedziczną) co najmniej jedną metodę dynamiczną, dla jednego typu tworzona jest jedna *DMT*. Jeśli typ obiektowy dziedziczy metody dynamiczne, ale nie przesłania żadnej z nich ani nie ma zdefiniowanej żadnej własnej metody dynamicznej, dziedziczy *DMT* od swego poprzednika. *DMT*, podobnie jak *VMT*, jest pamiętana w początkowej części segmentu danych.

Pierwsze słowo *DMT* wskazuje na *DMT* swojego poprzednika (zawiera offset segmentu danych, w którym jest ona umieszczona). Jeśli nie istnieje *DMT* poprzednika, pierwsze słowo zawiera zero. Drugie i trzecie słowo *DMT* jest używane do pamiętania indeksów ostatnio używanych metod i wskaźników na ich offset (ang. *cached index* i *cached entry offset*). Czwarte słowo (*licznik*) zawiera liczbę metod dynamicznych, których wskaźniki zawiera *DMT*. Kolejne słowa (tyle ile wskazuje licznik) zawierają indeksy metod dynamicznych, zdefiniowanych lub przesłanianych w danym typie. Dalej jest lista 32-bitowych wskaźników na metody dynamiczne.

Przykład 29.2.

```
type Pier = object
    X:Integer;
    constructor Init;
    destructor done; virtual;
    procedure P10; virtual 10;
    procedure P20; virtual 20;
    procedure P30; virtual 30;
    procedure P40; virtual 40;
end;
Poch = object(Poprzednik)
    Y:Integer;
    constructor Init;
    destructor done; virtual;
    procedure P10; virtual 10;
    procedure P30; virtual 30;
```

```

procedure P50; virtual 50;
end;

```

Tablicę Metod Wirtualnych i Tablicę Metod Dynamicznych dla zdefiniowanych wyżej typów przedstawiono na rysunku 4.

Pier VMT	Pier DMT	Poch VMT	Poch DMT
4	0	6	Wsk. Pier DMT
-4	cached index	-6	cached index
Wsk.Pier DMT	cach.entry off.	Wsk.Poch DMT	cach.entry off.
0	4	0	3
@ Pier.Done	10	@ Poch.Done	10
	20		30
	30		50
	40		@Poch.P10
	@Pier.P10		@Poch.P30
	@Pier.P20		@Poch.P50
	@Pier.P30		
	@Pier.P40		

Rys. 4. Tablice Metod Wirtualnych (VMT) i Metod Dynamicznych (TMD) dla typów *Pier* i *Poch*

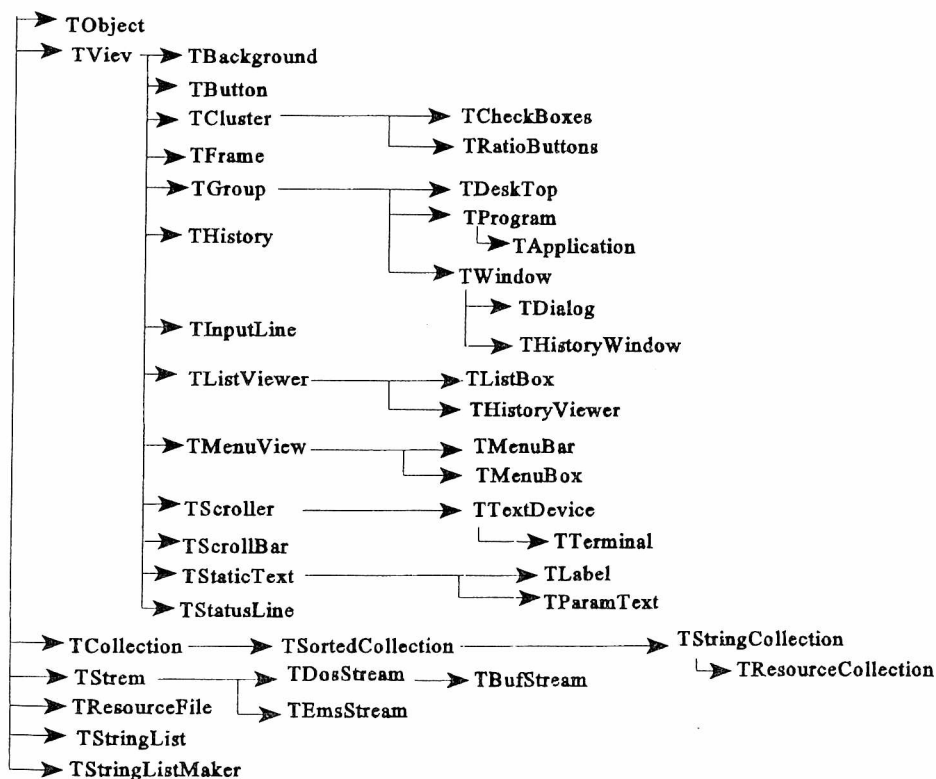
Stosowanie metod dynamicznych zamiast wirtualnych daje zazwyczaj oszczędności w zapotrzebowaniu na pamięć, ale odbywa się to kosztem czasu zużywanego na wywoływanie tych metod. Przy stosowaniu metod dynamicznych musi nastąpić przeglądanie *DMT* typu zmiennej, z której metoda została wywołana i *DMT* typu poprzednika, w celu wyszukania odpowiedniego indeksu metody. Dopiero później może nastąpić właściwe wywołanie metody przez odpowiednie wskazanie jej adresu. Biblioteka TP7.0 zawiera specjalną procedurę (*routine*) *Dispatch*, która jest używana przez system przy wywoływaniu metod dynamicznych. Po wywołaniu metody dynamicznej, w *VMT* odczytywany jest adres (offset) właściwej *DMT*. Następnie – przy użyciu pola *cached index* w *DMT*, sprawdzane jest, czy szukany indeks jest tym, który był ostatnio wywoływany. Jeśli tak, natychmiast odczytywany jest adres (offset) z pola *cached entry offset* i wywoływana jest właściwa metoda. Jeśli nie, szukany jest indeks w *DMT* lub w *DMT* poprzednika, pobierany jest adres odpowiedniej metody, indeks i adres są zapamiętywane w polach *cached index* i *cached entry offset* i dopiero wtedy jest wywoływana metoda. Procedury biblioteki TP7.0 wywołujące metody dynamiczne są dość dobrze zoptymalizowane, tak aby zużycie czasu było jak najmniejsze, niemniej wolniejsza praca jest ceną za mniejsze zapotrzebowanie na pamięć.

Dodatek A

BAZA DANYCH NA OBIEKTACH

1. Wprowadzenie

Mimo że istnieje wiele systemów do projektowania baz danych, czasami wygodne jest zaprogramowanie specyficznej bazy danych do własnych potrzeb, np. w języku Turbo Pascal.



Rys. 5. Hierarchia obiektów biblioteki Turbo Vision

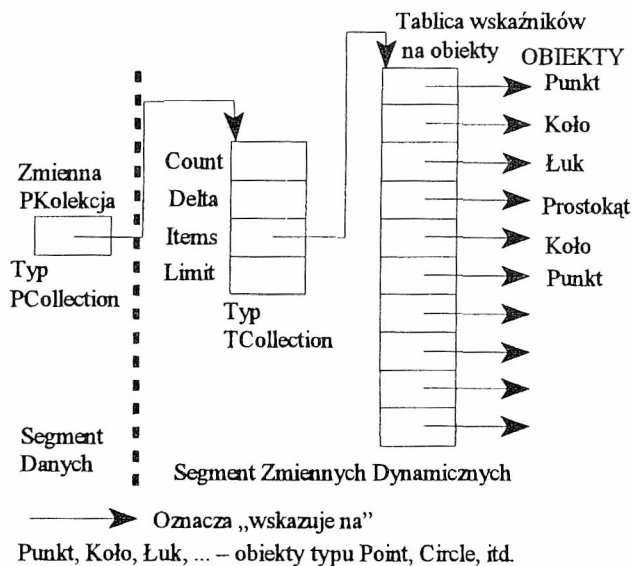
Definiując rekordy i pliki rekordów (*file of typ_rekordowy*), można dane takiej bazy zapamiętać na dysku, odczytać i modyfikować je według potrzeb. Nie można zdefiniować

pliku obiektów (błąd kompilacji: składnikiem pliku nie może być inny plik lub obiekt) i wobec tego niemożliwe jest utworzenie takiej bazy na obiektach. Dołączona do TP7.0 biblioteka *Turbo Vision* dostarcza wielu predefiniowanych typów obiektowych i metod ich przetwarzania, umożliwiającą operowanie okienkami (takimi, jakie są w środowisku Turbo Pascala), obsługę zdarzeń, myszki, tworzenie zasobów i kolekcji obiektów, tworzenie strumieni i zapisywanie do pliku dyskowego kolekcji obiektów różnych typów (rysunek 5). Biblioteka *Turbo Vision* nie jest przedmiotem niniejszej pracy, zainteresowani powinni sięgnąć do specjalistycznej literatury, np. do materiałów dokumentacyjnych dostarczanych wraz z systemem TP7.0 lub innej literatury, dostępnej w języku polskim. Pokazanie na rysunku 5. hierarchii obiektów zdefiniowanych w *Turbo Vision* ma jedynie uświadomić czytelnikowi jakie są możliwości programowania. W tym miejscu ograniczymy się do krótkiego omówienia tych elementów, które są wykorzystywane przy tworzeniu bazy danych oraz pokazania przykładowej bazy danych na obiektach.

Biblioteka *Turbo Vision*, umożliwia między innymi tworzenie kolekcji, zapisywanie jej na dysk, odczytywanie i modyfikację. Istnieją więc możliwości zaprogramowania bazy danych na obiektach.

2. Kolekcja

Kolekcja jest dynamiczną strukturą zbioru obiektów, przydziela ona pamięć na *stercie* dla dołączanych do niej obiektów i zapamiętuje wskaźniki na nie. Jest ona dynamiczną tablicą wskaźników na elementy (rysunek 6).



Rys. 6. Przykładowa struktura danych dla kolekcji

W skład kolekcji mogą wchodzić obiekty różnych typów, element takiej kolekcji może mieć dowolny typ i rozmiar, ponieważ kolekcja używa wskaźników na elementy. Mogą pojawić się problemy z dołączaniem do kolekcji (i przetwarzaniem) elementów nie będących obiektami, ponieważ niektóre metody używane w kolekcji wykorzystują pola i metody typu *TObject*, który jest pierwszym typem w hierarchii *Turbo Vision*. Elementy nie będące obiektem, nie mogą być potomkami typu *TObject* i wobec tego nie mogą być przetwarzane przez takie metody. Wygodniej jest definiować obiekty, jeśli zamierzeniem jest tworzenie z nich kolekcji. Zatem kolekcja obiektów graficznych może składać się z punktów, kół, łuków, prostokątów itp. Kolekcja może mieć dowolną liczbę elementów. Mimo iż na początku ustala się rozmiar kolekcji (liczbę jej elementów), to później może być ona dynamicznie powiększana, o ile zachodzi taka potrzeba i pozwala na to dostępna pamięć.

