helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

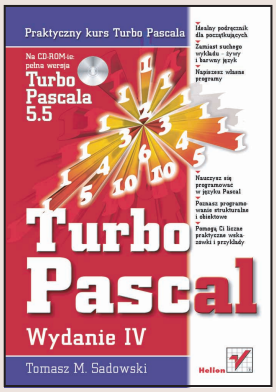
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Praktyczny kurs Turbo Pascala. Wydanie IV



Autor: Tomasz M. Sadowski
ISBN: 83-7361-214-9
Format: B5, stron: 288

Turbo Pascal nie jest wprawdzie powszechnie używany przy pisaniu profesjonalnych aplikacji, stanowi jednak wspaniałą propozycję dla początkujących programistów. Dzięki Pascalowi możesz zacząć szybko pisać własne programy, ucząc się jednocześnie poprawnego programowania strukturalnego i obiektowego.

„Praktyczny kurs Turbo Pascala” to książka przeznaczona dla wszystkich tych, którzy chcieliby wykorzystać praktyczne aspekty programowania w Turbo Pascalu. Prezentowane w niej zagadnienia pozwolą Ci na stopniowe zapoznanie się z zasadami programowania oraz elementami języka na drodze doświadczeń i rozwiązywania problemów napotykanych w trakcie tworzenia aplikacji. Książka napisana jest żywym i przystępnym językiem, a stopień „wtajemniczenia komputerowego” potrzebny do zrozumienia omówionych zagadnień jest minimalny.

Dzięki książce:

- Poznasz środowisko TurboPascala
- Napiszesz swój pierwszy program w tym języku
- Poznasz instrukcje TurboPascala
- Nauczysz się definiować zmienne i typy
- Dowiesz się, jak uruchamiać programy i usuwać błędy
- Stworzysz własne funkcje i procedury
- Poznasz programowanie obiektowe
- Poznasz podstawy algorytmiki i optymalizacji programów
- Zapoznasz się z funkcjami biblioteki standardowej

To już czwarte, uzupełnione i rozszerzone wydanie bestsellera wydawnictwa Helion. Dzięki poprzednim edycjom tej książki tysiące osób rozpoczęło swoją przygodę z programowaniem. Teraz i Ty możesz do nich dołączyć.

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

Spis treści

Do Czytelnika 5
Rozdział 1. Zaczynamy 7
Rozdział 2. Pierwszy program 17
Rozdział 3. Kalkulator 23
Rozdział 4. Jak programista z komputerem 29
Rozdział 5. Instrukcja warunkowa (i nie tylko) 37
Rozdział 6. Zapętlamy 43
Rozdział 7. Pojawia się więcej danych… 49
Rozdział 8. Czarna skrzynka, czyli procedury i funkcje 59
Rozdział 9. Konwersacja z procedurami 67
Rozdział 10. Definiujemy nowe typy 77
Rozdział 11. Stałe 83
Rozdział 12. Porządki w danych 89
Rozdział 13. Co zrobić, żeby nie stracić wyników pracy… 97
Rozdział 14. Pliki tekstowe 105
Rozdział 15. Co to takiego string? 111
Rozdział 16. Wskaźniki 119
Rozdział 17. Moduły standardowe 129
Rozdział 18. Coś dla relaksu — środki uruchomieniowe 137
Rozdział 19. Budujemy bazę danych 143
Rozdział 20. Rekord + wskaźnik = lista 149
Rozdział 21. Grafika 157

Rozdział 22. Grafika w zastosowaniach 163
Rozdział 23. Własne moduły 167
Rozdział 24. Obiekty w pigułce (dozwolone od wersji 5.5) 175
Rozdział 25. Dziedziczenie, polimorfizm i metody wirtualne 185
Rozdział 26. Nie tylko edytor: zaawansowane możliwości IDE 195
Rozdział 27. Błędy — rzecz ludzka 201
Rozdział 28. Jak pisać dobre programy 211
Dodatek A Pytania i odpowiedzi 225
Dodatek B Instalacja i konfiguracja Turbo Pascala 231
Dodatek C Odpowiedzi do zadań 237
Dodatek D Zawartość płyty CD 255
Dodatek E Kilka przydatnych pojęć komputerowych 257
Dodatek F Słowniczek angielsko-polski 267
Literatura 271
Skorowidz 273

Rozdział 4.

