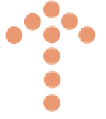
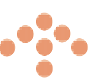
Turbo Pascal - Programowanie
Autor: Tomasz M. Sadowski
ISBN: 83-86718-40-4
Format: B5, 136 strony
Data wydania: 08/1996

WWW.PROGRAMOWANIE.TK



Jak wiesz, sam komputer jest jedynie zbiorem dość skomplikowanych układów elektronicznych, które same z siebie nie są w stanie wykonać jakiejkolwiek znaczącej operacji. Jego "inteligencja" bierze się w całości z wiedzy, którą przekazali mu programiści w postaci odpowiedniego oprogramowania. Niniejsza książka wprowadzi Cię w fascynujący świat programowania w Turbo Pascalu.



- Co to jest Problem, Metoda, Algorytm, Implementacja, Zdrowy rozsądek...?
- Jak korzystać z edytora Turbo Pascala?
- Od czego zacząć, czyli jak obliczyć pole koła za pomocą komputera?
- Czym się różni łańcuch od liczby rzeczywistej?
- Do czego służy instrukcja warunkowa?
- Jak rozwiązać dowolne równanie?
- Co się kryje pod tajemniczymi nazwami: funkcje i procedury?
- Do czego służą tablice, a do czego pętle?
- Jak zapisywać dane w pliku na dysku?
- Jak tworzyć i korzystać z modułów bibliotecznych?
- Jak uruchamiać "oporne" programy?

Odpowiedzi na te i inne pytania znajdziesz w tej książce! Wszystko opisane żywym i barwnym językiem, zilustrowane krótkimi, przejrzystymi i praktycznymi programami.

Szkoda czasu, żeby przegryzać się przez opasłe tomiska, aby nauczyć się podstaw programowania. Rozpocznij od tej krótkiej, ale niezwykle treściwej książki, dzięki której programowanie stanie się szybkie, łatwe i przyjemne...

Jak wiesz, komputer jest jedynie zbiorem skomplikowanych układów elektronicznych, które same z siebie nie są w stanie wykonać jakiejkolwiek znaczącej operacji. Jego „inteligencja” bierze się z wiedzy, którą przekazali mu programiści w postaci odpowiedniego oprogramowania. Niniejsza książka wprowadzi Cię w fascynujący świat programowania w Turbo Pascalu.

- Co to jest Problem, Metoda, Algorytm, Implementacja, Zdrowy rozsądek...?
- Jak korzystać z edytora Turbo Pascala?
- Od czego zacząć czyli jak obliczyć pole koła za pomocą komputera?
- Czym różni się łańcuch od liczby rzeczywistej?
- Do czego służy instrukcja warunkowa?
- Jak rozwiązać dowolne równanie?
- Co kryje się pod tajemniczymi nazwami: „funkcje i procedury?
- Do czego służą tablice, a do czego pętle?
- Jak zapisywać dane w pliku na dysku?
- Jak tworzyć i korzystać z modułów biblitecznych?
- Jak uruchamiać „oporne” programy?

Odpowiedzi na te i inne pytania znajdziesz w tej książce! Wszystko opisane żywym i barwnym językiem, zilustrowane krótkimi, przejrzystymi i praktycznymi programami.

Szkoda czasu żeby przegryzać się przez opasłe tomiska aby nauczyć się podstaw programowania. Rozpocznij od tej krótkiej ale niezwykle treściwej książki, dzięki której programowanie stanie się szybkie, łatwe i przyjemne...

Projekt okładki: Maciej Pasek „ARTGRAF”

© HELION, 1996

ISBN: 83-86718-40-4

Wszelkie prawa zastrzeżone.

Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Występujące w tekście znaki są zastrzeżonymi znakami firmowymi bądź towarowymi ich posiadaczy.

Autor oraz Wydawnictwo Helion dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich.

Printed in Poland

Spis treści

Po co mi programowanie? 5
Co to jest programowanie? 8
Jak to się robi naprawdę? 10
Problem 10
Metoda 11
Algorytm 11
Projekt 14
Implementacja 15
Uruchomienie 15
Zdrowy rozsądek 16
Narzędzie 17
Zaawansowane funkcje edytora 22
Zapamiętaj 25
Pierwszy program 26
Zapamiętaj 31
Nic nie jest doskonałe... 32
Zapamiętaj 36
Ulepszamy dalej 37
Zapamiętaj 40
Wyrażenia 41
Zapamiętaj 47
Instrukcja warunkowa 48
Zapamiętaj 51
Rozwiązujemy dowolne równanie 52
Zapamiętaj 57
Funkcje i procedury 59

Zapamiętaj ... 64
Jak program porozumiewa się z funkcją? ... 66
Zapamiętaj ... 73
Programowanie na poważnie ... 75
Typy strukturalne, czyli jak przechować więcej danych ... 77
Zapamiętaj ... 84
Struktura programu ... 85
Typy i stałe ... 87
Zapamiętaj ... 90
Do dzieła! ... 91
Pliki, czyli jak uchronić dane przed zgubą ... 101
Zapamiętaj ... 109
Łańcuchy ... 110
Zapamiętaj ... 113
Więcej pamięci! ... 114
Zapamiętaj ... 120
Pożyteczne drobiazgi, czyli moduły biblioteczne ... 121
Zapamiętaj ... 125
Moduły własne ... 126
Zapamiętaj ... 130
Jak uruchamiać oporne programy ... 131
Literatura ... 134

Po co mi programowanie?

Zanim zagłębimy się w dyskusję o tajnikach programowania, wypadałoby odpowiedzieć na podstawowe pytanie: *czy w ogóle jest Ci ono potrzebne?*

W zasadzie — nie. Brak umiejętności programowania nie przeszkodzi Ci w zdaniu matury, napisaniu listu, zrobieniu zakupów czy prowadzeniu samochodu. No, może od czasu do czasu będziesz musiał zaprogramować magnetowid, ale czynność ta z „prawdziwym" programowaniem ma raczej mało wspólnego... Tym niemniej — skoro sięgnąłeś po tę książkę — można założyć, że masz zamiar zawrzeć bliższą znajomość z komputerami. A to już zupełnie inna sprawa.

Jak wiesz sam komputer jest jedynie zbiorem dość skomplikowanych układów elektronicznych, które same z siebie nie są w stanie wykonać jakiejkolwiek znaczącej operacji. Jego „inteligencja" bierze się w całości z wiedzy, którą przekazali mu programiści w postaci odpowiedniego oprogramowania. Nawet tak trywialna operacja, jak wprowadzenie pojedynczego znaku z klawiatury, związana jest z wykonaniem sporej liczby operacji zakodowanych w oprogramowaniu systemowym. A co dopiero wyliczenie bilansu firmy czy wycieniowanie skomplikowanego rysunku...

A zatem nie ma mowy o funkcjonowaniu komputera bez oprogramowania. Oczywiście nie oznacza to, że aby wprowadzić z klawiatury swoje nazwisko, musisz cokolwiek programować: odpowiednie funkcje zostały już dawno utworzone i przekazane do dyspozycji użytkowników pod postacią systemu operacyjnego i oprogramowania użytkowego. O ile nie masz zamiaru zająć się tworzeniem podobnego oprogramowania, nie musisz nawet wiedzieć, jak takie funkcje działają, chociaż czasem przydaje się to w różnych kryzysowych sytuacjach. Tak więc znajomość programowania nie jest konieczna do obsługi komputera.

Może w takim razie programowanie jest potrzebne do korzystania z programów użytkowych? Okazuje się, że też nie. Nowoczesne edytory tekstów, arkusze kalkulacyjne i inne powszechnie używane aplikacje biurowe pozwalają na wykonywanie wymyślnych operacji przez naciśnięcie kilku klawiszy lub kliknięcie myszką. Typowa

księgowa (używająca Excela), pisarz (piszący w Wordzie) czy grafik (korzystający z Corela) nie mają zwykle pojęcia o programowaniu — bo do niczego nie jest im ono potrzebne.

No to o czym my w ogóle mówimy? — mógłbyś zapytać. Otóż sprawa wcale nie jest przesądzona. Przede wszystkim istnieje spora grupa czytelników, którzy zamierzają poznać sposoby przymuszenia komputera do wykonywania dokładnie tego, czego sobie zażyczą — czyli właśnie metody programowania. Część z nich zapewne zostanie w przyszłości programistami i zajmie się tworzeniem oprogramowania systemowego i użytkowego (działalność to wcale intratna). Inni, podchodzący do zagadnienia bardziej po amatorsku, zadowolą się satysfakcją intelektualną płynącą ze zrozumienia zasad działania programów i umiejętności zapanowania nad skomplikowanym narzędziem, jakim jest nowoczesny komputer. Jeszcze inni (tych będzie zapewne najwięcej) będą po prostu wykorzystywać zdobyte umiejętności do rozwiązywania problemów spotykanych w codziennej pracy, nie pretendując bynajmniej do miana programistów, choćby nawet amatorów.