Przy tworzeniu kolekcji obiektów korzysta się z predefiniowanych typów *TObject* oraz *TCollection*. Chcąc zapisać utworzoną kolekcję na dysk, należy korzystać z typów *TStream* i potomnego typu *TBufStream* (lub *TDosStream*) – do definiowania i rejestracji strumienia. W programach do pamiętania kolekcji stosuje się zmienną wskaźnikową na typ *TCollection*. Jest to predefiniowany typ o nazwie *PCollection*. Przykładowa struktura danych dla kolekcji jest pokazana na rysunku 6. Załóżmy, że mamy utworzoną kolekcję (poprzez przydzielenie jej pamięci i wywołanie konstruktora) o 10 elementach, która jest pamiętana w zmiennej o nazwie *PKolekcja* typu *PCollection*. Do kolekcji tej dołączonych zostało już (przez wywołanie metody *Insert*) sześć obiektów: punkt, koło, łuk, prostokąt, koło i punkt. Ponieważ w *tablicy wskaźników na obiekty* są pamiętane wskaźniki na dołączane do kolekcji obiekty, zatem typ tych obiektów może być różny.

3. Typ *TCollection*

W typie *TCollection* są zdefiniowane cztery pola oraz wiele metod służących do tworzenia, zapisywania na dysk i odczytywania, oraz przetwarzania kolekcji. Dokładny ich opis jest w dokumentacji TP7.0, *Turbo Vision*. Ograniczmy się tylko do krótkiego ich omówienia i pokazania sposobu tworzenia i przetwarzania kolekcji na przykładzie. W prezentowanym przykładzie nie wykorzystano wszystkich metod stosowanych typów, nie zachodziła taka potrzeba. Starano się tak dobrać przykład, by pokazać w nim najważniejsze czynności dla wykonania prostej bazy danych.

Typ *TCollection* ma cztery własne pola:

Count – (typ całkowity) liczba elementów w kolekcji (liczone są od zera),

Delta – typ całkowity, mówi, o ile elementów należy powiększyć kolekcję, jeśli jest pełna,

Items – typu *PItemList* – jest to pole tylko do odczytu, wskazuje na tablicę wskaźników do obiektów w kolekcji,

Limit – typu całkowitego – tylko do odczytu, zawiera informację ile jest aktualnie obiektów w kolekcji.

Metody typu *TCollection*:

constructor Init(ALimit,ADelta:Integer);

Ustala wartości pól *Limit* i *Delta*. Przydziela pamięć dla tylu elementów kolekcji (wskaźników na obiekty dołączane do kolekcji), ile wynosi wartość parametru *ALimit*. Rozmiar kolekcji może być później dynamicznie zmieniany, może być ona powiększana lub zmniejszana. Jeśli do kolekcji dodaje się element mimo wyczerpania *ALimit*, przydzielana pamięć kolekcji zwiększa się o *ADelta* elementów; czynność ta może być powtarzana aż do wyczerpania się pamięci lub do czasu, gdy liczba elementów osiągnie wartość *MaxCollectionSize*. *MaxCollectionSize* jest zmienną wynoszącą 65520 **div** SizeOf(Pointer).

constructor Load(**var** S:TStream);

Jest to konstruktor, który tworzy i ładuje ze strumienia kolekcję. Zmienna S jest typu *TStream*, wykorzystywanego przy zapisie kolekcji na dysk i krótko omówionego w dalszej części.

destructor Done;**virtual**;

Usuwa z kolekcji wszystkie elementy i zwalnia zajmowaną przez nie pamięć, przez wywołanie *FreeAll* (patrz dalej).

function At(Index:Integer):Pointer;

Pozwala traktować kolekcję jako tablicę o indeksach będących liczbami naturalnymi począwszy od zera. Zwraca adres elementu kolekcji o indeksie *Index*. *Index* musi się mieścić w przedziale [0, *Count*], inaczej będzie generowany błąd.

procedure AtDelete(Index: Integer);

Usuwa element znajdujący się na pozycji *Index* z kolekcji, przesuwając dalsze elementy o jedną pozycję w górę, zmniejsza pole *Count* o jeden, ale nie zwalnia pamięci dla usuwanego elementu.

procedure AtFree(index:Integer);

Usuwa z kolekcji element znajdujący się na pozycji *Index* i zwalnia zajmowaną przez niego pamięć. Działanie tej metody jest równoważne:

```
Item:=At (Index) ;
AtDelete (Index) ;
FreeItem (Item) ;
```

procedure AtInsert(Index:Integer;Item:Pointer);

Dodaje do kolekcji element wskazywany przez *Item* na pozycję określoną przez wartość parametru *Index* i przesuwając dalsze elementy o jeden w dół. Jeśli kolekcja ma już ustaloną liczbę (*Limit*) elementów, metoda ta zwiększa przydzieloną pamięć o *Delta* elementów przez wywołanie *SetLimit*.

procedure AtPut(Index:Integer;Item:Pointer);

Zamienia dotychczasowy element z pozycji *Index* na element wskazywany przez *Item*.

procedure Delete(Item:Pointer);

Usuwa z kolekcji element wskazywany przez *Item*, nie zwalnia pamięci zajmowanej przez usuwany element. Jest równoważne wykonaniu metody: *AtDelete(IndexOf(Item))*.

procedure DeleteAll;

Usuwa z kolekcji wszystkie elementy, nie zwalnia zajmowanej przez nie pamięci, ustala wartość pola *Count* na zero.

procedure Error(code,Info:Integer);**virtual**;

Procedura ta jest wywoływana przez inne metody w sytuacji, kiedy wykrywany jest błąd. Użytkownik może przesłonić tę metodę tak, by sygnalizowała błąd bez przerywania programu.

function FirstThat(Test:Pointer):Pointer;

Funkcja ta stosuje funkcję logiczną *Test*, dopasowującą podany warunek. Funkcji *FirstThat* przekazywany jest wskaźnik na funkcję *Test* zdefiniowaną przez użytkownika (patrz przykład kolekcji zamieszczony na końcu tego dodatku). Testowany jest każdy element kolekcji, począwszy od elementu o indeksie 0, aż zostanie znaleziony pierwszy element spełniający warunek określony w funkcji *Test* lub zostanie przejrzana cała kolekcja. Funkcja *FirstThat* zwraca adres znalezionego elementu lub **nil**, gdy taki element nie zostanie znaleziony w kolekcji. Funkcja testująca *Test* nie może być globalna i musi być zdefiniowana z dyrektywą **far**.

procedure ForEach(Action:Pointer);

Procedura powoduje wykonanie zadanej akcji dla każdego elementu w kolekcji. Akcja jest definiowana przez użytkownika w postaci procedury *Action*, której adres jest przekazywany do *ForEach* jako parametr. Procedura *Action* musi być procedurą lokalną, może to być np. procedura drukująca element kolekcji, jak w zamieszczonym dalej przykładzie.

procedure Free(Item:Pointer);

Usuwa z kolekcji element wskazywany przez *Item* i zwalnia zajmowaną przez niego pamięć. Jest równoważna wykonaniu dwóch metod:

```
Delete (Item) ;
FreeItem (Item) ;
```

Procedure FreeAll;

Usuwa z kolekcji wszystkie elementy i zwalnia zajmowaną przez nie pamięć, wywołuje *DeleteAll*.

procedure FreeItem(Item:Pointer);**virtual**;

Metoda powinna zwalniać pamięć zajmowaną przez element kolekcji wskazywany przez *Item*. Może być przesłonięta. Domyślnie metoda ta zakłada, że *Item* wskazuje na potomka typu *TObject* i wywołuje destruktor *Done* wskazywanego przez *Item* obiektu. Dla obiektów kolekcji nie będących potomkami typu *TObject*, metoda *FreeItem* powinna być przesłonięta tak, by zwalniała pamięć zajmowaną przez obiekt wskazywany przez *Item*.

function GetItem(var S:TStream):Pointer;**virtual**;

Funkcja czyta element ze strumienia *S*. Domyślnie zakłada, że elementy w kolekcji są potomkami typu *TObject* i wywołuje metodę *TStream.Get* aby wczytać element. Jeśli elementy nie są potomkami *TObject*, należy przesłonić metodę *GetItem* tak, by czytała ze strumienia właściwy element i zwracała wskaźnik na niego. *GetItem* nie powinna być wprost wywoływana w programie, jest wywoływana przez metodę *Load*.

function IndexOf(Item:Pointer):Integer;**virtual**;

Funkcja zwraca numer pozycji (indeks) w kolekcji elementu wskazywanego przez *Item*, jest odwrotną funkcją do *At*. Jeśli wskazywany przez *Item* element nie jest w kolekcji, funkcja zwraca wartość -1.

procedure Insert(Item:Pointer);**virtual**;

Procedura włącza do kolekcji element wskazywany przez *Item* i uaktualnia pozostałe indeksy, jeśli jest to potrzebne. Domyślnie, *Insert* dołącza element na koniec kolekcji przez wywołanie *AtInsert(Count,Item)*. Kolekcje potomne, np. kolekcje sortowane (*TSortetCollection*) mogą dołączać element w inne miejsce w kolekcji.

function LastThat(TestPointer):Pointer;

Funkcja bardzo podobna do *FirstThat* z tą różnicą, że przeszukiwanie kolekcji następuje w odwrotnej kolejności, „od końca”. *Test* jest lokalną funkcją logiczną, której adres jest przekazywany do *LastThat*.

procedure Pack;

Usuwa z kolekcji wszystkie wskaźniki o wartości **nil**.

procedure PutItem(var S:TStream;Item:Pointer);**virtual**;

Zapisuje do strumienia *S* element kolekcji wskazywany przez *Item*. Domyślnie zakłada, że elementy kolekcji są potomkami typu *TObject* i wywołuje metodę *TStream.Put*, by je zapamiętać. Dla innych elementów *PutItem* musi być przesłonięta, by właściwie zapisywała element do strumienia. Metoda ta nie powinna być wywoływana wprost w programie, jest wywoływana przez metodę *Store*.

procedure SetLimit(ALimit:Integer);**virtual**;

Procedura ustala, ile elementów ma być w kolekcji: zwiększa lub zmniejsza pamięć dla kolekcji tak, by mogła ona obejmować *ALimit* elementów. Jeśli *ALimit* jest większe od *Limit* – przydziela pamięć dla nowej tablicy wskaźników, przepisuje stare elementy do nowo utworzonej tablicy i dodaje nowe, zwalnia pamięć zajmowaną przez starą tablicę. Dla *ALimit* mniejszych od *Count* ustala rozmiar kolekcji na *Count*, dla większych od *MaxCollectionSize* ustala na *MaxCollectionSize*.

procedure Store(var S:TStream);

Zapamiętuje kolekcję i wszystkie jej elementy w strumieniu *S*. Metoda *Store* wywołuje dla każdego elementu w kolekcji metodę *PutItem*.