Jak programista z komputerem

- Jak wprowadzić dane dla programu
- Zmienne, ich deklarowanie i inicjalizowanie
- Typy proste
- Operatory logiczne

Przyjrzyjmy się ponownie naszemu programikowi-kalkulatorowi. Jego zasadnicza wada polega na tym, że aby uzyskać jakikolwiek wynik, należy za każdym razem wpisywać liczby do nawiasów procedury `writeln` i kompilować program. Istnieje oczywiście sposób, by tego nie robić. Jak w każdym przyzwoitym języku programowania, w Pascalu istnieje pojęcie *zmiennej*, która jest identyfikowanym przez nazwę pojemnikiem na wartość wykorzystywaną w trakcie działania programu. Pewną wadą zmiennych jest konieczność ich *deklarowania*, tj. informowania kompilatora, że dana zmienna jest określonego typu. W Pascalu deklaracja zmiennych ma postać:

```
lista-nazw : typ
```

Nazwy na *liście* oddziela się przecinkami, zaś grupa deklaracji musi rozpoczynać się słowem kluczowym `var` (ang. *variables* — zmienne). Nietrudno się domyślić, że deklaracje zmiennych powinny poprzedzać ich użycie, a zatem należy je umieszczać po nagłówku programu, a przed otwierającym słowem `begin`. Składnia języka zabrania deklarowania zmiennych w treści programu, tak więc próba umieszczenia takiej deklaracji pomiędzy słowami `begin` i `end` zakończy się błędem kompilacji.

Załóżmy, że nasz kalkulatorek ma się zajmować mnożeniem dwóch dowolnych liczb rzeczywistych. Przyjmie on teraz postać:

MNOZ1.PAS

```
program Mnozenie1;
(* Program mnożący dwie dowolne liczby *)

var
  Liczba1, Liczba2 : real;
```

```
begin
  readln(Liczba1);            { odczytaj liczby z klawiatury }
  readln(Liczba2);
  writeln(Liczba1*Liczba2);   { wyświetl ich iloczyn }
end.
```

Po słowie `begin` pojawiła się kolejna nowość: wywołanie standardowej procedury wejścia `readln`. Procedura ta jest „odwrotnością" poznanej poprzednio `writeln`, służy zaś do wprowadzenia wartości zmiennych `Liczba1` i `Liczba2` z klawiatury.

Po uruchomieniu programu możemy wpisać z klawiatury dwie kolejne liczby, naciskając po każdej klawisz *Enter*. Program obliczy i wyświetli ich iloczyn; zauważmy, że wynik jest wyświetlany właśnie we wspomnianej przed chwilą postaci naukowej, tj. *liczbaEwykładnik*. Jest ona bardziej pojemna, tj. łatwiej reprezentuje się w niej bardzo małe i bardzo duże liczby — ale mniej czytelna. Na szczęście już wkrótce poznamy metodę pozwalającą na wyprowadzanie liczb w bardziej zrozumiałej postaci.

Podobnie jak w przypadku `writeln`, istnieje też odmiana procedury `readln` — `read`, która przy kolejnych wywołaniach nie oczekuje znaku nowego wiersza. Jest ona stosowana dość rzadko i wymaga pewnej wprawy, nie będziemy się więc na razie nią zajmować. Ogólna postać wywołania procedury `readln` wygląda tak:

```
readln(lista-argumentów)
```

gdzie *`lista-argumentów`* to dowolnej długości ciąg zmiennych rozdzielonych przecinkami. Jak widać, `readln` umożliwia wprowadzenie za pomocą jednego wywołania kilku wartości, które w takim przypadku należy podczas wpisywania rozdzielić co najmniej jednym tzw. *białym znakiem* (ang. *whitespace*), tj. znakiem spacji, tabulacji lub nowego wiersza. Lista argumentów procedury `readln` może też być pusta, tak więc poprawna jest poniższa instrukcja:

```
readln;
```

Nie robi ona nic poza oczekiwaniem na wprowadzenie znaku nowego wiersza (czyli naciśnięcie klawisza *Enter*). Aby przekonać się o jej użyteczności, dopisz ją na końcu programu *Mnozenie1*.