Ale przecież równie dobrze mogą skorzystać z jakiegoś programu użytkowego. Zgoda, z pewnym drobnym zastrzeżeniem. Nowoczesne oprogramowanie jest w stanie zrobić prawie wszystko za jednym naciśnięciem klawisza czy kliknięciem myszką. Nie znaczy to jednak, że spełni ono wszystkie nasze żądania, ani też że spełni je dokładnie tak, jak byśmy chcieli. Poza tym duże, wszystko-wiedzące-i-mogące pakiety użytkowe są na ogół tak skomplikowane, że obmyślenie metody realizacji zadania często zajmuje więcej czasu, niż zaprogramowanie go od podstaw. Wreszcie może się po prostu zdarzyć (i najczęściej się zdarza), że odpowiedniego oprogramowania nie będzie pod ręką, albo też, że co prawda będzie, ale ze względu na wysokie wymagania sprzętowe (dotyczy to zwłaszcza nowej generacji programów dla Windows) jego uruchomienie będzie mniej lub bardziej kłopotliwe.

Czy do banalnego obliczenia średniej ocen potrzebny Ci będzie od razu zaawansowany pakiet matematyczny, jak *Statgraphics*, czy arkusz kalkulacyjny, jak *Excel*? Czy Twój ulubiony edytor potrafi zliczyć wszystkie znaki w tekście, który właśnie napisałeś? A jeśli tak, to czy jest w stanie określić średnią długość zdania? Wszystkie te operacje można zrealizować za pomocą prostych programików w Pascalu lub Basicu, których napisanie i uruchomienie nawet niezbyt wprawnemu użytkownikowi zajmuje kilkanaście minut.

Nawet jeśli jesteś wyłącznie użytkownikiem jakiegoś konkretnego programu (np. edytora tekstów) i nie masz zamiaru zawierać znajomości z Pascalem czy Basicem, zapoznanie się z podstawami sztuki programowania będzie nie od rzeczy. Dlaczego? Wyobraź sobie, że w dokumencie, nad którym pracujesz, znajduje się kilkanaście tabelek, a w każdej z nich musisz zamienić miejscami pierwszy wiersz z drugim. Praktycznie żaden edytor nie umożliwia zamiany dwóch wierszy tabeli w jednym kroku, a tym bardziej zrobienia tego dla wszystkich tabelek w dokumencie. Tak jednak się składa, że większość nowoczesnych edytorów (WordPerfect, Word dla Windows) umożliwia użytkownikowi tworzenie tak zwanych *makrodefinicji*, będących niczym innym, jak tylko *programami* napisanymi w specjalnym języku. Zamiast więc żmudnie wyszukiwać kolejne tabele i zamieniać wiersze miejscami, możesz napisać odpowiedni

program (makrodefinicję), która zrobi to automatycznie. Rzecz jasna, nie będziemy się tutaj zajmować językami makrodefinicji, będziesz jednak miał okazję zapoznać się z elementarnymi koncepcjami programowania — jak instrukcja warunkowa, pętla czy procedura — których znajomość w znaczący sposób ułatwi posługiwanie się takimi językami.

Po co więc potrzebne jest programowanie? Najogólniej mówiąc, po to, by móc zrealizować zadania kłopotliwe lub niemożliwe do wykonania „ręcznie". Program, niezależnie od tego, jakim narzędziem utworzony i jakiemu celowi służący, umożliwia automatyzację powtarzających się, rutynowych czynności, których wykonywanie byłoby zbyt nużące lub pochłaniało za dużo czasu. Oczywiście, wszystko można zrobić ręcznie — po co jednak przerzucać dwie tony piasku łopatą, skoro obok stoi gotowy do użycia spychacz... Spróbujmy nauczyć się go obsługiwać.

Co to jest programowanie?

Powiemy obecnie kilka słów na temat samego procesu programowania. Aby Cię za bardzo nie znudzić, ograniczymy się tutaj do krótkiego omówienia jego najważniejszych elementów, rezygnując z formalizmów i zbędnej teorii. Jeśli chcesz dowiedzieć się na ten temat czegoś więcej, możesz (ale nie musisz) przeczytać kolejny rozdział, zatytułowany *Jak to się robi naprawdę?* Jeśli nie masz ochoty, możesz wrócić do niego później.

Na rozwiązanie dowolnego zadania poprzez programowanie składa się kilka etapów. Przede wszystkim musisz wiedzieć, co właściwie chcesz zrobić, czyli mieć konkretny *problem* do rozwiązania. Rzecz jasna, problem musi nadawać się do rozwiazania za pomocą komputera (skąd masz o tym wiedzieć? Na obecnym etapie najlepiej zastosować zdrowy rozsądek). Analizując problem musisz sprecyzować, jakie informacje masz do dyspozycji i co chcesz uzyskać w wyniku jego rozwiązania. Następnie musisz zdecydować się, jak rozwiązać zadanie, czyli wybrać odpowiednią *metodę*. Kolejnym krokiem jest przyjęcie określonego sposobu postępowania, czyli schematu czynności (kroków) prowadzących do rozwiązania. Schemat taki określany jest mianem *algorytmu*. Dodatkową czynnością, którą należy wykonać na tym etapie, jest dobór *struktur danych*, czyli sposobów, w jakie przetwarzana przez nasz program informacja będzie reprezentowana wewnątrz komputera. Czynność ta jest na ogół wykonywana mechanicznie, istnieją jednak problemy, dla których możliwe jest przyjęcie kilku wariantów reprezentowania informacji, zróżnicowanych w zależności od konkretnej sytuacji.

Rzeczywiste problemy są w większości zbyt skomplikowane, by dały się skutecznie rozwiązać za pomocą pojedynczego algorytmu. Skuteczne zrealizowanie algorytmu wymaga ponadto wykonania pewnych czynności pomocniczych, jak choćby wprowadzenia czy wyprowadzenia danych. Tak więc zwykle musisz podzielić zadanie na pewne fragmenty, dające się rozwiązać za pomocą odpowiednich algorytmów, a całość obudować odpowiednim zestawem operacji „administracyjnych”. W ten sposób tworzysz *projekt* programu.

Dopiero teraz wchodzi do akcji komputer. Wykorzystując odpowiedni program użytkowy zapisujemy nasz projekt w postaci wykonywalnej przez komputer. Operacja ta nosi miano *kodowania* lub *implementacji*, a jej wynikiem powinien być działający program. Ponieważ jednak nie jest powiedziane, że program musi być od razu bezbłędny, ostatnim etapem naszej pracy jest jego testowanie, czyli inaczej *uruchamianie*. Na dobrą sprawę i to nie jest koniec: „poważne" programy muszą być przez cały okres swojej eksploatacji *konserwowane*. Określenie to oznacza sprawowanie przez autora nadzoru nad działaniem programu i korektę ewentualnych błędów. Z konserwacją programów wiąże się wreszcie zagadnienie opracowania ich dokumentacji. Dwie ostatnie czynności związane są głównie z programowaniem profesjonalnym, toteż nie będziemy się nimi szczegółowo zajmować.

Tak mniej więcej wygląda cała procedura programowania i — jak nietrudno zauważyć — samo pisanie i uruchamianie programu wcale nie jest w niej najważniejsze. Ponieważ jednak naszym celem jest pokazanie Ci podstaw rzemiosła, nie zaś dyskusja o algorytmach i teorii informatyki (na ten temat napisano wiele mądrzejszych książek), w kolejnych rozdziałach skupimy się głównie na wykładzie języka programowania i omówieniu jego konstrukcji od strony praktycznej. Jeśli chcesz rozszerzyć swoje wiadomości dotyczące „podłoża" programowania, zachęcam Cię do przeczytania następnego rozdziału. Jeśli nie masz ochoty, przejdź do rozdziału zatytułowanego *Narzędzie*, w którym poznasz głównego bohatera tej książki.

Jak to się robi naprawdę?

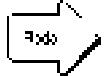
Rozdział ten stanowi rozszerzenie poprzedniego i wcale nie musisz go czytać, chociaż może Ci się to przydać. Dowiesz się z niego, jak wyglądają poszczególne kroki procesu programowania i co oznaczają takie pojęcia jak algorytm, schemat blokowy czy projekt. Poznasz również kilka metod i wskazówek pozwalających na usystematyzowanie i uproszczenie pracy.

Problem

Programowanie nie istnieje samo dla siebie: aby w ogóle miało ono sens, musi służyć konkretnemu celowi. Przyczyną utworzenia programu musi być problem, który masz rozwiązać. Oczywiście, nie każdy problem da się rozwiązać w ten sposób, nie każdy też wymaga zastosowania komputera. Co więc kwalifikuje problem do zaprogramowania?