4. Strumienie

Strumienie łączą w sobie zalety zarówno plików zdefiniowanych (elementowych), jak i niezdefiniowanych (amorficznych): zachowany jest typ wysyłanych do strumienia elementów, ale to, jakiego typu elementy będą wysyłane do strumienia nie decyduje się na etapie kompilacji programu. Jeśli obiekty są potomkami typu *TObject*, mogą być łatwo zapisywane do strumienia, niezależnie od tego czy wszystkie są tego samego typu, czy nie. Programista musi określić strumieniowi, które obiekty powinien on przysyłać i strumień sam wykonuje właściwą kontrolę nad zachowaniem typów przekazywanych do niego obiektów. Mechanizm działania strumienia, polegającego na tym, iż może on zapisywać (i odczytywać) obiekty różnych typów, jest całkiem inny niż tradycyjnego I/O Pascala. Polega on na *rejestracji* każdego typu, który ma być przekazywany do strumienia – czyli na przypisaniu mu unikalnego numeru. Numer ten jest zapisywany do strumienia jako

nagłówkę zapisywanej do niego danej typu obiektowego. Podczas czytania obiektu ze strumienia, najpierw jest odczytywany numer rejestracyjny typu odczytywanej danej, numer ten jest kojarzony z odpowiednim typem i jego tablicą *VMT*; dlatego też ze strumienia czytana jest odpowiednia liczba bajtów danych i są one właściwie interpretowane (jako pola czytanego obiektu). Strumień, podobnie jak plik, jest sekwencyjny, ale można też zapisywać do niego (lub odczytywać) różne obiekty na określonej pozycji. Programista musi zadbać o odpowiednie ustalenie parametrów strumienia i właściwy sposób zapisywania do niego elementów i ich odczytywania. Strumień, tak jak i plik, jest logicznym modelem fizycznych zbiorów na dysku. Po skończonej pracy z plikiem należało go zamknąć; po skończonej pracy ze strumieniem należy zwolnić zajmowaną przez niego pamięć (z wywołaniem destruktora). Zanim przejdziemy do przykładu, omówimy pokrótce pola i metody używanych typów *TStream* oraz *TBufStream*.

5. Typ *TStream*

W typie *TStream* są zdefiniowane dwa własne pola:

ErrorInfo – typu całkowitego, zawiera dodatkowe informacje w sytuacji, kiedy wykryty został błąd.

Status – typu całkowitego, wskazuje aktualny status strumienia. Status ten może mieć wartość:

stCreate – tworzenie nowego pliku,

stOpenRead – otwarcie istniejącego pliku z dostępem tylko do czytania,

stOpenWrite – otwarcie istniejącego pliku z dostępem tylko do zapisu,

stOpen – otwarcie istniejącego pliku z dostępem do zapisu i do odczytu.

Metody typu *TStream*:

procedure CopyFrom(**var** S:TStream;Count:LongInt);

Kopiuje do aktualnego strumienia *Count* bajtów ze strumienia *S*.

procedure Error(Code,Info:Integer);**virtual**;

Metoda jest wywoływana wtedy, gdy w strumieniu wystąpił błąd. Domyślnie, *Error* zapamiętuje pola *ErrorInfo* i *Status*, następnie, jeśli zmienna globalna *StreamError* nie wskazuje **nil**, wywołuje procedurę wskazywaną przez *StreamError*. Po wystąpieniu błędu w strumieniu nie można wykonywać żadnych operacji na nim, dopóki nie zostanie wykonana metoda *Reset* (patrz dalsza część).

procedure Flush;**virtual**;

Abstrakcyjna metoda, która musi być przesłonięta, aby mogła być stosowana. Jeśli strumienie stosują bufory, metoda ta służy do ich czyszczenia.

function Get:PObject;

Funkcja czyta obiekt ze strumienia i zwraca wskaźnik na niego. Obiekt ten powinien być wcześniej zapisany do strumienia metodą *Put*. Metoda *Get* czyta ze strumienia najpierw identyfikator typu, porównuje go z polem *ObjType* wszystkich zarejestrowanych typów obiektowych (*ObjType* jest polem w rekordzie rejestracyjnym typu *TStreamRec*, rozdz. 7).

Następnie wywołuje metodę *Load*, będącą konstruktorem typu czytanego obiektu, by utworzyć i zainicjować obiekt.

function GetPos:LongInt;**virtual**;

Abstrakcyjna metoda (musi być przesłonięta) zwracająca aktualną pozycję w strumieniu.

function GetSize:LongInt;**virtual**;

Abstrakcyjna metoda (musi być przesłonięta) zwracająca całkowity rozmiar strumienia.

procedure Put(P:PObject);

Zapisuje obiekt *P* do strumienia. Najpierw *Put* odszukuje rekord rejestracyjny typu przez porównanie offsetu *VMT* z polem *VmtLink* wszystkich zarejestrowanych typów obiektowych (rozdz. 7). Następnie zapisuje numer rejestracyjny typu (pole *ObjType* z rekordu rejestracyjnego) do strumienia i wywołuje metodę *Store* typu zapisywanego obiektu, by zapisać sam obiekt.

Jeśli *P* wskazuje *nil*, *Put* zapisuje do strumienia słowo zawierające zero. Jeśli typ obiektu *P* nie był zarejestrowany, *Put* wywołuje metodę *Error* i nic nie zapisuje do strumienia.

procedure Read(var Buf;Count:Word);**virtual**;

Abstrakcyjna metoda (musi być przesłonięta dla wszystkich typów potomnych) powinna czytać *Count* bajtów ze strumienia do bufora i przesuwac aktualną pozycję strumienia o *Count* bajtów. W typie *TBufStream* metoda ta jest przesłonięta i programista może z niej korzystać.

function ReadStr:PString;

Czyta łańcuch znaków ze strumienia (począwszy od aktualnej pozycji strumienia), zwraca wskaźnik na ten łańcuch, czyli wartość typu *PString*. Wywołuje procedurę *GetMem(Długość+1)* do przydzielenia łańcuchowi pamięci.

procedure Reset;

Usuwa błędy ze strumienia przez ustawienie pola *Status* i *ErrorInfo* na 0. Umożliwia dalszą pracę ze strumieniem, w którym wcześniej wystąpił błąd.

procedure Seek(Pos:LongInt);**virtual**;

Abstrakcyjna metoda (musi być przesłonięta dla wszystkich typów potomnych) powinna ustawiać aktualną pozycję strumienia na *Pos*.

procedure Truncate;**virtual**;

Abstrakcyjna metoda (musi być przesłonięta dla wszystkich typów potomnych) powinna usuwać dane z aktualnego strumienia od aktualnej pozycji do końca.

procedure Write(var Buf;Count:Word);**virtual**;

Abstrakcyjna metoda (musi być przesłonięta dla wszystkich typów potomnych) powinna zapisywać *Count* bajtów z *Buf* do strumienia i ustawiać aktualną pozycję strumienia.

procedure WriteStr(P:PString);

Zapisuje łańcuch *P* do strumienia, począwszy od aktualnej pozycji w strumieniu. *P* jest wskaźnikiem na łańcuch.

6. Typ *TBufStream*

Jest to typ stosowany w podanym niżej przykładzie, dlatego pokrótce go omówimy. Typ *TStream* ma dwa bezpośrednie typy potomne:

- *TDosStream* – tworzy plik DOSowski o ustalonej nazwie i trybie dostępu. Ma dodatkowe pole *Handle*, które jest tradycyjnym polem w plikach DOSowskich (pole w rekordzie *TFileRec*), używanym (wewnętrznie przez Turbo Pascal) do dostępu do otwartego pliku. W typie tym przesłonięte są wszystkie abstrakcyjne metody z typu *TStream*, z wyjątkiem *TStream.Flush*. W większości aplikacji, zamiast niebuforowany strumień typu *TDosStream*, stosuje się typ *TBufStream*, który jest bezpośrednim potomkiem typu *TDosStream*.

- *TEmsStream* – tworzy specjalizowany strumień w pamięci EMS. Przesłania 6 abstrakcyjnych metod z typu *TStream* (nie przesłania tylko *TStream.Flush*). Przy używaniu strumienia w EMS, jedynie wywołanie destruktor *Done* lub *restart* komputera powoduje zwolnienie pamięci EMS.

Bezpośrednim potomkiem typu *TDosStream* jest typ *TBufStream*. Typ ten implementuje buforowaną wersję strumienia DOSowskiego. Zawiera dodatkowe pola, przesłania 8 metod z typu *TDosStream* i tworzy abstrakcyjną metodę *Flush*. Jest to najczęściej używany typ do tworzenia strumieni w aplikacjach, dlatego zostanie pokrótce omówiony.

Pola własne typu *TBufStream* to:

Buffer – typu *Pointer*, wskazuje na początek bufora strumienia.

BufEnd – typu *Word*, zawiera offset z *Buffer*, wskazujący na ostatni bajt użyty w buforze.

BufPtr – typu *Word*, zawiera offset z *Buffer*, wskazujący na aktualną pozycję w buforze.

BufSize – typu *Word*, zawiera rozmiar bufora w bajtach.

Metody typu *TBufStream* to:

constructor *Init*(*FileName*:*FNameStr*; *Mode*, *Size*:*Word*);

Tworzy obiekt i otwiera plik o nazwie *FileName* z dostępem *Mode*, wywołując *Init* dziedziczony z *TDosStream*. Ustala wartości *BufPtr* i *BufEnd* na zero. Przydziela *Size* bajtów dla bufora na *stercie*. Typowy rozmiar bufora to 512 do 2048 bajtów. Typ *FNameStr* jest zdefiniowany w module *Objects* jako łańcuch o długości 79 znaków (*string*[79]).

destructor *Done*; **virtual**;

Wywołuje metodę *Flush* do opróżniania zawartości bufora na dysk, następnie zwalnia buforowany strumień z pamięci wywołując destruktor dziedziczony z poprzednika.

procedure *Flush*; **virtual**;

Opróżnia zawartość bufora, ustala status strumienia na *stOK*. Jest wywoływana przez destruktor.

function *GetPos*:*LongInt*; **virtual**;

Zwraca aktualną pozycję w strumieniu.

function *GetSize*:*LongInt*; **virtual**;

Opróżnia bufor i zwraca rozmiar strumienia (w bajtach).

procedure *Read*(*var* *Buf*; *Count*:*Word*); **virtual**;

Jeśli status strumienia jest *stOK*, metoda czyta ze strumienia *Count* bajtów do bufora *Buf*, zaczynając od aktualnej pozycji w strumieniu. *Buf* jest wskaźnikiem na zmienną, w której mają być zapamiętane czytane dane.

procedure Seek(Pos:LongInt);**virtual**;

Opróżnia bufor i ustawia aktualną pozycję w strumieniu na *Pos* bajtów. Strumień zaczyna się od pozycji 0.

procedure Truncate;**virtual**;

Opróżnia bufor i usuwa ze strumienia wszystkie dane od aktualnej pozycji w strumieniu wywołując dziedziczną metodę *Truncate*.

procedure Write(var Buf;Count:Word);**virtual**;

Jeśli status strumienia jest *stOK*, metoda zapisuje do strumienia *Count* bajtów z bufora *Buf*, zaczynając od aktualnej pozycji w strumieniu. *Buf* wskazuje na zmienną, która ma być zapisana do strumienia.