Pora teraz zająć się bliżej deklaracjami zmiennych. Każda zmienna w Pascalu posiada swoją *nazwę* i *typ*. Nazwa zmiennej służy do identyfikowania jej w programie i zależy tylko od programisty (podlega ona tym samym ograniczeniom, co każdy identyfikator — może zawierać litery, cyfry i podkreślenia, ale nie może zaczynać się od cyfry). Jak już powiedziano, duże litery nie są odróżniane od małych, ale zróżnicowanie liter pomaga przy czytaniu nazw (`WiekPracownika` czyta się łatwiej niż `wiekpracownika`).

Użycie dużych liter i znaków podkreślenia poprawia czytelność identyfikatorów.

Typ zmiennej określa z kolei jej postać zewnętrzną, czyli zakres wartości i zestaw operacji, które można na niej wykonać, a także *reprezentację wewnętrzną*, czyli sposób jej przechowywania i traktowania przez inne obiekty programu. Swego rodzaju „atomami" wśród typów są tzw. *typy proste*, tj. takie, których struktura nie składa się z innych typów. Do najczęściej stosowanych typów prostych Turbo Pascala należą:

integer	liczba całkowita ze znakiem
real	liczba rzeczywista
char	pojedynczy znak kodu ASCII
Boolean	wartość logiczna: prawda lub fałsz

Pierwsze dwa z wymienionej listy typów z grubsza odpowiadają znanym w matematyce liczbom całkowitym i rzeczywistym (stanowią ich ograniczone podzbiory). Typ Boolean służy do deklarowania zmiennych logicznych (*boolowskich*), przyjmujących jedynie dwie wartości: true (prawda) i false (fałsz). Jest on szeroko stosowany w sterowaniu wykonaniem programu (w instrukcjach warunkowych i pętlach). Warto zauważyć, że wszelkie operacje porównania (czyli wyrażenia logiczne), np.

```
LiczbaPacjentow > 100
```

dają w wyniku wartość boolowską (są prawdziwe lub fałszywe). Do obsługi bardziej skomplikowanych wyrażeń logicznych służą w Turbo Pascalu następujące operatory logiczne (wymienione w kolejności malejącego priorytetu):

not	zaprzeczenie, negacja logiczna (daje true, gdy argument ma wartość false)
and	iloczyn logiczny, koniunkcja (daje true, gdy wszystkie argumenty mają wartość true)
or	suma logiczna, alternatywa (daje true, gdy co najmniej jeden argument ma wartość true)
xor	suma modulo 2, alternatywa wyłączająca (daje true, gdy nieparzysta liczba argumentów ma wartość true)

Przykładami złożonych wyrażeń logicznych są np.:

```
(Cena > 1000) and (Zapas > 500)
(Srednia > 4.0) or (Srednia > 3.0) and (Dochody < 800)
```

Zwróćmy uwagę, że porównania są ujęte w nawiasy. Wynika to stąd, że operatory not, and, or i xor mają wyższy priorytet od operatorów relacji, my zaś chcemy, by porównania zostały wykonane w pierwszej kolejności. Pominięcie nawiasów może więc przysporzyć pewnych kłopotów (zobacz też zadanie 3. na końcu rozdziału).

Warto tu zauważyć, że operatory not, and, or i xor mają dwoisty charakter: oprócz działania logicznego, mogą one również operować na bitach liczb całkowitych (są wówczas tzw. *operatorami bitowymi*). Operator bitowy interpretuje swoje argumenty nie jako „prawdziwe” liczby, lecz jako ciągi bitów (zer i jedynek), dokonując stosownych operacji na odpowiadających sobie pozycjach (parach bitów). Jednoargumentowy operator not neguje bity swojego argumentu. Pozostałe operatory są dwuargumentowe: operator and ustawia dany bit na jeden tylko wówczas, gdy odpowiednie bity obu argumentów są równe jeden, zaś or — gdy co najmniej jeden z bitów jest równy jeden. Wreszcie operator xor ustawia bit na jeden, jeśli odpowiednie bity argumentów mają przeciwne wartości (0 i 1 lub 1 i 0).