- Zastosowanie komputera do rozwiązania problemu powinno być uzasadnione i opłacalne. Pisanie na użytek domowy programu pozwalającego na zapamiętywanie adresów znajomych jest stratą czasu — prościej to zrobić za pomocą notesu. Inaczej wygląda sprawa w banku, w którym obsługuje się setki klientów, a od czasu do czasu trzeba wysłać do wszystkich np. zawiadomienie o stanie rachunku.
- Szczególnie podatne na zaprogramowanie są zadania wykonywane wielokrotnie lub złożone z ciągu powtarzających się czynności. Pisanie „do jednorazowego użytku” programu wykonującego zestaw skomplikowanych obliczeń na ogół mija się z celem, gdyż można je wykonać na kalkulatorze. Co innego, gdy przewidujesz możliwość wielokrotnego wykorzystania takiego programu.
- Problem powinien być *dobrze określony*. Oznacza to, że musisz dokładnie wiedzieć, jakie informacje masz do dyspozycji i co chcesz uzyskać, a także potrafić zapisać to w formie dającej się „wytłumaczyć” komputerowi. Przykładem problemu źle określonego może być „rozwiązanie zadania z fizyki”, gdyż nie wiadomo ani jakimi informacjami dysponujemy, ani co chcemy właściwie

obliczyć. Prawidłowe określenie zadania miałoby postać np. „wyznaczenie drogi przebytej przez ciało w spadku swobodnym w pierwszych 10 sekundach ruchu". Przy okazji mamy tu nawiązanie do poprzedniego punktu: te same obliczenia trzeba wykonać dziesięciokrotnie (dla 1, 2, 3,..., 10 sekund).



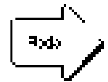
Problem przeznaczony do rozwiązania powinien być jasno zdefiniowany i wystarczająco złożony, by trud włożony w programowanie mógł się opłacić.

Metoda

Po ustaleniu co chcesz osiągnąć i jakimi informacjami dysponujesz, musisz ustalić sposób rozwiązania problemu. Niekiedy (jak w przypadku spadku swobodnego) sytuacja jest oczywista: należy zastosować wzór, który ktoś kiedyś wymyślił. Kiedy indziej trzeba wymyślić rozwiązanie samemu. Warto również zastanowić się nad przyjęciem założeń upraszczających, które mogą zmniejszyć nakład pracy potrzebny na stworzenie programu z niewielkim uszczerbkiem na dokładności.

Przykładem niech będzie wspomniana w poprzednim rozdziale kwestia wyznaczenia średniej długości zdania w tekście. Problem nie jest wcale wzięty „z sufitu": zarówno zbyt krótkie, jak i zbyt długie zdania zmniejszają czytelność tekstu, aby więc nie znudzić czytelnika, musisz trzymać się pewnego optimum, wynoszącego np. 60 znaków. Oto kilka konkurencyjnych rozwiązań tego problemu:

- ustalamy liczbę znaków zawartych w każdym zdaniu w tekście, po czym uzyskane wartości uśredniamy. Metoda ta jest dość oczywista, jednak wymaga określenia ścisłej definicji zdania, co może być kłopotliwe;
- przyjmujemy upraszczające założenie, że każde zdanie kończy się kropką. W tym momencie liczba zdań w tekście równa będzie liczbie kropek (co dla tekstów literackich jest w zasadzie prawdziwe). Zatem wystarczy zliczyć wszystkie znaki w tekście, ustalić, ile jest wśród nich kropek, podzielić pierwszą wartość przez drugą i wynik gotowy.



Najlepszą metodą na znalezienie metody jest sprawdzenie, czy ktoś jej już nie wymyślił. Po ustaleniu metody warto też zastanowić się nad jej uproszczeniem.

Algorytm

Skoro już mamy metodę, musimy zapisać ją w sposób formalny i szczegółowy. Po co? Komputer nie jest niestety na tyle inteligentny, by zrozumieć polecenie „zlicz wszystkie znaki w tekście"[1], toteż musimy nasze zadanie przedstawić w sposób bardziej elementarny. Jak? Właśnie za pomocą algorytmu.

[1] Co prawda niektóre programy, jak np. Word dla Windows, udostępniają polecenie zliczania znaków w tekście, jednak jest ono właściwością programu, nie zaś komputera, i musiało zostać wcześniej zaprogramowane przez twórców edytora.

Zgodnie z definicją, algorytm to „zbiór określonych reguł postępowania, które realizowane zgodnie z ustalonym porządkiem umożliwiają rozwiązanie określonego zadania". Przekładając to na bardziej ludzki język możemy powiedzieć, że algorytm jest zapisem czynności, które należy krok po kroku wykonać w celu uzyskania wyniku. Zapisując algorytm musisz więc ustalić, jakie elementarne operacje będzie trzeba wykonać w trakcie realizacji zadania i jaka powinna być ich kolejność. Dodatkowo musisz określić tak zwane *kryterium stopu*, czyli warunek, którego spełnienie powoduje zakończenie wykonywania operacji (nie chcesz chyba czekać na wynik w nieskończoność).

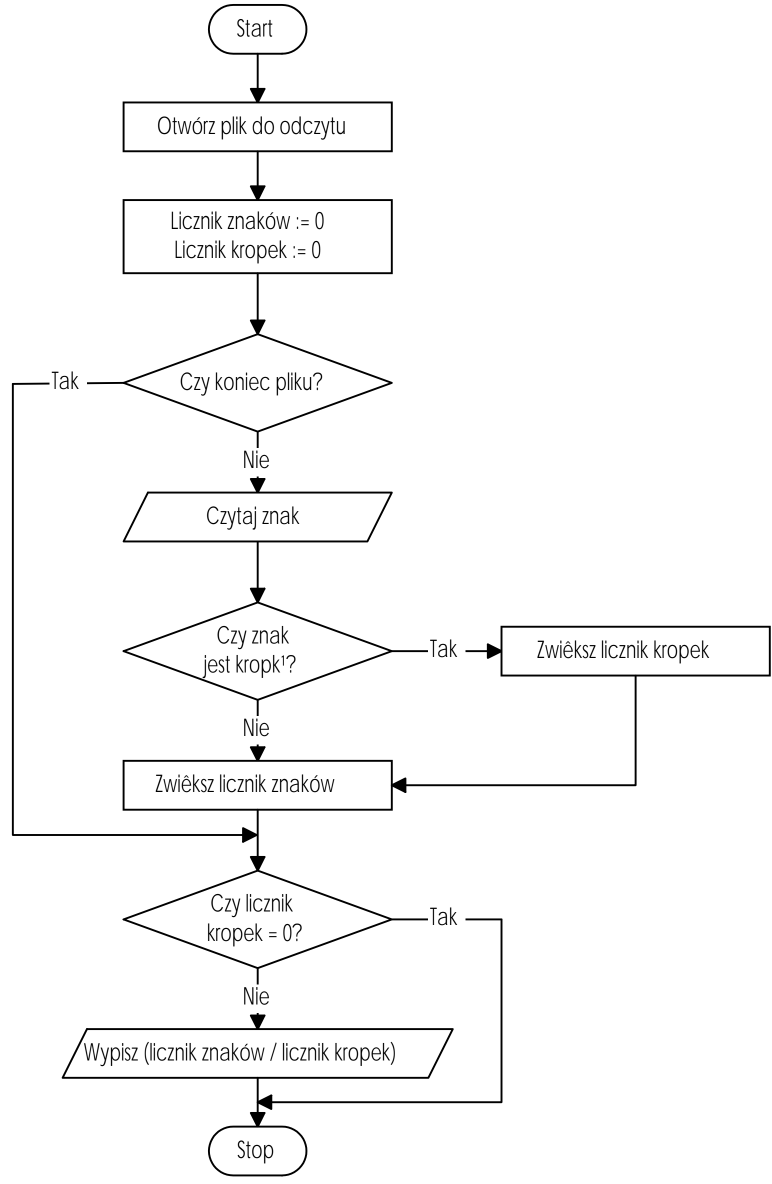
Jak zapisać algorytm? Najprostszym sposobem jest słowne wyrażenie poszczególnych operacji w postaci punktów, np. tak:

1. *otwórz plik z tekstem do odczytu*
2. *wyzeruj licznik znaków*
3. *wyzeruj licznik kropek*
4. *jeśli koniec pliku, to idź do punktu 9*
5. *wczytaj kolejny znak z pliku*
6. *jeśli znak jest kropką, zwiększ licznik kropek o 1*
7. *zwiększ licznik znaków*
8. *idź do punktu 4*
9. *jeśli licznik kropek wynosi zero, idź do punktu 11*
10. *wypisz (licznik znaków/(licznik kropek))*
11. *STOP*

To samo można przedstawić w postaci graficznej, zwanej schematem blokowym lub z angielska *flowchart*.

Schemat postępowania można wreszcie zapisać w postaci tak zwanego *pseudokodu*, czyli imitacji „prawdziwego" języka programowania:

```
start
otwórz(plik);
znaki := 0;
kropki := 0;
dopóki nie koniec_pliku(plik)
   start
      czytaj (plik,znak)
      jeżeli znak = '.'
         kropki := kropki+1;
      znaki := znaki+1;
   stop;
wypisz(znaki/kropki);
stop.
```



Rysunek 1. *Schemat blokowy obliczania średniej liczby znaków w zdaniu*

Każda z powyższych metod ma swoje wady i zalety. Zapis poszczególnych kroków w postaci punktów jest prosty, ale przy bardziej złożonych zadaniach staje się mało czytelny. Zapis w postaci schematu blokowego jest bardzo czytelny, ale pracochłonny w wykonaniu. Wreszcie zapis w pseudokodzie ma tę zaletę, że po przetłumaczeniu na język angielski staje się praktycznie gotowym tekstem programu.