7. Tworzenie kolekcji i strumienia

Pierwszym zadaniem, które należy wykonać, jest zdefiniowanie typu obiektów (oraz typu wskaźnikowego na ten obiekt, ponieważ kolekcja jest dynamiczną tablicą wskaźników na obiekty), które będą zapamiętywane w kolekcji. Jeśli kolekcja będzie zapisywana na dysk, obiekt powinien mieć zdefiniowaną metodę *Store*, jeśli w tym samym programie będzie również odczytywana z dysku, potrzebna też jest metoda – definiowana jako konstruktor – *Load*. Metoda *Store* musi prawidłowo zapisywać pola danych obiektu do strumienia, metoda *Load* – odczytywać dane ze strumienia. Dla omawianego wcześniej przykładu figur graficznych, metody te mogą mieć postać:

```
procedure TPoint.Store(var S:TStream);
begin
    S.Write(X,SizeOf(X));           {zapisz pole X do strumienia}
    S.Write(Y,SizeOf(Y));           {zapisz pole Y do strumienia}
end;

procedure TCircle.Store(var S:TStream);
begin
    inherited Store(S);              {albo powtórzyć zapisywanie pól X i Y}
    S.Write(Radius,SizeOf(Radius)); {zapisz pole Radius do strumienia}
end;

constructor TPoint.Load(var S:TStream);
begin
    S.Read(X,SizeOf(X));             {odczytaj pole X ze strumienia}
    S.Read(Y,SizeOf(Y));
end;

constructor TCircle.Load(var S:TStream);
begin
    inherited Store(S)              {albo powtórzyć odczytywanie pól X i Y}
    S.Read(Radius,SizeOf(Radius));
end;
```

- Dla typów, które mają być zapisywane do strumienia (i odczytywane) definiuje się rekord typu *TStreamRec*. Jest to zwykły typ rekordowy, zdefiniowany w module *Objects* następująco:

```
type TStreamRec=
  record
    ObjType:Word;           {unikalny numer rejestracyjny typu}
    VmtLink:Word;           {połączenie z VMT rejestrowanego typu}
    Load:Pointer; {wskaźnik na konstruktor Load rejestrowanego typu}
    Store:Pointer; {wskaźnik na metodę Store rejestrowanego typu}
    Next:Word;             {wskaźnik na następny TStreamRec}
  end;
```

- Pole *ObjType* typu *TStreamRec* zawiera numer rejestracyjny typu. Numery od 0 do 99 są zarezerwowane dla biblioteki *Turbo Vision*, programista może rejestrować typ korzystając z numerów od 100 do 65535. Typy zdefiniowane w *Turbo Vision* (np. *TCollection*) mają już ustalone rekordy rejestracyjne, programista nie musi ich definiować.

Pole *VmtLink* zawiera offset adresu Tablicy Metod Wirtualnych *VMT* rejestrowanego typu. Ustala się go korzystając ze standardowych funkcji *Ofs* i *TypeOf*.

Każdy typ, który będzie korzystał ze strumienia, oprócz zdefiniowanego rekordu rejestracyjnego, musi być zarejestrowany w programie przez wywołanie w bloku programu procedury zdefiniowanej w module *Objects* następująco:

```
procedure RegisterType (var S:TStreamRec);
```

S jest nazwą zdefiniowanego rekordu rejestracyjnego.

Definiowanie rekordów rejestracyjnych dla obiektów graficznych korzystających ze strumienia zapisującego je do pliku może być następujące:

```
Const RPoint:TStreamRec = (
  ObjType:150; {nadawany numer rejestracyjny typu: 150}
  VmtLink:Ofs (TypeOf (TPoint) ^); {adres tablicy VMT}
  Load:@TPoint.Load; {adres konstruktora Load}
  Store:@TPoint.Store); {adres metody Store}

Const RCircle:TStreamRec = (
  ObjType:151;
  VmtLink:Ofs (TypeOf (TCircle) ^);
  Load:@TCircle.Load;
  Store:@TCircle.Store);
```

Rejestracja tych typów (w bloku programu):

```
RegisterType (RCollection); {typ TCollection też musi być zarejestrowany}
RegisterType (RPoint); {dla TCollection jest predefiniowany rek. TStreamRec}
RegisterType (RCircle);
```

Utworzenie kolekcji, to przydzielenie pamięci dla zdefiniowanej wcześniej zmiennej typu *PCollection* z jednoczesnym wywołaniem konstruktora zdefiniowanego w typie *PCollection*:

```
var KObiektówGraficznych:PCollection; {w części definicyjnej}
KObiektówGraficznych:=New (PCollection, Init (10,3); {w bloku programu}
```

W powyższym przypadku utworzona zostanie kolekcja obiektów, z przydzieloną pamięcią wystarczającą do pamiętania wskaźników na 10 obiektów, pamięć tę można powiększyć o 3 obiekty. Utworzona kolekcja jest w tym momencie pusta, nie zawiera żadnych elementów. Korzystając z metody *Insert* można element włączyć do kolekcji. Zakładając, że są zdefiniowane typy *PPoint* i *PCircle* wskazujące odpowiednio na typy *TPoint* i *TCircle* oraz zmienna *P* wskazująca na obiekt zdefiniowany jako pierwszy w hierarchii, można utworzyć dowolny obiekt z hierarchii i dołączyć go do kolekcji jak w poniższym fragmencie:

```
for i:=1 to 10 do
begin
  case i mod 2 of
    0: P:=New(PPoint, Init);
    1: P:=New(PCircle, Init);
  end;
  KObiektowGraficznych^.Insert(P);
end;
```

Do kolekcji *KObiektowGraficznych* mogą być dołączane obiekty różnych typów, jeśli tylko te typy są zarejestrowane. W powyższym fragmencie do kolekcji będą dołączane punkty (*TPoint*) i koła (*TCircle*).

Utworzoną kolekcję można zapisać do strumienia, ponieważ wszystkie konieczne metody zostały już zdefiniowane, typy zarejestrowane. Teraz należy zainicjować strumień:

```
var SObiektowGraficznych:TBufStream;           {w części definicyjnej}
SObiektowGraficznych.Init('Plik.stm', stCreate, 1024); {w bloku programu}
```

Za pomocą powyższej instrukcji został utworzony strumień skojarzony z plikiem o nazwie *Plik.stm* w trybie dostępu: *tworzony nowy strumień*, o rozmiarze 1024 bajtów.

Zapisanie istniejącej kolekcji do strumienia to:

```
SObiektowGraficznych.Put(KObiektowGraficznych); {zapisanie kolekcji
                                                    do pliku}
SObiektowGraficznych.Done; {zwolnienie pamięci zajmowanej przez strumień}
```

Aby odczytać dane ze strumienia, należy zainicjować strumień z dostępem do czytania:

```
SObiektowGraficznych.Init('Plik.stm', stOpen, 1024); {inicjowanie str.}
KObiektowGraficznych:=PCollection(Get); {odczyt kolekcji ze strumienia}
SObiektowGraficznych.Done; {zwolnienie pamięci zajmowanej przez strumień}
```

8. Przykład bazy danych

Ciekawy przykład tworzenia kolekcji obiektów różnych typów, zapisywania jej na dysk i odczytu z dysku, zamieszczony jest w katalogu \DOCDEMOS w TP7.0, pliki *stream1.pas* i *stream2.pas*. Jest to kolekcja obiektów graficznych: punkt, koło, prostokąt, itp.

Poniżej jest zamieszczony program, w którym jest pokazana baza danych – informacje o książkach. Pola danych obiektów zapisywanych przez strumień do pliku są wskaźnikami na wartości, które są danymi o zapisywanej książce. Przykład jest stosunkowo prosty,

niemniej ilustruje możliwości wykonania bazy danych na obiektach.

W przykładzie jest używany identyfikator typu *PString*. Jest to typ zdefiniowany w module *Objects*, określa dynamiczne łańcuchy. Jego deklaracja to:

```
type PString = ^String;
```

Łańcuch taki zajmuje tyle pamięci, ile potrzeba na jego prawidłowe zapamiętanie. Zmiennej typu *PString* przydziela się pamięć procedurą zdefiniowaną w module *Objects* o nagłówku:

```
procedure NewStr(var str:PString);
```

Inne elementy zdefiniowane w module *Objects* nie są wykorzystywane w prezentowanym przykładzie i nie są omawiane w niniejszej pracy.

Przykład pokazuje również możliwości przetwarzania takiej bazy; w tym celu zdefiniowano procedurę *SzukajTytul*. Pokazano przy tej okazji, jak definiuje się funkcję dopasowującą, umożliwiającą wybór z kolekcji elementów spełniających określony warunek, np. zadana wartość określonego pola obiektu. W przykładzie jest to funkcja o nagłówku:

```
function DopasujTytul(Ksiazka:PKsiazka):Boolean; far;
```

Jej adres jest przekazywany do metody *FirstThat* po to, aby wyszukać pierwszy obiekt spełniający określony warunek.

```
program KolekcjaKsiazek;
uses Crt, Objects;

type PKsiazka = ^TKsiazka;           {wskaźnik na obiekt TKsiazka}
TKsiazka = object(TObject) {obekt, który będzie elementem kolekcji}
  Autor, Tytul, Wyd, Gdzie:PString; {wskaźniki na łańcuch}
  RokWyd: ^Integer;                 {wskaźnik na liczbę całkowitą}
  constructor Init(NAutor, NTytul, NWyd, NGdzie:String;
                   NRokWyd:Integer);
  destructor Done; virtual;
  constructor Load(var S:TStream); {do odczytania kolekcji}
  procedure Store(var S:TStream); {do zapisu kolekcji}
end;

constructor TKsiazka.Init(NAutor, NTytul, NWyd, NGdzie:String;
                          NRokWyd:Integer);
{ tworzy obiekt w pamięci: przydziela pamięć dla informacji o książce -
  napisów (autor, tytuł, ...) i roku wydania, ustala wartości pól obiektu -
  wskaźniki na poszczególne informacje o książce }
begin
  Autor:=NewStr(NAutor);             {przydzielanie pamięci polu typu PString}
  Tytul:=NewStr(NTytul);
  Wyd:=NewStr(NWyd);
  Gdzie:=NewStr(NGdzie);
  New(RokWyd);
  RokWyd^:=NRokWyd;
end;

constructor TKsiazka.Load(var S:TStream);
```

```

{ odczytuje dane dla obiektu typu TKsiazka, ustala wartości pól obiektu
jako wskaźników na odczytane informacje o książce }
var A,T,W,G:string; i:integer;
begin
  S.Read(A,SizeOf(Autor^));           {wczytanie łańcucha ze strumienia}
  Autor:=NewStr(A);                    {zapisanie łańcucha, zapamiętanie jego adresu
                                       w polu o nazwie Autor}

  S.Read(T,SizeOf(Tytul^));
  Tytul:=NewStr(T);
  S.Read(W,SizeOf(Wyd^));
  Wyd:=NewStr(W);
  S.Read(G,SizeOf(Gdzie^));
  Gdzie:=NewStr(G);
  S.Read(i,SizeOf(RokWyd^));
  New(RokWyd); RokWyd^:=i;
end;

procedure TKsiazka.Store(var S:TStream);
{ zapisuje do strumienia element kolekcji: informacje o książce
wskazywane przez pola obiektu typu Tksiazka }
begin
  S.write(Autor^,SizeOf(Autor^));
  S.write(Tytul^,SizeOf(Tytul^));
  S.write(Wyd^,SizeOf(Wyd^));
  S.write(Gdzie^,SizeOf(Gdzie^));
  S.write(RokWyd^,SizeOf(RokWyd^));
end;

destructor TKsiazka.Done;
{ zwalnia pamięć zajmowaną przez informacje o książce, których adresy
pamiętane są jako pola obiektu TKsiazka }
begin
  DisposeStr(Autor);
  DisposeStr(Tytul);
  DisposeStr(Wyd);
  DisposeStr(Gdzie);
  Dispose(RokWyd);
end;

procedure DrukujListe(L:PCollection);
{ zwykła procedura, drukuje zawartość całej kolekcji }

procedure DrukujKsiazke(K:PKsiazka);far;
{procedura drukowania pól jednego obiektu typu TKsiazka}
begin
  with K^ do
  begin
    writeln('Autor:   ',Autor^);
    writeln('Tytul:   ',Tytul^);
    writeln('Wydanie: ',Wyd^,' ',RokWyd^);
    writeln('Jest na: ',Gdzie^);
  end;
end;

begin
  writeln(#13#10, ' L I S T A K S I A Z E K':40);
{ teraz wykonaj procedurę drukowania pojedynczego obiektu dla każdego