Aby zrozumieć działanie operatorów bitowych, przeanalizuj poniższy przykład. Argumentami w przykładowych operacjach bitowych będą ośmiobitowe liczby bez znaku: 15 (00001111) i 85 (01010101).

	01010101	01010101	01010101
not 00001111	and 00001111	or 00001111	xor 00001111
11110000 (240)	00000101 (5)	01011111 (95)	01011010 (90)

Omówione w tym rozdziale typy nie są oczywiście wszystkimi dostępnymi (zobacz *Aneks*), jednakże stanowią grupę o podstawowym znaczeniu.

Tak więc zawarta w programie *Mnozenie1* deklaracja

```
var
  Liczba1, Liczba2 : real;
```

poleca kompilatorowi zarezerwowanie miejsca na dwie zmienne typu rzeczywistego, które w programie będą widoczne pod nazwami `Liczba1` i `Liczba2`. Na tym jednak nie koniec problemu: samo zadeklarowanie zmiennej nie nadaje jej żadnej wartości, a jedynie przydziela miejsce w pamięci (zwykle wypełnione dość przypadkową zawartością). Pomijając bezużyteczność zmiennej nie przechowującej żadnej konkretnej wartości, próba odwołania się do niej zwykle nie wychodzi nikomu na dobre. Aby więc zmienna nadawała się do użytku, musi zostać:

- *zadeklarowana*, co poinformuje kompilator, jak ma się z nią obchodzić (jaka jest jej reprezentacja wewnętrzna),
- *zainicjalizowana*, czyli wypełniona sensowną zawartością.

Pominięcie inicjalizacji zmiennej nie jest wykrywane przez kompilator (program jest formalnie poprawny), potrafi jednak spowodować złośliwe i trudne do wykrycia błędy wykonania!

Zauważmy, że zmiana w powyższej deklaracji typu `real` na `integer` nie będzie błędem, ale ograniczy swobodę działania kalkulatora do liczb całkowitych (czyli np. ze względu na ograniczenie zakresu nie będzie już możliwe pomnożenie 123456 przez 654321). Liczby całkowite mają natomiast tę przewagę nad rzeczywistymi, że są reprezentowane z absolutną dokładnością. Ponadto, w ich zbiorze ma sens liczba kolejna po danej i poprzednia przed daną, a co za tym idzie, można stosować je w charakterze liczników i indeksów.

Aby zaś było wiadomo, czego właściwie życzy sobie program po uruchomieniu (zauważmy, że ostatnia wersja kalkulatorka wprawdzie już coś potrafi, ale nie jest zbyt rozmowna), uzupełnijmy go o jeszcze dwie instrukcje:

MNOZ2.PAS

```
program Mnozenie2;
(* Bardziej rozmowny kalkulator *)

var
  Liczba1, Liczba2 : real;
```

```
begin
  write('Podaj pierwszą liczbę: ');    { dodano }
  readln(Liczba1);
  write('Podaj drugą liczbę: ');       { dodano }
  readln(Liczba2);
  writeln('Iloczyn podanych liczb wynosi ', Liczba1*Liczba2);
  readln
end.
```

Spróbuj wykonać ten program dla różnych wartości liczb (czy da się pomnożyć np. `1e20` przez `1e20`?). Spróbuj również zmienić wykonywaną operację na dzielenie i podzielić coś przez zero… Jak uniknąć tego typu wpadek? Cierpliwości!