W myśl tego, co powiedzieliśmy na zakończenie poprzedniego punktu, nie musisz, a nawet nie powinieneś, wymyślać algorytmu samodzielnie. Jest to co prawda zajęcie

bardzo kształcące i przynoszące dużo satysfakcji intelektualnej, jednak często okazuje się, że korzystając z gotowych rozwiązań można zrobić to samo znacznie efektywniej i szybciej. Niektóre problemy są natomiast na tyle nietypowe, iż metodę ich rozwiązania trzeba wymyślić od podstaw.

Omawiając algorytmy nie sposób pominąć kwestii struktur danych. Jest oczywiste, iż rozwiązanie jakiegokolwiek problemu wiąże się z przetwarzaniem informacji. Tę z kolei trzeba jakoś przechować, nie mówiąc już o wprowadzeniu jej do komputera i późniejszym wyprowadzeniu. Komputerowe sposoby reprezentowania informacji są znacznie ściślej określone, a jednocześnie znacznie mniej różnorodne niż te, których używamy na co dzień. W większości przypadków dobór odpowiedniej struktury danych jest automatyczny, gdyż albo narzuca się sam (np. rozwiązując zadanie matematyczne nie będziesz przedstawiał liczb w postaci znakowej), albo jest wymuszony przez algorytm. Czasem jednak sprawa nie jest tak oczywista. Przykładowo, konstruując program zajmujący się przetwarzaniem danych osobowych pracowników możesz zdecydować się na przechowywanie tychże danych w tak zwanych tablicach lub listach. Obydwa rozwiązania mają swoje zalety i wady, jak również wiążą się z zastosowaniem odpowiednich metod przetwarzania danych, czyli właśnie algorytmów.

Projekt

Jak już powiedzieliśmy, algorytm przeznaczony jest do rozwiązania zasadniczej części zadania i „nie przejmuje się” takimi detalami, jak wprowadzenie i wyprowadzenie danych czy utworzenie struktur danych. Na dodatek większość problemów jest na tyle skomplikowana, iż nie daje się rozwiązać za jednym zamachem. W takiej sytuacji musisz rozbić swoje zadanie na mniejsze fragmenty, dające się wyrazić za pomocą prostych algorytmów i uzupełnić ten zestaw przez opis operacji pomocniczych. Wszystko to należy powiązać w spójną całość, zwaną *projektem* programu.

Projekty mogą być tworzone dwiema przeciwstawnymi metodami, znanymi jako projektowanie wstępujące (ang. *bottom-up design*) oraz projektowanie zstępujące (ang. *top-down design*). Pierwsza z nich jest realizacją zasady „od szczegółu do ogółu”: na samym początku określamy zestaw elementarnych zadań, których wykonanie będzie konieczne do realizacji głównego problemu (np. wprowadzanie danych, wyprowadzanie wyników, obliczanie wartości pośrednich). Po skonstruowaniu takich cegiełek budujemy z nich większe struktury, zajmujące się przetwarzaniem odpowiednio większych fragmentów zadania, z tych — struktury jeszcze bardziej ogólne, aż w końcu dochodzimy do głównego schematu działania, który zarządza poszczególnymi modułami.

Projektowanie zstępujące, jak nietrudno się domyślać, wygląda odwrotnie. Początkiem procesu jest ogólne sformułowanie zadania, które następnie poddaje się analizie i rozbiciu na współdziałające ze sobą części, te zaś dzieli się dalej aż do uzyskania elementarnych fragmentów, których zaprogramowanie jest już łatwe, a nawet typowe. Jednocześnie na każdym etapie podziału ulegają uściśleniu kompetencje poszczególnych fragmentów programu (a raczej projektu), czyli zakres przetwarzanych przezeń danych i postać zwracanych wyników.

Niezależnie od tego, jaką metodę zastosujesz, efektem projektowania powinno być uściślenie wybranej metody rozwiązania zadania i dostosowanie jej do konkretnych okoliczności i możliwości. Osiąga się to rozbijając zadanie na elementarne składniki, których rozwiązanie zostało już opisane lub jest łatwe do samodzielnego opracowania. Gotowy projekt może przyjąć postać podobną do omawianych przy okazji algorytmów: zapisu słownego lub graficznej reprezentacji ciągu czynności. Odpowiednie zapisanie poszczególnych elementów projektu znacznie ułatwi Ci jego implementację oraz przyszłe wprowadzanie zmian i ulepszeń.

Implementacja

Uff. Przynajmniej w teorii wszystko powinno już działać. Dysponując odpowiednio rozpisanym projektem można przystąpić do mechanicznej w zasadzie czynności programowania. *Tak, to nie pomyłka*: praktycznie całość pracy umysłowej poświęcanej na rozwiązanie danego zadania poprzez programowanie skupia się w etapach opisanych poprzednio. Samo programowanie (kodowanie) jest jedynie czynnością polegającą na przetłumaczeniu zapisu projektu na odpowiedni zapis symboliczny, czyli tak zwany język programowania (najczęściej wysokiego poziomu).

Zanim do tego dojdzie, możesz jeszcze stanąć przed koniecznością wyboru odpowiedniego języka, czyli narzędzia pracy (zakładając, że masz z czego wybierać, tj. potrafisz się posługiwać kilkoma językami).

Uruchomienie

Uruchamianie świeżo napisanych programów należy do bardziej ekscytujących momentów w życiu programisty. Pełny proces uruchamiania składa się z kilku etapów, które krótko opiszemy poniżej.

Napisany program należy po pierwsze przetłumaczyć na postać wykonywalną przez komputer, czyli (na ogół) *skompilować* go. Większość narzędzi programistycznych skutecznie wyłapie przy tej okazji wszelkie błędy literowe i składniowe, które miałeś okazję popełnić podczas wpisywania programu (błędy takie zwane są ogólnie *błędami kompilacji*).

Po wyeliminowaniu wszystkich błędów kompilacji możesz spróbować wykonać program. Przedtem jednak **bezwzględnie zapisz go na dysku**. W przypadku bardziej skomplikowanych programów efekty pierwszego uruchomienia mogą być dość zaskakujące, z zawieszeniem komputera włącznie: lepiej zapamiętać program, niż narażać się na konieczność odtwarzania części lub całości wyników pracy. Wszelkie błędy, które pojawiają się w trakcie pracy już uruchomionego programu, noszą nazwę *błędów wykonania*. Błędy te mogą powodować przerwanie działania programu (a nawet zawieszenie lub restart komputera) lub jego zachowanie niezgodne z oczekiwaniami (np. brak reakcji na działania użytkownika, zwracanie błędnych wyników). Jeśli program po uruchomieniu zachowuje się w miarę poprawnie (tj. nie zawiesił się, reaguje na próby

komunikowania się z nim, zwraca w miarę poprawne wyniki i daje się zakończyć), możesz przejść do jego testowania.

Testowanie programu polega z grubsza na badaniu jego reakcji na różne zachowania użytkownika. Przede wszystkim należy sprawdzić, czy wynik pracy programu jest zgodny z naszymi oczekiwaniami. Jeśli tak, warto spróbować podrzucić mu do przetworzenia inne dane, dla których wynik znamy lub potrafimy obliczyć. Dobór danych do testowania nie powinien być przypadkowy, lecz winien opierać się na przewidywaniu możliwych słabych punktów programu. Przykładowo, jeśli program zajmuje się obsługą kartoteki pracowników, warto sprawdzić jego zachowanie w przypadku niepodania żadnego nazwiska, podania nazwiska dłuższego niż przyjęte maksimum itp. Dobrą metodą jest również testowanie programów przez użytkowników, którzy zwykle nie mają pojęcia o zasadach fair play obowiązujących w programowaniu i potrafią bardzo skutecznie „rozłożyć” programy uważane przez ich twórców za całkowicie bezbłędne. W przypadku uruchamiania bardzo opornych programów nieocenione usługi oddają wreszcie tzw. *narzędzia uruchomieniowe*, żargonowo zwane *debuggerami*. Pozwalają one na śledzenie i analizę zachowania programów instrukcja po instrukcji, co umożliwia lokalizację nawet bardzo wyrafinowanych błędów.

Zdrowy rozsadek

Jest on najważniejszym elementem procesu programowania (a dokładniej, jakiejkolwiek działalności praktycznej podejmowanej w życiu). Zastosowanie zdrowego rozsądku w programowaniu sprowadza się przede wszystkim do zachowania odpowiedniej proporcji pomiędzy treścią i formą oraz doboru metod odpowiednich do realizowanych zadań. Aby rozwiązać równanie kwadratowe możesz oczywiście napisać w Borland C++ 5.0 odpowiedni program dla Windows, wykonując uprzednio solidne przygotowanie w postaci schematu blokowego, listy zmiennych i spisu literatury. Możesz również nic nie pisać, a jedynie sięgnąć po kalkulator. W większości przypadków najlepsze jest rozwiązanie kompromisowe, tj. napisanie krótkiego programiku w Turbo Pascalu.

Samo przygotowanie teoretyczne również powinno być traktowane z umiarem. Dobór algorytmu czy opracowanie projektu bardzo często wykonywane są automatycznie, zaś ich rozpisywanie ma na celu jedynie ułatwienie ich zrozumienia i nie powinno być celem samym w sobie. Przeważająca część zadań, które będziesz rozwiązywał, jest na tyle prosta, iż nie wymaga specjalnych przygotowań. Powyższe uwagi powinieneś więc traktować raczej jako sposoby ułatwienia sobie rozwiązywania bardziej złożonych problemów, ewentualnie organizowania pracy zespołowej (o ile masz zamiar zostać zawodowym programistą).