```



```

elementu kolekcji )
    L^.ForEach(@DrukujKsiazke);
end;

procedure SzukajTytul(L:PCollection;T:String);           {T-szukany tytul}
{ procedura wyszukiwania z kolekcji książki o zadanym tytule }
function DopasujTytul(Ksiazka:PKsiazka):Boolean;far;
{ funkcja dopasowująca }
begin
    DopasujTytul:=Pos(T,Ksiazka^.Tytul^)<>0; {jeśli jest tytuł, zwróć True}
end;

var ZnalezionaKsiazka:PKsiazka;
begin
    ZnalezionaKsiazka:=L^.FirstThat(@DopasujTytul);
    if ZnalezionaKsiazka=nil then
        writeln(#13#10'Nie ma książki o tytule ',T)
    else with ZnalezionaKsiazka^ do
        writeln(#13#10,'Książka o szukanym tytule to: '#13#10,Autor^,' - ',
            Tytuł^,#13#10,Wyd^,' ', 'RokWyd^,' jest na: ',Gdzie^);
end;

var ListaKsiazek:PCollection;           {definicja zmiennych globalnych}
    KsiazkaStream:TBufStream; znak:Char;
const RKsiazka:TStreamRec=(              {rekord rejestracyjny}
    ObjType:150;
    VmtLink:Ofs(KindOf(TKsiazka)^);
    Load:@TKsiazka.Load;
    Store:@TKsiazka.Store);

procedure zaniechaj(Inf:String);           {procedura przerywająca program}
begin
    Writeln(#13#10,Inf);
    writeln('Program zaniechany');
    Halt(1);
end;

procedure czekaj;
begin
    while KeyPressed do znak:=readKey;
    repeat until KeyPressed;
end;

begin {program główny}
    ClrScr;
    RegisterType(RCollection);           {rejestracja typu PCollection}
    RegisterType(RKsiazka);             {rejestracja typu PKsiazka}
    ListaKsiazek:=New(PCollection,Init(10,5)); {tworzenie kolekcji}
    with ListaKsiazek^ do                {dołączanie elementów do kolekcji}
    begin
        Insert(New(PKsiazka, Init('Adam Mickiewicz','Balladyna','PWN',
            'Polka 1',1963)));
        Insert(New(PKsiazka, Init('Gustaw Meyrink','Golem','Wyd.Literackie',
            'Wypożyczona',1981)));
        Insert(New(PKsiazka, Init('Henryk Sienkiewicz','Krzyzacy',
            'Czytelnik','Polka 4',1973)));
    end;
end;

```

```

Insert(New(PKsiazka, Init('Stefan Themerson',
    'Wyklad Profesora MMAA', 'PIW', 'Polka 4', 1958)));
Insert(New(PKsiazka, Init('Jan Parandowski', 'Mitologia',
    'Wyd. Poznanskie', 'Polka 4', 1987)));
end;      {utworzenie kolekcji, tu w programie, może być w procedurze}
DrukujListe(ListaKsiazek);      {drukowanie elementów kolekcji}
czekaj;
SzukajTytul(ListaKsiazek, 'Krzyzacy');      {wyszuk. książki o tytule
                                             Krzyzacy}
czekaj;
KsiazkaStream.Init('ksiazka.stm', stCreate, 1024);      {utworzenie
                                                         strumienia}
KsiazkaStream.Put(ListaKsiazek);      {zapisanie kolekcji do strumienia}
KsiazkaStream.Done;      {"zamknięcie" strumienia, zwolnienie pamięci}
Dispose(ListaKsiazek, Done);      {usunięcie kolekcji z pamięci}
writeln('Z A P I S A N E   N A   D Y S K':60);
czekaj;
with KsiazkaStream do
begin
    Init('ksiazka.stm', stOpen, 1024); {utworz. strum., status do odczytu}
    ListaKsiazek:=PCollection(Get); {odczyt zapisanej wcześniej kolekcji}
    Done;      {"zamknięcie" strumienia}
    if Status <> 0 then zaniechaj('Blad'); {przerwanie, gdy wystąpił błąd}
end;
writeln('#13#10, 'O D C Z Y T A N E   Z   D Y S K U':60);
DrukujListe(ListaKsiazek);      {drukowanie zawartość odczytanej kolekcji}
czekaj;
Dispose(ListaKsiazek, Done);      {usunięcie kolekcji z pamięci}
end.

```

Wyniki wykonania powyższego programu to następujący wydruk na ekranie monitora:

LISTA KSIĄZEK

Autor: Adam Mickiewicz
 Tytuł: Balladyna
 Wydanie: PWN, 1963
 Jest na: Polka 1
 Autor: Gustaw Meyrink
 Tytuł: Golem
 Wydanie: Wyd.Literackie, 1981
 Jest na: Wypożyczona
 Autor: Henryk Sienkiewicz
 Tytuł: Krzyzacy
 Wydanie: Czytelnik, 1973
 Jest na: Polka 4
 Autor: Stefan Themerson
 Tytuł: Wyklad Profesora MMAA
 Wydanie: PIW, 1958
 Jest na: Polka 4
 Autor: Jan Parandowski
 Tytuł: Mitologia
 Wydanie: Wyd. Poznanskie, 1987
 Jest na: Polka 4

Książka o szukanym tytule to:
Henryk Sienkiewicz - Krzyzacy
Czytelnik, 1973 jest na: Polka 4

ZAPISANE NA DYSK

ODCZYTANE Z DYSKU

LISTA KSIĄZEK

Autor: Adam Mickiewicz
Tytuł: Balladyna
Wydanie: PWN, 1963
Jest na: Polka 1
Autor: Gustaw Meyrink
Tytuł: Golem
Wydanie: Wyd.Literackie, 1981
Jest na: Wypożyczona
Autor: Henryk Sienkiewicz
Tytuł: Krzyzacy
Wydanie: Czytelnik, 1973
Jest na: Polka 4
Autor: Stefan Themerson
Tytuł: Wykład Profesora MMAA
Wydanie: PIW, 1958
Jest na: Polka 4
Autor: Jan Parandowski
Tytuł: Mitologia
Wydanie: Wyd. Poznańskie, 1987
Jest na: Polka 4

W programie zamieszczono wiele komentarzy, pokazano wydruk wyników, zatem analiza programu nie powinna nastręczać trudności, pozostawiamy ją czytelnikowi.

Dodatek B

KRÓTKI PRZEGLĄD RÓŻNIC JĘZYKA TP7.0 W STOSUNKU DO TP6.0

Zgodnie z podręcznikiem *USER'S GUIDE TP7.0*, dostarczonym razem z pakietem TP7.0, rozszerzenie języka w wersji 7.0 obejmuje zarówno Zintegrowane Środowisko Turbo Pascala, jak i możliwości językowe.

1. Dwa zintegrowane środowiska

- **TPX.EXE** – pracuje w trybie ochrony DOSa (*DOS protected mode*). W środowisku tym mogą być kompilowane i uruchamiane nawet bardzo duże aplikacje. Jednakże minimalne wymagania sprzętowe dla tego środowiska, to co najmniej mikroprocesor 80286 i 2 MB pamięci RAM.
- **TURBO.EXE** – środowisko nie ma wymagań odnośnie do sprzętu, podobnie jak TP6.0. Duże programy mogą nie mieścić się w pamięci, należy wtedy stosować moduły nakładkowe lub kompilować na dysk i uruchamiać z poziomu DOSa plik *.EXE.

1.1. Nowe cechy środowiska

- **ObjectBrowser** – umożliwia przeglądanie obiektów, modułów, procedur, funkcji, typów, zmiennych, stałych, i innych symboli w używanym programie. Okienko *ObjectBrowser* pojawia się po wybraniu opcji *Search|Objects*. Można wyświetlić hierarchię obiektów, a dla wybranego obiektu – wszystkie procedury i funkcje odwołujące się do niego, itp. Należy przy tym mieć włączone opcje kompilatora: *Debug Information*, *Local Symbols* i *Symbol Information*.
- **Syntax highlighting** – umożliwia oznaczanie kolorem określonych symboli w programie, np. słowa kluczowe są wyświetlane innym kolorem niż identyfikatory czy komentarze. Oznaczenie to jest możliwe również na wydrukach programów. Użytkownik może wybrać kolory w okienku dialogowym, które pojawi się w opcji *Option|Environment|Colors*.
- **Undo i Redo** – umożliwia unieważnienie edycji programu w momencie, gdy popełnimy błąd (*Edit|Undo*), jeśli okaże się, że jednak chcemy odzyskać dokonaną zmianę – wybieramy *Edit|Redo*.
- **Tools menu** – umożliwia dodanie do menu *Tools* wybranych przez użytkownika programów. Programy te można uruchamiać będąc w środowisku TP7.0. Dodawanie takich programów odbywa się przez okienko dialogowe pojawiające się po wyborze *Option|Tools*.

Jeśli chcemy, by nasz dodany program był uruchamiany ilekroć wciśnięte zostaną „gorące klawisze”, to można to określić przez wybór jednej z opcji Hot Key.

- *Messages window* (Tools|Messages) – okienko to może być wykorzystywane do wyświetlania wyjściowych informacji z niektórych programów narzędziowych (np. GREP), które wysyłają informacje przez filtr DOSa. Filtr ten dokonuje konwersji wyjścia programu do postaci, która może być wyświetlona w tym okienku. Przykładem takiego filtru jest program dostarczany z pakietem TP7.0 o nazwie GREP2MSG.PAS, znajdujący się w kartotece UTIL. Okienko to może być przewijane, można wybrać do edycji linię programu związaną z wyświetlaną informacją.

- *Local menu* – otwarcie tego okienka powoduje wyświetlenie listy komend menu specyficznego dla aktywnego okienka.

- *Multiple user-installable Help file support* – możliwość dodawania do systemu *Help* TP7.0 innych systemów pomocy *Help*. Aby pojawiło się dialogowe okienko instalacyjne, należy wybrać opcję *Help|Files*.

- Zachowanie informacji o symbolach w czasie sesji w pliku *.PSM – po ponownym uruchomieniu sesji (TP7.0) można od razu przeglądać program, śledzić (*debug*), itp. Wyboru opcji dokonuje się przez *Option|Environment|Preferences*; w grupie *Desktop File Options group* należy wybrać *Desktop* i *Symbols*. Opcja *Auto Save Desktop* powinna być przy tym włączona.

- Zachowanie informacji o symbolach w czasie kompilacji – jeśli aktualna kompilacja nie zakończy się sukcesem, będą dostępne symbole z poprzedniej kompilacji. Można więc będzie użyć *ObjectBrowser*, aby łatwiej zlokalizować problem. W tym celu powinna być włączona opcja *startup /Y*. Opcja ta standardowo jest włączona, dotyczy ona tylko środowiska TPX.EXE.