Podsumowanie

- W Pascalu możemy posługiwać się zmienną jako pojemnikiem na dane.
- Każda zmienna musi być określonego typu.
- Typy zmiennych określa się w ich deklaracjach, po słowie kluczowym `var`.
- Deklaracja zmiennej nie powoduje nadania jej sensownej wartości (nie jest połączona z inicjalizacją).
- Do wprowadzania wartości zmiennych z zewnątrz do programu służą procedury `read` i `readln`.
- Procedury te mogą (i powinny) być uzupełniane objaśniającymi procedurami `write` i `writeln`.
- Do przechowywania liczb w programie służą zmienne typu `integer` (liczby całkowite) i `real` (liczby rzeczywiste).
- Wartości logiczne przechowuje się w zmiennych typu `Boolean`.
- Do operowania na wartościach logicznych służą operatory logiczne: `not`, `and`, `or` i `xor`.
- Operatory te służą również do manipulacji na bitach liczb całkowitych.

Zadania

1. Napisz program obliczający długość przeciwprostokątnej w trójkącie prostokątnym z twierdzenia Pitagorasa. Pierwiastek z liczby oblicza funkcja `sqrt` (jak liczbę podnieść do kwadratu — pomyśl!).
2. Zmodyfikuj program *Mnozenie1* tak, by wprowadzanie wartości czynników odbywało się za pomocą jednego wywołania procedury `readln`. Co się stanie, jeśli w miejsce `readln` zastosujesz procedurę `read`?

3. Jaki będzie wynik (i dlaczego) obliczenia następujących wyrażeń:

a) `16+7` **b)** `16 > 7` **c)** `3+4 and 1` **d)** `(3+4) and 1`

e) `1 > 2 or 3` **f)** `(1 > 2) or 3` **g)** `1.7 or 13.0` **h)** `15 xor 7`

Aneks (dla amatorów typów)

- Pozostałe typy całkowitoliczbowe…
- …i rzeczywiste
- Zastosowanie koprocesora arytmetycznego

Przedstawione przed chwilą cztery typy stanowią jedynie niewielką część wachlarza dostępnego w Turbo Pascalu. I tak w zakresie typów całkowitoliczbowych (pokrewnych `integer`) mamy do wyboru następujące typy proste:

Nazwa	Zakres	Rozmiar	Liczba całkowita…
`integer`	–32768..32767	2 bajty	ze znakiem
`shortint`	–128..127	1 bajt	krótka ze znakiem
`longint`	–2147483648..2147483647	4 bajty	długa ze znakiem
`byte`	0..255	1 bajt	krótka bez znaku (bajt)
`word`	0..65535	2 bajty	bez znaku (słowo)

Każdy z nich może być stosowany do określonych celów, np. typ `byte` nadaje się do reprezentacji zawartości pamięci (w bajtach), a `longint` — do prowadzenia statystyk astronomicznych. Rozważny dobór typów pozwala na zaoszczędzenie pamięci przeznaczonej na dane bez utraty jakości obliczeń (chociaż objawia się to dopiero przy stosowaniu większych struktur danych).

Wszystkie typy całkowitoliczbowe, typ `char` (przyjmujący *de facto* wartości od 0 do 255, tyle że reprezentowane w postaci znakowej) i `Boolean`, a także nie omawiane tutaj typy wyliczeniowe, stanowią grupę tzw. typów porządkowych (ang. *ordinal types*). Nazwa ta bierze się stąd, iż w reprezentowanych przez nie zbiorach wartości zachowana jest tzw. *relacja porządku*, określająca kolejność poszczególnych elementów w zbiorze wartości.

Jeśli idzie o liczby typu rzeczywistego, to wszystkie typy proste poza `real` są związane z zastosowaniem tzw. *koprocesora arytmetycznego* (układu współpracującego z mikroprocesorem, wyspecjalizowanego w wykonywaniu obliczeń na liczbach rzeczywistych, a przez to znacznie je przyspieszającym). Typy te przestawiają się następująco:

Nazwa	Zakres	Rozmiar	Dokładność (cyfr)
`real`	$2{,}9 * 10^{-39} – 1{,}7 * 10^{38}$	6 bajtów	11 – 12
`single`	$1{,}5 * 10^{-45} – 4{,}0 * 10^{38}$	4 bajty	7 – 8
`double`	$5{,}0 * 10^{-324} – 1{,}7 * 10^{308}$	8 bajtów	15 – 16
`extended`	$3{,}4 * 10^{-4932} – 1{,}1 * 10^{4932}$	10 bajtów	19 – 20
`comp`	$9{,}2 * 10^{-18} – 9{,}2 * 10^{18}$	8 bajtów	19 – 20