Dysponując tymi wiadomościami możesz już spokojnie przejść do dzieła. W kolejnym rozdziale zapoznasz się z narzędziem, za pomocą którego będziesz tworzył programy.

Narzędzie

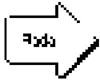
Skoro powiedzieliśmy już kilka słów na temat teorii, czas omówić podstawę działalności praktycznej — język programowania, czyli narzędzie, za pomocą którego tworzy się programy. Język programowania umożliwia zapisanie w zrozumiałej dla człowieka postaci czynności, które mają zostać zrealizowane przez komputer. Odpowiednie narzędzie, zwane *translatorem* (czyli tłumaczem), przekłada taki zapis na instrukcje zrozumiałe dla komputera. Nowoczesne narzędzia do tworzenia programów zwykle zawierają oprócz translatora również *edytor*, pozwalający na wprowadzania i poprawianie tekstu programu, *konsolidator* umożliwiający efektywne tworzenie większych programów, *program uruchomieniowy* (ang. *debugger*) ułatwiający lokalizację i usuwanie błędów w programach oraz różne narzędzia pomocnicze.

Liczba języków programowania używanych obecnie na świecie sięga setek, jednak tylko kilka zyskało sobie popularność na tyle dużą, by zastanawiać się nad ich wybraniem. Są to bez wątpienia języki Basic, C i Pascal. Który z nich wybrać? Basic jest językiem prostym i popularnym, jednak nie promuje nawyków eleganckiego programowania. Język C zyskał sobie ostatnio ogromną popularność i zdominował środowisko zawodowych programistów. Jednak i on nie nadaje się zbytnio do nauki podstaw programowania ze względu na wysoki poziom komplikacji i mało czytelną składnię. Trzeci z kandydatów, Pascal, ustępuje nieco popularnością językowi C, jest jednak znacznie czytelniejszy i przez to najczęściej używany do celów dydaktycznych.

W dalszym ciągu naszego wykładu będziemy więc używali języka Pascal. Jedną z wciąż najpopularniejszych wersji Pascala przeznaczonych dla komputerów PC jest bez wątpienia Turbo Pascal 7.0, stanowiący wręcz idealne narzędzie do nauki i tworzenia mniej skomplikowanych programów. Jego też wybierzemy jako narzędzie naszej pracy; podstawom posługiwania się środowiskiem Turbo Pascala poświęcimy resztę tego rozdziału.

Zacznijmy od uruchomienia naszego narzędzia. Na początek, rzecz jasna, musisz włączyć komputer, a po wyświetleniu znaku gotowości systemu operacyjnego (zwykle `C:\>`) napisać

```
turbo↵
```



Jeśli używasz systemu Windows, musisz wcześniej przejść do DOS-u (w tym celu znajdź i kliknij dwukrotnie ikonę MS-DOS Prompt). Jeśli Turbo Pascal jest dostępny pod postacią ikony, możesz go uruchomić klikając na niej dwukrotnie.

Może się jednak okazać, że po wydaniu powyższego polecenia ujrzysz na ekranie komunikat

```
Bad command or file name
```

oznaczający, że wywoływany program nie jest dostępny z poziomu bieżącego katalogu lub w ogóle nie został zainstalowany. W takiej sytuacji musisz odszukać katalog zawierający kompilator Turbo Pascala o nazwie TURBO.EXE (zwykle jest to katalog C:\TP\BIN lub C:\TP7\BIN) i dołączyć jego nazwę do tzw. *systemowej ścieżki dostępu*, umożliwiającej wywoływanie programów znajdujących się w katalogach innych niż bieżący. W tym celu do znajdującego się w pliku C:\AUTOEXEC.BAT polecenia PATH dopisz na końcu średnik, a po nim nazwę katalogu zawierającego Turbo Pascala, np.

```
path=c:\dos;c:\windows;c:\norton;c:\tp7
```

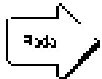
Jeżeli nie uda Ci się odszukać Turbo Pascala, będziesz musiał go zainstalować. W tym celu weź pierwszą z dyskietek instalacyjnych programu (opisaną „Install”), włóż ją do napędu A: (lub B:) i napisz

```
install↵
```

Uruchomiony w ten sposób program instalacyjny dostarczy Ci dalszych wskazówek odnośnie instalacji. Więcej informacji na temat instalacji i wywoływania Turbo Pascala znajdziesz w literaturze [1]; możesz również zasięgnąć pomocy lokalnego specjalisty od komputerów.

Jeśli udało Ci się bezkolizyjnie wywołać Turbo Pascala, na ekranie monitora powinieneś zobaczyć widok przedstawiony na sąsiednim rysunku.

Środkową część ekranu zajmuje obszerne okno, w którym znajduje się migający kursor. Okno to, zwane *okienkiem edytora* będzie służyło nam do wpisywania i poprawiania programów. Na samej górze ekranu znajduje się *menu główne*, zawierające grupy poleceń służących do zarządzania programem. Wreszcie najniższy wiersz ekranu — tak zwany *wiersz statusowy* — zawiera informacje na temat najczęściej wykorzystywanych kombinacji klawiszy lub aktualnego stanu programu. Wszystkie elementy dostępne poprzez okno edytora, menu główne oraz kombinacje klawiszy tworzą razem *zintegrowane środowisko robocze* Turbo Pascala, określane również angielskim skrótem IDE (*Integrated Development Environment*).



*Jeśli po wywołaniu Turbo Pascala zobaczysz na ekranie jedynie jasne tło, będziesz musiał otworzyć okienko edytora poleceniem **New** (Nowy) z menu **File** (Plik). Sposób posługiwania się menu poznasz za chwilę.*

Rysunek 2. *Zintegrowane środowisko robocze (IDE) Turbo Pascala*

Omawianie poszczególnych elementów IDE zaczniemy od góry ekranu, czyli od menu głównego. Menu Turbo Pascala 7.0 zawiera dziesięć pozycji:

File (*Plik*) — zawiera polecenia umożliwiające odczytywanie, zapisywanie i drukowanie tekstów programów, a także zakończenie pracy;

Edit (*Edycja*) — zawiera polecenia pozwalające na modyfikację tekstu programu i przenoszenie fragmentów tekstu pomiędzy programami;

Search (*Wyszukiwanie*) — zawiera polecenia służące do wyszukiwania i zamiany fragmentów tekstu programu oraz lokalizacji błędów;

Run (*Uruchomienie*) — zawiera polecenia sterujące wykonaniem gotowego programu;

Compile (*Kompilacja*) — zawiera polecenia umożliwiające kompilację, czyli przetłumaczenie tekstu źródłowego programu na postać wynikową — wykonywalną przez komputer;

Debug (*Usuwanie błędów*) — zawiera polecenia ułatwiające usuwanie błędów z programów i wspomagające ich uruchamianie;

Tools (*Narzędzia*) — zawiera polecenia umożliwiające wywołanie pomocniczych programów zewnętrznych;

Options (*Opcje*) — zawiera polecenia pozwalające na konfigurowanie elementów IDE (kompilatora, edytora, programu uruchomieniowego), czyli dostosowanie ich do wymogów i potrzeb użytkownika;

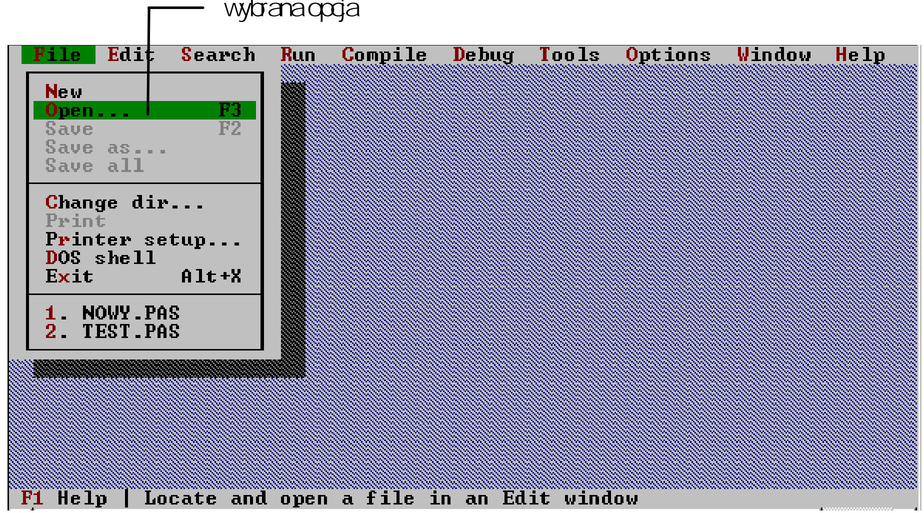
Window (*Okno*) — zawiera polecenia sterujące układem okienek na ekranie;

Help (*Pomoc*) — zawiera polecenia umożliwiające dostęp do obszernego systemu pomocy (suflera) Turbo Pascala.