2. Nowe dyrektywy kompilatora

W wersji TP7.0 są dodane cztery dyrektywy kompilatora:

- *Open String Parameters* – **\$P**. Dyrektywa lokalna, umożliwia stosowanie „otwartych parametrów” (patrz rozdz. 5).

Włączenie: **{SP+}** lub zaznaczenie X przy *Open String Parameters* w okienku dialogowym, pojawiającym się po wybraniu opcji *Opiton|Compiler*.

Wyłączenie: **{SP-}** lub likwidacja znaku X w opcjach kompilatora.

Działanie: Kontroluje znaczenie parametrów deklarowanych jako **string**. W stanie **{SP-}** parametry deklarowane jako **string** są traktowane jak zwykłe zmienne typu **string**, natomiast w stanie **{SP+}** parametry deklarowane jako **string** są parametrami o nieokreślonej długości łańcucha (patrz rozdz. 5.2).

Domyślnie: **{SP-}**.

- *Type-checked Pointers* – **\$T**. Dyrektywa globalna, steruje typem wartości wskazywanej, generowanej przez operator adresowy **@**.

Włączenie: **{ST+}** lub znak X w polu *Typed @ operator* w opcjach *Opiton|Compiler*.

Wyłączenie: {**ST**-} lub likwidacja znaku X w opcjach kompilatora.

Działanie: Przy wyłączonej dyrektywie, operator @ zawsze zwraca wartość typu *Pointer* – typu wskaźnikowego zgodnego z każdym typem wskaźnikowym. Jeśli w stanie {**ST**+) będzie odwołanie @V, gdzie V jest zmienną typu T1, wynik będzie typu ^T1, czyli typu wskazującego na typ zmiennej V.

Domyślnie: {**ST**-}.

■ *Overflow Checking* – **\$Q**. Dyrektywa lokalna, steruje generowaniem kodu sprawdzania zakresów dla pewnych arytmetycznych operacji całkowitych.

Włączenie: {**\$Q**+) lub w polu *Overflow checking* w opcjach Opiton|Compiler.

Wyłączenie: {**\$Q**-} lub likwidacja znaku X w opcjach kompilatora.

Działanie: W stanie {**\$Q**+) niektóre operacje arytmetyczne wykonywane na wartościach całkowitych (+, -, *, *Abs*, *Sqr*, *Succ*, *Pred*) są kontrolowane pod kątem przekroczenia zakresów. Kod dla każdej z tych operacji jest poprzedzany dodatkowym kodem, który sprawdza, czy wynik mieści się w odpowiednim przedziale wartości. Jeśli nie, program jest zatrzymywany i generowany jest błąd *run-time error*. Procedury *Inc* i *Dec* nie podlegają kontroli odnośnie zakresów. Dyrektywa **\$Q** jest używana zwykle razem z dyrektywą **\$R**, która umożliwia lub nie generowanie kodu z kontrolą zakresów.

Włączenie tej dyrektywy powoduje zwiększenie kodu programu i spowolnienie jego pracy, dlatego dla przetestowanych programów należy **\$Q** i **\$R** wyłączyć.

Domyślnie: {**\$Q**-}.

■ *Symbol Information* – **\$Y**. Dyrektywa globalna, umożliwia – lub nie – generowanie informacji o odwołaniach do symboli (patrz rozdz. 1.1, ostatni akapit).

Odnosi się tylko do środowiska TPX.EXE

Włączenie: {**\$Y**+) lub znak X w polu *Symbol Information* w opcjach Opiton|Compiler.

Wyłączenie: {**\$Y**-} lub likwidacja znaku X w opcjach kompilatora.

Działanie: W stanie {**\$Y**+) generowana jest informacja w postaci tabeli zawierającej numery linii wszystkich deklaracji i odwołań do symboli w kompilowanej części (programie lub module). Dla modułu, informacja ta umieszczana jest w pliku *.TPU, powiększa to jego rozmiar, ale nie wpływa na rozmiar i szybkość pracy programu wykonawczego (*executable*).

Dyrektywę tę stosuje się zwykle w połączeniu z dyrektywami **\$D** i **\$L** (*Debug Information* i *Local Symbol*). Bez włączonych obu wymienionych dyrektyw, **\$Y** nie dałaby żadnych efektów.

Domyślnie: {**\$Y**+).

3. Parametry procedur lub funkcji deklarowane jako stałe (ang. constant parameters)

W liście parametrów formalnych procedury lub funkcji może znajdować się deklaracja stałej, z użyciem słowa kluczowego **const**. Nagłówek procedury (funkcji) ma postać odpowiednio:

```
procedure nazwa_p(const lista_identyfikat:nazwa_typu_param);
function nazwa_f(const lista_identyfikat:nazwa_typu_param):typ_funkcji;
```

Stały parametr (*constant parameter*) zachowuje się jak lokalna zmienna tylko do odczytu (*read-only*). Wewnątrz procedury (funkcji) można pobrać wartość parametru aktualnego odpowiadającego danemu parametrowi formalnemu, ale nie można tej wartości zmienić. Taki parametr nie może wystąpić po lewej stronie instrukcji przypisania ani być przekazywany jako parametr aktualny w wywołaniu innej procedury czy funkcji. Parametr formalny i odpowiadający mu parametr aktualny muszą umożliwiać wykonywanie tych samych operacji.

W sytuacjach, gdy nie trzeba zmieniać wartości parametru formalnego wewnątrz procedury (funkcji) powinno się używać *constant parameter* zamiast przekazywania parametru przez wartość. Przemawiają za tym dwie przesłanki:

- niemożliwa będzie przypadkowa zmiana wartości parametru, np. poprzez przeoczenie,
- kompilator generuje efektywniejszy kod wynikowy dla *constant parameters* niż dla parametrów przekazywanych przez wartość.

Przykład 5.1b (Dodatek B, rozdz. 5.1) pokazuje użycie funkcji ze stałym parametrem.

4. Nowe procedury i funkcje w module System

Poza ulepszeniem niektórych procedur (szybsze operacje we/wy na plikach tekstowych, szybsze działanie funkcji *Pos*, optymalizacja niektórych działań na *LongInt* na mikroprocesorze 80386) w module *System* dodane są następujące procedury i funkcje:

- Funkcja określająca czy wskaźnik lub zmienna proceduralna wskazuje **nil**. Nagłówek:

```
function Assigned (var P):Boolean;
```

P – zmienna typu wskaźnikowego lub proceduralnego. Funkcja zwraca wartość *True*, jeśli *P* nie wskazuje **nil** lub *False*, jeśli *P* wskazuje **nil**. Wywołanie *Assigned(P)* jest równoważne sprawdzeniu czy $P \neq \text{nil}$ dla zmiennej wskaźnikowej *P* oraz czy $@P \neq \text{nil}$ dla zmiennej proceduralnej *P*.

- Procedura przerywająca wykonywanie pętli programowych. Nagłówek:

```
procedure Break;
```

Procedury tej można używać tylko wewnątrz instrukcji **for**, **while**, **repeat**, w przeciwnym wypadku kompilator będzie sygnalizował błąd. Działanie tej procedury jest równoznaczne z działaniem instrukcji skoku bezwarunkowego **goto** do etykiety umieszczonej przed instrukcją pierwszą po instrukcji pętli. W programowaniu strukturalnym nie należy używać instrukcji skoku bezwarunkowego, natomiast *Break* jest wygodnym narzędziem do natychmiastowego opuszczenia pętli.

Przykład 4.a.

```
var S:string;
begin
```

```

while True do
  begin
    readln( S);
    if S='' then Break;           (jeśli łańcuch pusty, przerwij pętlę)
    writeln(S)
  end
end.

```

■ Procedura nakazująca rozpoczęcie wykonywania kolejnej iteracji. Nagłówek:

```
procedure Continue;
```

Podobnie jak poprzednia, procedura ta może być użyta tylko wewnątrz instrukcji **for**, **while**, **repeat**, w przeciwnym wypadku kompilator będzie sygnalizował błąd. Wywołanie tej procedury wewnątrz **for**, **while** lub **repeat** spowoduje zaniechanie dalszych czynności wewnątrz pętli i rozpoczęcie wykonywania kolejnej iteracji.

Przykład 4.b.

```

var i: integer;
    nazwa: string[79];
    f:file;

begin
  for i:=1 to ParamCount do
    begin
      nazwa:=ParamStr(i);
      Assign(f,nazwa);
      {$I-}
      Reset(f,1);
      {$I+}
      if IOResult <>0 then
        begin
          Writeln('Nie odnaleziony plik: ',nazwa);
          Continue;      (zaczynij od początku kolejną iterację (kolejne i))
        end;
      Writeln(nazwa,': ',FileSize(f),' bajtów');
      Close(f);
    end;
  end.

```

*W przykładzie założono, że jest uruchomiony program z parametrami – nazwami plików. Jeśli plik o nazwie podanej jako parametr istnieje, należy określić jego rozmiar. Jeśli jakiś plik nie zostaje odnaleziony, następuje przejście do pierwszej instrukcji wewnątrz **for**, czyli sprawdzanie kolejnej nazwy pliku.*

■ Procedura włączająca element do zbioru. Nagłówek:

```
procedure Include(var S:set of T; i:T);
```

S jest zmienną typu zbiorowego, przy czym typem bazowym jest typ T , parametr i jest wyrażeniem typu T . Działanie tej procedury jest równoważne instrukcji: $S:=S+[i]$; jednak generowany jest efektywniejszy kod programu. Procedura powoduje dołączenie do zbioru wartości wskazanej jako drugi parametr.

■ Procedura usuwająca element ze zbioru. Nagłówek:

```
procedure Exclude (var S:set of T; i:T);
```

Element podany jako parametr *i* zostanie usunięty ze zbioru *S*, zatem działanie tej procedury jest równoważne instrukcji *S:=S-[i]*; podobnie jak przy *Include*, generowany jest efektywniejszy kod.

■ Funkcja zwracająca największą dopuszczalną wartość argumentu (lub pierwszego indeksu argumentu). Nagłówek:

```
function High (X):T;
```

gdzie *T* jest typem *X* lub indeksów *X*, zależnie od tego, czym jest ten parametr. *X* może być identyfikatorem typu lub identyfikatorem zmiennej. Typ o identyfikatorze *X*, lub typ zmiennej *X*, musi być typem porządkowym, tablicowym lub łańcuchowym. Dla typów porządkowych funkcja *High* zwraca najwyższą wartość z przedziału tego typu, dla typu tablicowego – zwraca najwyższą wartość pierwszego indeksu typu tablicowego, dla łańcuchów – deklarowany rozmiar łańcucha. Dla *open-array* lub *open-string* (tablic o nieokreślonej liczbie elementów lub łańcuchów o nieokreślonej maksymalnej długości, patrz rozdz. 5 tego dodatku) funkcja *High* zwraca wartość typu *Word*, określającą liczbę elementów aktualnego parametru pomniejszoną o jeden.

Przykład 4.c.

Definicje typów lub zmiennych:

Wartości funkcji *High*

<code>type T = (a,b,c);</code>	<code>High(T) = c</code>
<code>T = 5..10;</code>	<code>High(T) = 10</code>
<code>T = array[1..11,1..17] of byte;</code>	<code>High(T) = 11</code>
<code>T = array['a'..'z',1..9] of byte;</code>	<code>High(T) = 'z'</code>
<code>var V:'1'..'a';</code>	<code>High(V) = 'a'</code>
<code>var V:byte;</code>	<code>High(V) = 255</code>

Jak widać z powyższych przykładów, dla tablicy dwuwymiarowej, funkcja *High* zwraca maksymalną wartość rozmiaru pierwszego indeksu. Dla *open-parameters* przykłady są zamieszczone w rozdz. 5.1.