Ostatni typ, mimo że jest tylko bardzo długą (64 bity) liczbą całkowitą, znalazł się w tym gronie, ponieważ jest typem specyficznym dla koprocesora arytmetycznego. Wykorzystując odpowiedni typ do reprezentacji liczb rzeczywistych, można sobie pozwolić np. na zwiększenie dokładności obliczeń poprzez zastosowanie typu `double` w miejsce `real` (przy okazji wzrasta zużycie pamięci) lub na upakowanie większej ilości danych w pamięci poprzez zastosowanie krótszego typu `single` (przy okazji spada dokładność obliczeń).

W pierwszych modelach komputerów PC koprocesor był oddzielnym układem montowanym na płycie głównej. Począwszy od procesora 80486, koprocesor arytmetyczny jest zintegrowany z procesorem w obrębie pojedynczego układu. W dominujących obecnie na rynku komputerach opartych na procesorach z rodziny Pentium oraz AMD K7 koprocesor jest zawsze dostępny. Użytkownicy leciwych już komputerów z procesorami 80486SX i 80386 (układy 80286 można obecnie znaleźć głównie w muzeach) mogą wykorzystać tzw. *emulację*, polegającą na zastąpieniu sprzętowej realizacji operacji arytmetycznych przez wywołania procedur obliczeniowych z odpowiedniej biblioteki. Złożoność takich operacji powoduje wydłużenie czasu obliczeń; dołączenie biblioteki zwiększa również o kilkanaście kilobajtów objętość samego kodu wynikowego.

Aby umożliwić programowi korzystanie z koprocesora, należy poinformować kompilator o jego obecności poprzez włączenie opcji *Options-Compiler-Numeric Processing 8087/80287* lub umieszczenie w programie tzw. *dyrektywy kompilatora* `{$n+}`. W razie konieczności włączenia emulacji należy użyć polecenia *Options-Compiler-Emulation On* (Turbo Pascal 5.0 i 5.5) lub *Options-Compiler-Numeric Processing Emulation* (Turbo Pascal 6.0 i 7.0), ewentualnie dyrektywy `{$e+}`[1].

Niestety, na tym nie koniec problemów. Włączenie koprocesora bez zmiany typu `real` na `single`, `double` lub `extended` niewiele daje. Dzieje się tak dlatego, że przed wykonaniem obliczenia liczba typu `real` musi zostać przekształcona na wewnętrzną reprezentację „koprocesorową” (dokładniej — na wartość typu `extended`), co pochłania część zaoszczędzonego czasu. Ponadto, konwersja wcale nie powoduje wzrostu dokładności obliczeń (a wręcz może spowodować jej spadek), gdyż uzupełnienie liczby `real` do `extended` odbywa się przez dodanie przypadkowych cyfr na jej najmniej znaczących pozycjach.

[1] Zwolennikom radykalnych rozwiązań warto polecić modernizację sprzętu.

Aby skutecznie wykorzystać koprocesor arytmetyczny, należy zamiast typu `real` stosować w programach typy `single`, `double` lub `extended`. Jeśli możemy z dużą pewnością założyć, że nasz kod będzie uruchamiany na nowszym komputerze, najlepiej od razu deklarować zmienne rzeczywiste jako `single` lub `double`. Alternatywne rozwiązanie — przedefiniowanie typu `real` — pokażemy w rozdziale 10. Aby nie komplikować programów przykładowych, w dalszej części książki pozostaniemy przy typie `real`, jednak teksty źródłowe zamieszczone na płycie CD będą zawierały odpowiednie instrukcje, zastępujące go typami `single` lub `double` (zależnie od wymagań danego programu).

Warto na koniec zwrócić uwagę na naukowy format zapisu liczby typu `double` lub `extended` (tj. liczbę cyfr po kropce i w wykładniku). Porównanie go z zapisem liczby typu `real` daje namacalny dowód różnicy w dokładności reprezentacji (co np. przy odwracaniu macierzy jest sprawą niebagatelną!).