Jak dostać się do poleceń zawartych w menu? Metod jest kilka. Możesz np. nacisnąć klawisz F10, a następnie wybrać żądaną pozycję (opcję) menu głównego używając klawiszy kursora (strzałek) ← oraz → i „rozwinąć" ją naciskając ENTER. Znacznie prościej jest jednak wykorzystać kombinację klawisza ALT z wyróżnioną w nazwie opcji literą (na przykład dla menu ***File*** będzie musisz jednocześnie nacisnąć klawisze ALT i F). Trzecią możliwością jest użycie myszki — w tym celu musisz kliknąć jej lewym przyciskiem po wskazaniu wybranej pozycji menu głównego.

Niezależnie od tego, której metody użyjesz, wybranie dowolnej pozycji z menu głównego spowoduje pojawienie się kolejnego menu (tzw. *menu rozwijanego* lub z angielska *pull-down menu*), zawierającego bardziej szczegółowy wykaz poleceń związanych z danym tematem. Aby wybrać żądane polecenie, musisz tym razem użyć klawiszy ↑ lub ↓ (i oczywiście ENTER), nacisnąć klawisz odpowiadający wyróżnionej literze (uwaga: tym razem bez klawisza ALT!), ewentualnie kliknąć w odpowiednim miejscu myszką. Zauważ przy tej okazji, że niektórym poleceniom towarzyszą nazwy klawiszy funkcyjnych, specjalnych lub ich kombinacji, wyświetlone w menu z prawej strony. Są to tak zwane *klawisze skrótu* (*shortcut keys*), umożliwiające wywołanie danego polecenia bez rozwijania menu. Użycie klawiszy skrótu znacznie upraszcza korzystanie z IDE, toteż warto zapamiętać kilka z nich (np. F2 — zapisanie programu na dysku, CTRL-F9 — uruchomienie programu itd.).

Oto przykład: aby otworzyć nowe okienko edytora, musisz wydać polecenie ***File-New***, czyli wybrać z menu głównego (F10) pozycję ***File*** (F), a z rozwiniętego menu polecenie ***New*** (*Nowy*). Menu ***File*** możesz również rozwinąć za pomocą myszki lub naciskając jednocześnie klawisze ALT i F (ALT-F). Aby wreszcie zakończyć pracę w IDE, wystarczy nacisnąć klawisz skrótu ALT-X (można zrobić to samo za pomocą menu, chociaż jest to nieco bardziej czasochłonne).



Rysunek 3. *Rozwinięte menu **File***

Zauważ przy okazji, że rozwinięcie menu wiąże się ze zmianą zawartości wiersza statusowego, w którym będzie się obecnie znajdował krótki opis wybranego polecenia lub opcji.

Skoro potrafimy już otworzyć okienko edytora, możemy wypróbować jego możliwości. Treść wpisywanego tekstu nie ma na razie znaczenia, ale jeśli chcesz, możesz od razu zajrzeć do następnego rozdziału i wpisać znajdujący się tam program przykładowy. Jak już prawdopodobnie wiesz, do wprowadzania tekstu służą tzw. *klawisze alfanumeryczne*, czyli główny blok klawiszy zawierający litery QWERTY... i cyfry. Blok ten zawiera również kilka dodatkowych klawiszy zawierających symbole (np. + czy >) oraz klawisze specjalne, którymi zajmiemy się niżej. Zauważ, że podczas wpisywania kursor (migający znak podkreślenia) przesuwa się tak, by zawsze wskazywać na pozycję kolejnego wprowadzanego znaku, zaś po przekroczeniu prawego brzegu okienka tekst nie jest automatycznie łamany (jak w większości edytorów tekstu). Jeśli rzucisz okiem na ramkę otaczającą okienko edytora, to w okolicy lewego dolnego rogu zauważysz gwiazdkę i dwie liczby rozdzielone dwukropkiem. Gwiazdka sygnalizuje, że zawartość okienka edytora została zmodyfikowana (przez dopisania lub usunięcie tekstu), zaś liczby oznaczają numer wiersza i kolumny, w których znajduje się kursor. Okienko edytora ma szerokość co najwyżej 78 znaków (można je pomniejszyć) i dobrym zwyczajem jest nieprzekraczanie tej właśnie długości wiersza.

Znajdujące się na klawiaturze klawisze specjalne i funkcyjne nie służą do bezpośredniego wprowadzania tekstu, umożliwiają jednak jego poprawianie i sterowanie działaniem IDE. Działaniem klawiszy sterujących nie będziemy się na razie zajmować: na obecnym etapie znajomości Turbo Pascala wystarczy Ci znajomość kombinacji ALT-X (zakończenie pracy) i klawisza F1 (wywołanie pomocy). Poniżej omówimy krótko działanie klawiszy edycyjnych.

Klawisz ENTER, jak już zapewne wiesz, służy do przejścia do nowego wiersza. Jego naciśnięcie powoduje przemieszczenie kursora do początku kolejnego wiersza (znajdujący się pod spodem tekst jest przesuwany w dół), zaś jeśli włączony jest tzw. tryb automatycznego wcinania (ang. *autoindent mode*), kursor lokuje się nie w pierwszej kolumnie, lecz na wysokości początku poprzedniego wiersza.

Klawisze BACKSPACE (położony nad ENTER) i DELETE (znajdujący się z prawej strony klawiatury w bloku klawiszy numerycznych i sterowania kursorem) pozwalają na usuwanie tekstu. Naciśnięcie klawisza BACKSPACE usuwa znak położony na lewo od kursora (i przesuwa kursor w lewo), zaś klawisz DELETE usuwa znak znajdujący się „pod" kursorem, nie przesuwając tego ostatniego. Aby usunąć wiersz wskazany kursorem, wystarczy nacisnąć klawisze CTRL-Y. Warto również wiedzieć, że omyłkowo usunięty tekst można odtworzyć naciskając klawisze ALT-BACKSPACE (odpowiadają one poleceniu ***Undo*** z menu ***Edit***). Możliwość ta przydaje się zwłaszcza osobom roztargnionym.

Klawisz TAB (tabulator) powoduje przesunięcie kursora o kilka kolumn do przodu, co umożliwia *wcinanie tekstu*. Używanie wcięć podczas pisania programów jest sprawą bardzo ważną, pozwala bowiem na zachowanie przejrzystości i czytelnego układu tekstu. Do sprawy wcinania wrócimy jeszcze w kolejnych rozdziałach.

Kolejna grupa klawiszy, w większości klawiatur ulokowana pomiędzy blokiem alfanumerycznym a numerycznym, służy do poruszania się po tekście. Są to klawisze strzałek (przesuwające kursor o jedną pozycję lub wiersz) oraz klawisze HOME, END, PAGEUP i PAGEDOWN, umożliwiające przemieszczenie kursora do początku lub końca wiersza oraz wyświetlenie poprzedniego lub następnego ekranu (strony) tekstu.

Naciśnięcie klawisza INS pozwala na wybór pomiędzy tzw. *trybem wstawiania* (ang. *insert mode* — wpisywane znaki są wstawiane w już istniejący tekst i „rozpychają" go) i *trybem zastępowania* (ang. *overwrite mode* — nowo wpisywane znaki zastępują już istniejące). Działanie obu trybów możesz sprawdzić ustawiając kursor „wewnątrz" już napisanego wiersza tekstu i wpisując nowy tekst z klawiatury. Zwróć również uwagę na kształt kursora oraz działanie klawiszy ENTER i TAB.

Ostatnim ważnym klawiszem jest ESCAPE (ESC). Służy on (jak sama nazwa sugeruje) do wycofywania się z poczynionych działań (zamykania pól dialogowych, „zwijania" menu itp.). Jeśli przez przypadek trafisz w jakieś nieznane Ci okolice, możesz spróbować przywrócić poprzedni stan IDE właśnie naciskając ESC. Funkcji tego klawisza nie powinieneś jednak mylić z poleceniem Undo (ALT-BACKSPACE), które umożliwia anulowanie zmian dokonanych w treści wpisywanego programu.

Przedstawione wyżej wiadomości powinny wystarczyć Ci do rozpoczęcia pracy z prostymi programami. Jeśli chcesz, możesz pominąć kolejny podrozdział i od razu przejść do następnego rozdziału, z którego dowiesz się, jak napisać i uruchomić najprostszy program w Pascalu. Jeżeli interesują Cię dodatkowe funkcje i możliwości edytora, zapraszam do dalszej lektury.

Zaawansowane funkcje edytora

Omówimy obecnie najczęściej używane z dodatkowych funkcji edytora Turbo Pascala, a mianowicie operacje blokowe oraz wyszukiwanie i zastępowanie fragmentów tekstu.

Podczas pisania większych programów często zachodzi konieczność powielenia, przeniesienia lub usunięcia większych fragmentów tekstu — zarówno wewnątrz pojedynczego okienka edytora (w obrębie jednego pliku), jak i pomiędzy okienkami (plikami). Zamiast przepisywać tekst ręcznie, można wykorzystać tak zwane *operacje blokowe*, umożliwiające manipulowanie całymi jego fragmentami.