■ Funkcja zwracająca najmniejszą wartość argumentu. Nagłówek:

```
function Low (X):T;
```

gdzie *T* – typ *X* lub indeksów typu *X*, jak dla funkcji *High*. *X* może być identyfikatorem typu lub identyfikatorem zmiennej. Typ o identyfikatorze *X* lub typ zmiennej *X* musi być typem porządkowym, tablicowym lub łańcuchowym. Dla typów porządkowych funkcja *Low* zwraca najmniejszą wartość z przedziału tego typu, dla typu tablicowego – zwraca najmniejszą wartość pierwszego indeksu typu tablicowego, dla łańcuchów – zawsze zwraca 0. Dla *open-array* lub *open-string* (tablic o nieokreślonej liczbie elementów lub łańcuchów o nieokreślonej maksymalnej długości, patrz rozdz. 5) funkcja *Low* zwraca wartość 0.

*Przykład 4.d.**Zerowanie tablicy może być wykonane:*

```

var A:array[1..100] of word;
    i:byte;
begin
    for i:=Low(A) to High(A) do A[i]:=0;
end.

```

5. Rozszerzenia języka dotyczące stosowania parametrów o nie ustalonej liczbie elementów

5.1. Tablice o nie ustalonej liczbie elementów (ang. open-array parameter)

Jednym z istotnych rozszerzeń jest możliwość przetwarzania tablic o różnej liczbie elementów przez te same procedury. Zmienne typu tablicowego, jak każde inne, mogły być stosowane jako parametry wywołania (parametry aktualne) procedury lub funkcji, jeśli ich typ był identyczny z typem parametru formalnego tej procedury.

✂ Zmienne są identycznych typów jeśli:

- są zadeklarowane na tej samej liście deklaracji, tzn. ten sam identyfikator określa ich typ,
- gdy do ich deklaracji są użyte różne identyfikatory typu, przy czym identyfikatory te oznaczają ten sam typ:

Przykład 5.1.a

```

(1) type string10 = string[10]; var zml,zm2:string10;
    type Tab1 = array[1..10] of byte; var zm3,zm4:Tab1;
(2) type string10 = string[10]; T1 = string10; var zml:T1;
    var zm2:string10;
    type Tab1 = array[1..10]; Tab2 = Tab1; var zm3:Tab1; var zm:Tab2;

```

Zarówno w (1), jak i w (2) zmienne zml i zm2 oraz zm3 i zm4 są identycznych typów.

Parametr procedury lub funkcji może być deklarowany jako parametr o otwartej liczbie elementów tablicy tylko wtedy, gdy jest włączona dyrektywa kompilatora {SP+}. Typ takiego parametru określa się jak typ tablicowy, bez podawania typu indeksów:

```
var parametr: array of określenie_typu_elementów;
```

Wewnątrz procedury lub funkcji, tablica taka będzie się zachowywać jak tablica zadeklarowana:

```
array[0..N-1] of określenie_typu_elementów;
```

gdzie N jest liczbą elementów tablicy będącej parametrem aktualnym.

- ✂ Zakres indeksów parametru aktualnego jest odwzorowywany na przedział liczb całkowitych od 0 do $N-1$. Zatem zawsze parametr aktualny jest traktowany jako tablica

jednowymiarowa i tylko jednowymiarowe tablice mogą być parametrami *open-array*! Jeśli przekazywany parametr (aktualny) jest pojedynczą zmienną typu T, to jest on traktowany jako jednoelementowa tablica typu T.

Przykład 5.1.b.

```
type T1 = array['a'..'d'] of real;
var t:T1;

procedure PiszTab(var z:array of real);
var i:byte;
begin
  for i:=0 to High(z) do write(z[i]:6:2);
end;
...
PiszTab(t);                                     {wywołanie w bloku programu}
{tablica t będzie traktowana jak tablica zadeklarowana:
 t:array[0..3] of real}
```

Tablice o nie ustalonej liczbie elementów (*open-array*) mogą być przekazywane przez wartość, stałą (*constant parameter*) lub zmienną, jak zwykle parametry regularne (tablice o ściśle ustalonym typie), z tymi samymi ograniczeniami, np. otwartemu parametrowi przekazywanemu jako stała (**const**) nie można przypisać wartości w instrukcji przypisania.

Przy przekazywaniu „otwartych tablic” przez wartość tworzy się w segmencie pamięci przeznaczonym na zmienne lokalne (stos, ang. *stack*) zmienna lokalna, będąca kopią parametru aktualnego. Należy zwracać uwagę, by nie doprowadzić do przepełnienia stosu.

Standardowa funkcja *Low* zastosowana z parametrem *open-array* zawsze zwraca zero, standardowa funkcja *High* – zwraca wartość najwyższego indeksu aktualnego parametru (odwzorowanego automatycznie na przedział liczb całkowitych od 0 do N-1), będzie to więc wartość N-1. Standardowa funkcja *SizeOf* zwraca rozmiar (w bajtach) aktualnego parametru.

Jeśli elementy tablicy – parametru *open-array* – są typu *Char*, to można w odwołaniach (jako parametr aktualny) stosować napis (patrz przykład 5.1d).

Przykład 5.1.c.

```
type Tt1 = array[1..5] of byte;
var t1:Tt1;
    j:byte;

procedure UstalWartosc_1(var t:array of byte);           {open-array}
var i:byte;
begin
  for i:=Low(t) to High(t) do t[i]:=i; {Zamiast Low(t) można użyć wartość 0}
end;

procedure UstalWartosc_2(var t:array of byte;znak:Char);
var i: byte;
begin
```

```

    for i:=0 to High(t) do {użyta jest wprost wartość 0 jako dolny indeks}
        FillChar(t,SizeOf(t),znak);           {wypełnia znakiem komórki pamięci}
    end;
procedure DrukujTablice(var t:Tt1);
begin
    for j:=1 to 5 do write(t1[j]:4);
end;
function Suma(const t:array of byte):integer;{liczy sumę elementów tablicy}
var i,S:integer;
begin
    S:=0;
    for i:=0 to High(t) do S:=S+t[i];
    Suma:=S;
end;
begin
    ClrScr;
    UstalWartosc_1(t1);           {ustali wartości odpowiednio: 0,1,2,3,4}
    DrukujTablice(t1);           {wydrukuje wartości: 0 1 2 3 4}
    writeln;writeln('Suma elementow: ',Suma(t1),#13#10);      {wypisze: 10}
    UstalWartosc_2(t1,'5');      {znak '5' ma kod 53, zatem tak będą
    interpretowane wartości elementów tablicy zdefiniowanej jako of byte}
    DrukujTablice(t1);
    writeln;writeln('Suma elementow: ',Suma(t1),#13#10);      {wypisze: 265}
end.

```

W powyższym przykładzie, w funkcji Suma parametr mógłby być przekazywany również przez wartość; wtedy jej nagłówek byłby następujący:

```
function Suma(t:array of byte):integer;
```

Przykład 5.1.d.

```

var l:Char;
const st:Char='S';
procedure PiszTab(const s:array of Char);
var i:byte;
begin
    for i:=0 to High(s) do
        if s[i]<>#0 then write(s[i]) else Break;
end;
...
PiszTab(st);           {w bloku programu}
l:='V';               {wypisze: S}
PiszTab(l);           {wypisze: V}
PiszTab('napis');     {wypisze: napis}
PiszTab('A');         {wypisze: A}
...

```

5.2. Parametry łańcuchowe o nie określonej długości łańcucha (ang. open-string parameters)

Podobnie jak dla innych zmiennych, tak i dla zmiennych typu łańcuchowego, typy parametrów formalnych i aktualnych procedur i funkcji muszą być identyczne. W innym przypadku wystąpi błąd kompilacji wynikający z niezgodności typów.

Wersja TP7.0 daje możliwość deklarowania parametru procedury jako *OpenString*, co oznacza, że parametrem aktualnym może być dowolny łańcuch znaków, o różnej długości.

■ Parametry o nie określonej długości łańcucha (*open-string*) mają znaczenie tylko przy parametrach przekazywanych przez zmienną.

Deklaracji takich parametrów możemy dokonywać na dwa sposoby:

- Stosując identyfikator *OpenString*.
- Stosując słowo kluczowe **string** i dyrektywę {SP+}.

Identyfikator *OpenString* jest deklarowany w module *System*. Oznacza on specjalny typ łańcucha, który może być stosowany w odniesieniu tylko do parametrów łańcuchowych. Identyfikator ten nie jest słowem kluczowym i może być przesłonięty przez programistę. Dyrektywa kompilatora {SP+} zmienia znaczenie słowa kluczowego **string**: zmienne deklarowane jako **string** są *OpenString* parametrami. Dla parametru *OpenString* aktualny parametr wywołania może być zmienną każdego typu łańcuchowego. Wewnątrz procedury lub funkcji atrybut *rozmiar* (maksymalna długość łańcucha) parametru formalnego będzie taki jak parametru aktualnego, patrz poniższy przykład:

Przykład 5.2.a.

```
var S1:string[10];
    S2:string[20];

procedure UstalWartoscLancucha (var S:OpenString);
begin
    S:='1234567890ABCDEFGHijkl'           {nadaje wartość łańcuchowi S o
                                           długości=22 znaki}
end;

begin
    UstalWartoscLancucha(S1);              {ustali S1='1234567890' - 10 znaków}
    UstalWartoscLancucha(S2); {ustali S2='1234567890ABCDEFGHIJ' - 20 znaków}
end.
```

Do parametrów *OpenString* można stosować standardowe funkcje *Low* i *High*. Funkcja *Low* zawsze zwraca wartość równą zero, natomiast funkcja *High* zwraca deklarowaną maksymalną długość aktualnego parametru. Funkcja *SizeOf* zwraca rozmiar aktualnego parametru.

Przykład 5.2b.

Poniższa procedura realizuje wypełnienie zadany znakiem łańcucha będącego

parametrem wywołania procedury stosującej parametr typu `OpenString`:

```
procedure WypelnijLancuch (var S:OpenString;znak:Char);
begin
  S[0]:=Chr(High(S));      {ustal długość łańcucha-pierwszy bajt zmiennej}
  FillChar(S[1],High(S),znak);    {wypełnij, poczynawszy od bajtu S[1],
                                   High(S) bajtów pamięci znakiem znak}
end;
```

- ☛ Parametry procedur i funkcji przekazywane przez wartość lub jako stałe (**const**) deklarowane z użyciem identyfikatora *OpenString* (lub z użyciem słowa kluczowego **string** i dyrektywą **{SP+}**) nie są traktowane jako parametry o otwartej długości łańcucha. Są one traktowane tak, jakby były deklarowane z użyciem słowa kluczowego **string**, czyli jak łańcuchy o długości 255 znaków. Standardowa funkcja *High* zastosowana do takich parametrów zawsze zwraca wartość równą 255.