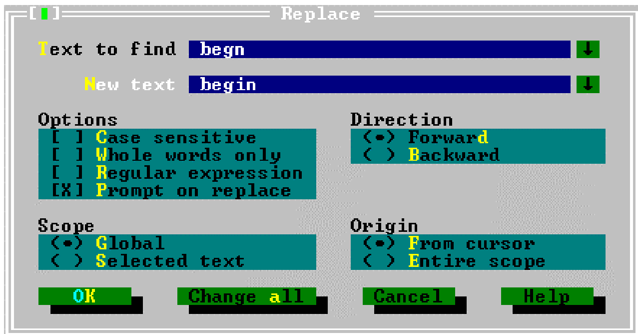
Pierwszą czynnością jest zaznaczenie fragmentu tekstu, co najłatwiej zrobić naciskając klawisze sterujące kursorem (również HOME, END, PAGEUP i PAGEDOWN) przy wciśniętym klawiszu SHIFT. Zaznaczany fragment zostaje na ogół wyświetlony w negatywie. Blok można również zaznaczyć za pomocą myszki, przesuwając jej kursor przy wciśniętym lewym przycisku (jest to tzw. *ciągnięcie* — ang. *dragging*)[2].

---

[2] Trzecią możliwością jest użycie klawiszy skrótu CTRL-K, B i CTRL-K, H, będących zaszłością z poprzednich wersji Turbo Pascala i edytora WordStar. Dokładniejsze informacje na ten temat znajdziesz w literaturze i systemie pomocy.

Po zaznaczeniu odpowiedniego fragmentu możemy go przenosić i powielać, wykorzystując tzw. *schowek* (ang. *clipboard*). W celu skopiowania zaznaczonego fragmentu do schowka musisz nacisnąć klawisze CTRL-INS (kombinacja SHIFT-DEL przeniesie fragment do schowka, usuwając go z tekstu). Aby wstawić tekst ze schowka w wybrane miejsce, wskaż je kursorem i naciśnij klawisze SHIFT-INS. Opisane operacje umożliwią Ci nie tylko powielanie i przenoszenie fragmentów zawartości aktywnego okienka edytora, lecz również przenoszenie tekstu do innych okienek. Możesz również zapisać zaznaczony blok na dysku (CTRL-K, W) i odczytać go z dysku (CTRL-K, R). Aby usunąć wyróżnienie, użyj kombinacji CTRL-K, H. Aktualną zawartość schowka możesz wreszcie podejrzeć poleceniem ***Show clipboard*** z menu ***Edit*** (powoduje ono wyświetlenie okienka zatytułowanego ***Clipboard***, które możesz zamknąć naciśnięciem klawiszy ALT-F3).

Drugą często wykorzystywaną własnością edytora jest wyszukiwanie i zamiana tekstu. Pozwala to na szybkie znalezienie w programie żądanego fragmentu tekstu lub zastąpienie go innym tekstem. Funkcje te wywoływane są odpowiednio kombinacjami klawiszy CTRL-Q, F i CTRL-Q, A (naciśnięcie klawiszy CTRL-L powoduje wyszukanie lub zamianę kolejnego wystąpienia danego fragmentu). Ponieważ operacja wyszukania lub zamiany wymaga ustalenia kilku parametrów (jak choćby poszukiwany tekst), wywołanie jednej z omawianych funkcji powoduje wyświetlenie tzw.*okna dialogowego*. Poniżej pokazano okno dialogowe wyświetlane dla operacji zamiany tekstu.



Rysunek 4. O*kno dialogowe wyświetlane dla operacji zamiany tekstu*

Znajdujące się u góry okienka dwa *pola tekstowe* służą do wprowadzenia tekstu, który ma być zastąpiony (***Text to find***) i tekstu zastępującego (***New text***). Poniżej położone są cztery *grupy*, zawierające spokrewnione ze sobą opcje. Grupa ***Options*** umożliwia ustalenie parametrów przeszukiwania (rozróżnianie małych i dużych liter, wyszukiwanie tylko całych słów i tzw. wyrażeń regularnych, zamiana z potwierdzeniem). Grupa ***Direction*** pozwala na określenie kierunku wyszukiwania (w przód lub wstecz). Grupa ***Scope*** pozwala na zawężenie zakresu działania funkcji do zaznaczonego bloku tekstu, zaś grupa ***Origin*** umożliwia rozpoczęcie wyszukiwania od bieżącej pozycji kursora lub od początku tekstu. Na samym dole okienka znajdują się cztery *przyciski* służące do bezpośredniego uruchomienia (wywołania) lub anulowania wybranej czynności. Przycisk ***OK*** powoduje rozpoczęcie operacji z ustalonymi wcześniej parametrami, ***Change***

all pozwala na zamianę wszystkich wystąpień poszukiwanego tekstu, ***Cancel*** umożliwia rezygnację z wykonywania operacji, zaś ***Help*** wywołuje system pomocy.

Jeśli już wypróbowałeś działanie funkcji ***Replace*** (funkcja wyszukiwania, ***Find***, działa podobnie, nie powoduje jednak zamiany znalezionego tekstu, a jedynie wskazuje go kursorem), spróbujemy obecnie omówić konstrukcję i działanie pola dialogowego. Pola dialogowe są powszechnie wykorzystywane przez IDE do komunikacji z użytkownikiem, toteż warto od razu zaznajomić się z ich elementami.

Wyboru elementu pola dialogowego dokonujesz kliknięciem myszką lub naciśnięciem klawisza TAB (albo SHIFT-TAB — wybór elementu poprzedniego). Zwróć uwagę, że w przypadku grupy naciśnięcie TAB udostępnia jej pierwszy element — aby dostać się do pozostałych, musisz użyć klawiszy kursora. Wybranie elementu sygnalizowane jest przez jego wyróżnienie (na ogół kolorem) i umożliwia zmianę jego stanu. W przypadku pola tekstowego zmianą stanu jest po prostu wprowadzenie tekstu. Działanie elementów grup wymaga nieco szerszego omówienia. Jak zapewne zauważyłeś, grupy zawarte w polu ***Replace*** zawierają dwa rodzaje „elementów manipulacyjnych”: tak zwane *pola wyboru* (ang. *check box*), oznaczone nawiasami kwadratowymi, oraz *przyciski opcji* (ang. *radio button*), oznaczone nawiasami okrągłymi. Pola wyboru umożliwiają niezależne włączenie lub wyłączenie danej funkcji; jak łatwo sprawdzić, opcje umieszczone w grupie ***Options*** nie wykluczają się wzajemnie i mogą być włączane niezależnie od siebie. Z kolei przyciski opcji pozwalają na wybór jednej z kilku wzajemnie wykluczających się możliwości (np. przeszukiwanie tekstu może odbywać się albo do przodu, albo do tyłu), toteż włączenie jednego z nich powoduje automatyczne wyłączenie pozostałych. Stan włączenia sygnalizowany jest dla pola wyboru symbolem [X], zaś dla przycisku opcji — symbolem (•).

Ostatnią grupę elementów pola dialogowego stanowią *przyciski* (ang. *button*). „Naciśnięcie” przycisku (tj. wyróżnienie go za pomocą klawisza TAB i naciśnięcie spacji lub klawisza ENTER, ewentualnie kliknięcie myszką) powoduje natychmiastowe wykonanie odpowiadającej mu czynności i (na ogół) zamknięcie pola dialogowego. Trzy spośród znajdujących się w polu ***Replace*** przycisków — ***OK***, ***Cancel*** i ***Help*** — znajdziesz praktycznie w każdym innym polu dialogowym. Jak łatwo się domyślić, „naciśnięcie” przycisku ***OK*** powoduje zaakceptowanie informacji wprowadzonej do pola dialogowego, jego zamknięcie i wykonanie danej operacji (np. zamiany znaków). Przycisk ***Cancel*** pozwala na rezygnację z wykonywania operacji (wprowadzone do pola informacje są tracone), zaś przycisk ***Help*** wywołuje okienko systemu pomocy zawierające odpowiednią do kontekstu informację. Warto zapamiętać, że omówione wyżej przyciski mają swoje odpowiedniki na klawiaturze: są to mianowicie ENTER (odpowiadający przyciskowi ***OK***, o ile ten został wyróżniony, tzn. jest tzw. przyciskiem domyślnym), ESCAPE (odpowiadający przyciskow ***Cancel***) oraz F1 (odpowiadający przyciskowi ***Help***).

Warto jeszcze powiedzieć kilka słów na temat systemu pomocy, czyli właśnie funkcji ***Help***. „Podręczna ściągawka” dostępna w IDE oferuje, oprócz bardzo szerokiego zestawu informacji, kilka nader przydatnych funkcji. Przede wszystkim system pomocy Turbo Pascala jest tzw. *systemem kontekstowym*. Oznacza to, że po naciśnięciu klawisza F1 uzyskasz informację ściśle związaną z sytuacją, w jakiej się znajdujesz (np. po

otwarciu okna dialogowego ***Replace*** system pomocy wyświetli opis tegoż pola i jego elementów). Kontekstowość oznacza również możliwość łatwego uzyskania informacji na temat błędów sygnalizowanych przez kompilator.