Dodatek C

PODZIAŁ PAMIĘCI PODCZAS URUCHAMIANIA PROGRAMU NAPISANEGO W JĘZYKU TURBO PASCAL (PLIKU *.EXE) Z POZIOMU SYSTEMU OPERACYJNEGO DOS

Pamięć operacyjna komputera jest zajmowana – w kierunku rosnących adresów pamięci – przez następujące segmenty programu:

1.SWP – Segment Wstępny Programu (ang. *Program Segment Prefix*). Zajmuje on 256 bajtów, tworzony jest przez system operacyjny DOS podczas wczytywania programu *.EXE do pamięci. Jest to segment umożliwiający komunikację pomiędzy systemem operacyjnym a programem. Adres tego segmentu jest pamiętany w predefiniowanej zmiennej *PrefixSeg* (typu Word).

2. SKP – Segment Kodu Programu (ang. *Program Code Segment*). Składa się z następujących części:

2.1. Segment kodu programu głównego.

2.2. Segmenty kodów modułów zadeklarowanych w programie. Moduły są pamiętane w kolejności odwrotnej, niż są zapowiadane w programie (kolejność występowania na liście modułów po słowie kluczowym *uses*).

2.3. Segment standardowego modułu *System*. Jego użycia nie należy zapowiadać w programie głównym. Ten moduł zawsze jest używany i jest pamiętany jako ostatni z używanych w programie modułów.

✚ Żaden z występujących w p. 2. segmentów kodu nie może zajmować więcej niż 64kB pamięci, lecz łączny obszar pamięci zajmowanej przez wszystkie te segmenty razem jest ograniczony jedynie dostępną w komputerze pamięcią operacyjną.

3. SDP - Segment Danych Programu (ang. *Data Segment*), w którym pamiętane są:

3.1. Wszystkie literały zmienne (zmienne z wartością początkową) zdefiniowane w programie głównym i modułach używanych przez program.

3.2. Wszystkie zmienne globalne zadeklarowane w częściach opisowych programu głównego i używanych w programie modułów.

✚ Rozmiar Segmentu Danych Programu nie może być większy niż 64 kB. Adres początku tego segmentu jest pamiętany w rejestrze DS, zawartość tego rejestru nie może ulec zmianie podczas wykonywania programu. Standardowa funkcja *Dseg* typu Word (function *Dseg*: Word) zwraca zawartość tego rejestru

✚ Segmenty wymienione w punktach 2.1, 2.2, 2.3 oraz 3.1 łącznie stanowią zawartość pliku *.EXE programu.

4. SS – Segment Stosowy (ang. *Stack Segment*). Przeznaczony jest na zmienne

lokalne, deklarowane wewnątrz procedur i funkcji.

W segmencie stosowym, kolejne zmienne lokalne są zapamiętywane w kierunku malejących adresów pamięci. Adres ostatniego zajętego bajtu pamięci w segmencie stosowym jest pamiętany w dwóch rejestrach: w rejestrze SS jest pamiętany adres segmentu (nie ulega on zmianie w trakcie wykonywania programu), w rejestrze SP – adres względny w segmencie. Zawartość rejestru SP zmienia się w trakcie wykonywania programu i zależy od liczby zmiennych lokalnych, którym aktualnie przydzielona jest pamięć. Im więcej pamięci zajmują zmienne lokalne, tym rejestr SP zawiera niższy adres.

Rozmiar segmentu stosowego standardowo jest ustalony na 16 kB. Może on być zmieniany w okienku wejściowym *Stack size* polecenia *Memory sizes* opcji *Options* menu systemu Turbo Pascala lub dyrektywą kompilatora **\$M**, przy czym rozmiar ten nie może być większy niż 64 kB.

Po uruchomieniu programu rejestry SS i SP wskazują na pierwszy bajt powyżej segmentu stosowego. Zawartość rejestru SS zwracana jest przez standardową funkcję *SSeg*, a rejestru SP przez funkcję *SPtr* – obie te funkcje są typu *Word*.

5. BN – Bufor Nakładkowy (ang. *Overlay Buffer*). Jest to część pamięci przeznaczona do przechowywania segmentów nakładkowych programu, o ile program takie posiada. Standardowo rozmiar tego bufora jest równy rozmiarowi największego segmentu nakładkowego. Jeśli program nie posiada struktury nakładkowej (nie ma segmentów nakładkowych) to rozmiar bufora nakładkowego jest równy 0. Rozmiar bufora nakładkowego można zmienić standardową procedurą *OvrSetBuf*. Należy zwrócić uwagę, że zwiększenie bufora nakładkowego powoduje zmniejszenie dostępnej pamięci dla zmiennych dynamicznych. Cały bufor nakładkowy jest bowiem umieszczony w części pamięci operacyjnej komputera przeznaczonej na zmienne dynamiczne, nazywanej często *stertą* (ang. *Heap*). Adres początku i końca bufora nakładkowego jest pamiętany odpowiednio w standardowych zmiennych *OvrHeapOrg* i *OvrHeapEnd*, obie typu *Word*.

6. SZD – Stos Zmiennych Dynamicznych (ang. *Heap*), zwany *stertą* lub *składem*. Przeznaczony jest do pamiętania zmiennych dynamicznych, którym pamięć jest przydzielana przez wywołanie standardowych procedur *New* i *GetMem*. Aktualny rozmiar *sterty* (*składu*) zależy od określenia jego minimalnej i maksymalnej wielkości. Wielkości te określa się przez dyrektywę kompilatora **\$M** lub w podokienkach wejściowych *Low heap limit* i *High heap limit* polecenia *Memory sizes* opcji *Options* menu systemu TP6.0. Standardowo przyjmowana jest minimalna wielkość *sterty* równa 0, maksymalna = 640 kB, co oznacza, że jest to cała dostępna pamięć.

Ustalanie wielkości poszczególnych bloków (segmentów) pamięci

Programista może określać rozmiar Segmentu Stosowego (blok SS) oraz minimalny i maksymalny rozmiar *sterty* (blok SZD).

Do tego celu służy globalna dyrektywa kompilatora **\$M**. Zastosowanie tej dyrektywy na poziomie modułu nie da efektów. Dyrektywa **\$M** ma następującą postać:

{**\$M** rozmiar_stosu,min_rozmiar_sterty,max_rozmiar_sterty}

gdzie:

rozmiar_stosu określa w bajtach rozmiar segmentu stosowego, musi być liczbą z przedziału [1024,65520]. Standardowo wielkość ta równa jest 16384 (16 kB).

min_rozmiar_sterty określa w bajtach minimalny rozmiar *sterty* - segmentu przeznaczonego na zmienne dynamiczne. Musi być liczbą z przedziału [0,655360]. Standardowo równy jest 0.

max_rozmiar_sterty określa w bajtach maksymalny rozmiar *sterty*, musi być z przedziału [*min_rozmiar_sterty*,655360]. Standardowa wartość *max_rozmiar_sterty* wynosi 655360.

Te same wartości można ustalić przez parametr *Memory sizes* opcji *Compiler* w opcji *Options* w menu systemu.

Literatura

- [1] *Borland Pascal with Objects, version 7.0. Programmer's Reference*. Scotts Valley: Borland International, Inc., 1992.
- [2] *Borland Pascal with Objects, version 7.0. Language Guide*. Scotts Valley: Borland International, Inc., 1992.
- [3] *Borland Pascal with Objects, version 7.0. User's Guide*. Scotts Valley: Borland International, Inc., 1992.
- [4] *Borland Pascal with Objects, version 6.0. Programmer's Reference*. Scotts Valley: Borland International, Inc., 1992.
- [5] *Borland Pascal with Objects, version 6.0. Language Guide*. Scotts Valley: Borland International, Inc., 1992.
- [6] *Borland Pascal with Objects, version 6.0. User's Guide*. Scotts Valley: Borland International, Inc., 1992.
- [7] Bielecki J., *Turbo Pascal 5.5 – wersja obiektowa*. WNT, Warszawa, 1990.
- [8] Marciniak A., *Turbo Pascal 5.5*. PWN, Warszawa, 1990.
- [9] Marciniak A., *Nowe elementy systemu i języka Turbo Pascal w wersji 6.0 z opisem biblioteki Turbo Vision*, Wydawnictwo Nakom, Poznań, 1991.
- [10] Marciniak A., *Turbo Pascla 7.0 z elementami programowania, cz.1 i 2.*, Wydawnictwo Nakom, Poznań, 1991
- [11] Zalewski A., *Turbo Vision – programowanie systemów interakcyjnych w Turbo Pascalu*, Wydawnictwo Nakom, Poznań, 1993.

Indeks

bufor nakładkowy	94
constant parameters	84
constructor	30
destructor	47
definicja	47
rola	49
dyrektywa	
public	59
dyrektywa kompilatora	
\$P	83, 88, 91
\$Q	84
\$T	83
\$Y	84
M	94
dziedziczenie	9, 11
funkcja standardowa	
Assigned	85
High	87, 89, 91
Low	87, 89, 91
New	45
SizeOf	54
TypeOf	54
Help	
dołączanie	83
hermetyzacja	9
hierarchia obiektów w Turbo Vision	66
indeks metod dynamicznych	61
inherited	59
interface	18
kolekcja	66
konstruktor	
definicja	30
rola	31
Local menu	83
Messages window	83
metody	40
definicja	15
definiowanie	15
kontrola wywołań	34
statyczne	23, 41
wirtualne	24, 28, 41
zasięg	16
zasłanianie	21
Multiple user-installable Help file support	83
obiekt	9
dynamiczny	43
reprezentacja	31
object	11
ObjectBrowser	82
open-array parameter	88
open-string parameters	91
parametr	
Self	16, 17, 49
parametry stałe	84
pola obiektu	13, 15
pole VMT	31, 32, 35
polimorfizm	9, 24, 27, 49
private	20
procedura standardowa	
Break	85
Continue	86
Dispose	46
Exclude	87
Fail	55
Include	86
NewStr	77
prywatne	
metody	20
pola	20
przekazywanie obiektów	
jako parametru procedur	26
poprzez moduły	18
public	59
rejestracja typu	70, 75
rekord rejestracyjny	75
rozszerzanie obiektów	21
segment danych programu	93
segment kodu programu	93
segment stosowy	93
segment wstępny programu	93
stała	
stCreate	71
stOpen	71
stOpenRead	71
stOpenWrite	71
stos zmiennych dynamicznych	94
strumień	70
symbole	
zachowanie informacji	83
Syntax highlighting	82
Tablica Metod Dynamicznych DMT	63
Tablica Metod Wirtualnych VMT	32, 35, 49, 62
Tools menu	82
TPX.EXE	82
TURBO.EXE	82
typ	
FNameStr	73
PString	77
TBufStream	72
TCollection	67
TDosStream	73
TEmsStream	73
TObject	67

TStream	71
TStreamRec	75
typ obiektowy	
definiowanie	11
definiowanie obiektu potomnego	11
definiowanie zmiennych	12, 13
zgodność	25, 26

VMT	
pole	31, 32, 35
tablica	32, 35, 49





**Wydawnictwa Politechniki Wrocławskiej
są do nabycia w następujących księgarniach:
„Politechnika”**

**Wybrzeże Wyspiańskiego 27, 50-370 Wrocław
budynek A-1 PWr., tel. (0-71) 20 25 34
oraz Oficyny Wydawniczej Politechniki Wrocławskiej
plac Grunwaldzki 13, 50-377 Wrocław
budynek D-1 PWr., tel. (0-71) 20 32 52
Prowadzimy sprzedaż wysyłkową**

ISBN 83-7085-157-6