Drugą ciekawą cechą systemu pomocy jest możliwość uzyskania informacji o wybranym elemencie programu (o ile jego nazwa została zdefiniowana w ramach Turbo Pascala). Funkcja ta wywoływana jest naciśnięciem klawiszy CTRL-F1, po uprzednim wskazaniu interesującego nas słowa kursorem. Tak więc, aby dowiedzieć się czegoś na temat procedury `write` (będziesz z niej korzystał w następnym rozdziale), musisz napisać `write` i nacisnąć CTRL-F1 pamiętając, by kursor znajdował się w obrębie napisanego słowa. W podobny sposób możesz uzyskać informację na temat innych obiektów.

Dostęp do skorowidza haseł opisanych w systemie pomocy możesz uzyskać naciskając klawisze SHIFT-F1. Aby wybrać hasło ze skorowidza, wystarczy wskazać je kursorem i nacisnąć ENTER (ewentualnie kliknąć myszką). Podczas korzystania z systemu pomocy istnieje również możliwość powrotu do poprzednio wyświetlanych opisów przez naciśnięcie klawiszy ALT-F1. Pozostałe, rzadziej używane polecenia zgrupowane są w menu ***Help***.

W ten sposób zakończyliśmy przegląd podstawowych informacji na temat środowiska Turbo Pascala. Możesz teraz poćwiczyć posługiwanie się edytorem i poznanymi funkcjami, zakończyć pracę naciskając klawisze ALT-X lub od razu przejść do następnego rozdziału, w którym zademonstrujemy pierwszy program w Pascalu.

Zapamietaj

- Aby uruchomić Turbo Pascala, wydaj polecenie `turbo`.
- Chcąc zakończyć pracę z Turbo Pascalem, naciśnij klawisze ALT-X.
- Wszystkie polecenia IDE Turbo Pascala dostępne są poprzez menu główne (wywoływane klawiszem F10 lub za pomocą myszki). Część poleceń można również wywołać naciskając klawisze skrótu.
- Do wprowadzania i poprawiania tekstów programów służy okienko edytora.
- Konwersacja z IDE prowadzona jest za pomocą pól dialogowych.
- Klawisz ESCAPE umożliwia Ci rezygnację z wykonywania rozpoczętej operacji.
- Jeśli potrzebujesz pomocy, naciśnij klawisz F1.

Pierwszy program

Najwyższa pora na napisanie jakiegoś programu. By nie był on całkowicie niepraktyczny, spróbujmy za pomocą komputera obliczyć pole koła o promieniu 5 centymetrów. Jak wiadomo, pole koła wyraża się wzorem

$$S = \pi r^2$$

gdzie *r* oznacza promień, zaś π jest znaną od dość dawna stałą wynoszącą około 3.14. Sam program powinien jedynie wypisywać obliczoną wartość na ekranie, czyli działać z grubsza tak:

początek
wypisz pi razy r do kwadratu
koniec

OK. Pomijając niejasne na razie kwestie mnożenia, podnoszenia liczby do kwadratu i wartości π, możesz włączyć komputer i uruchomić Turbo Pascala, pisząc

```
turbo
```

Po pojawieniu się na ekranie okienka edytora (jeśli takowe się nie pojawi, z menu ***File*** wybierz polecenie ***New***, czyli *Nowy*), spróbuj przetłumaczyć nasz schemacik na język angielski i wprowadzić go do komputera. Powinno to wyglądać mniej więcej tak:

```
begin
write(Pi*5*5);
end.
```

Zauważ, że powyższy program jest prawie dosłownym tłumaczeniem tego, co napisaliśmy poprzednio. Ta właśnie „dosłowność” stanowi jedną z najpoważniejszych zalet Turbo Pascala: tworzenie prostych programów można w dużej części sprowadzić do tłumaczenia odpowiedniego algorytmu na język angielski (warto jeszcze znać podstawy angielskiego, ale naprawdę wystarczą podstawy).

Sam program chyba nie może być prostszy. Składa się on z jednej jedynej instrukcji wypisującej wartość *wyrażenia* „pi razy r do kwadratu”, przy czym podnoszenie liczby do kwadratu zostało zastąpione wymnożeniem jej przez siebie. Samo mnożenie symbolizowane jest *operatorem* * (gwiazdka), zaś stała π to po prostu `Pi`. O wyrażeniach,

operatorach i stałych powiemy nieco więcej już wkrótce, na razie zaś wyjaśnijmy rolę pozostałych składników programu.

Każdy program w Pascalu rozpoczyna się słowem kluczowym **begin** (*początek*) i kończy słowem kluczowym **end.** (*koniec* — z kropką). *Słowa kluczowe* stanowią podstawowy „alfabet" języka, służący do tworzenia bardziej złożonych struktur. Turbo Pascal automatycznie wyróżnia słowa kluczowe w treści programu przez wyświetlanie ich w innym kolorze, natomiast w książce będziemy zaznaczali je **czcionką pogrubioną**.

Pomiędzy słowami kluczowymi oznaczającymi początek i koniec programu znajduje się jego zasadnicza treść, czyli tak zwana *część operacyjna*. Składają się na nią *instrukcje* opisujące kolejne czynności wykonywane przez komputer. W naszym przypadku program zawiera tylko jedną instrukcję, a mianowicie wywołanie tzw. procedury bibliotecznej `write` (*wypisz*) wypisującej na ekranie uprzednio obliczoną wartość wyrażenia. Jak się wkrótce przekonasz, instrukcje mogą rówież zawierać zapis działań arytmetycznych, wywołania funkcji systemowych i wiele innych operacji. Każda instrukcja, z wyjątkiem instrukcji znajdującej się bezpośrednio przed słowem **end**, musi być zakończona średnikiem (`;`), który w Pascalu jest tzw. *separatorem instrukcji*.

Aby procedura `write` wiedziała, co właściwie ma wypisać, musisz przekazać jej odpowiednie informacje w postaci tzw. *argumentów*. W naszym przypadku argumentem procedury `writeln` jest wyrażenie `Pi*5*5`, które z kolei składa się z identyfikatorów, stałych i operatorów (o tym za chwilę). Jeśli procedura wymaga kilku argumentów, należy rozdzielić je przecinkami, zaś cała lista argumentów musi być zawsze ujęta w nawiasy okrągłe. Tak więc składnia (czyli symboliczny opis sposobu wykorzystania) naszej procedury `write` jest następująca:

```
write[(argument, argument...)]
```

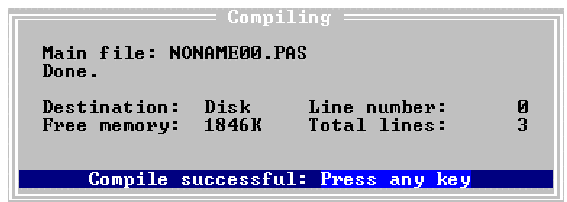
W powyższym zapisie nawiasy kwadratowe oznaczają elementy nieobowiązkowe (możliwe jest użycie procedury `write` bez argumentów), zaś wielokropek sygnalizuje możliwość użycia dalszych argumentów w bliżej nieokreślonej liczbie. Poprawne będą więc następujące wywołania procedury `write`:

- `write(5);` wypisze liczbę 5
- `write(1, 2, 3, 4);` wypisze liczby 1, 2, 3 i 4 (jako „1234")
- `write;` nic nie wypisze

Pozostało nam jeszcze wytłumaczyć pojęcia stałej, operatora i identyfikatora. *Stała* to po prostu... stała, czyli pewna stała wartość zapisana jawnie (na przykład liczba 5 czy nazwa 'Warszawa'). O tak zwanych *stałych symbolicznych* powiemy nieco później. *Operatorem* nazywamy symbol reprezentujący pewne działanie, np. dodawanie (+) czy porównanie (=). Operatory, będące słowami zastrzeżonymi, stanowią jeden z podstawowych elementów *wyrażeń*, o których powiemy więcej w następnym rozdziale. Wreszcie *identyfikator* jest po prostu nazwą obiektu (np. stałej π) lub operacji (na przykład

wypisywania wartości — `write`). Może on zawierać litery (duże litery nie są odróżniane od małych, zatem `Write` i `write` to ten sam identyfikator), cyfry i znaki podkreślenia, lecz nie może zaczynać się od cyfry. Musisz również pamiętać, że tworzenie identyfikatorów brzmiących tak samo, jak słowa kluczowe i operatory (będące słowami zastrzeżonymi) jest niedozwolone, tak więc nie możesz utworzyć obiektu o nazwie `End` czy `=`.

Tyle teorii. Żeby przekonać się, że program naprawdę działa, spróbuj go skompilować (naciskając klawisz F9 — ***Make***). O ile nic nie popsułeś przy przepisywaniu, wynikiem kompilacji powinno być przedstawione niżej okienko.



Rysunek 5. *Okienko wyświetlane po prawidłowej kompilacji programu*

Nazwa pliku, rozmiar wolnej pamięci i inne szczegóły mogą być u Ciebie nieco inne — najważniejszy jest komunikat `Compile successful: Press any key` wyświetlany u dołu okienka.

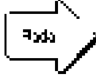
Jeśli udało Ci się popełnić jakiś błąd, kompilator zasygnalizuje to odpowiednim komunikatem, wskazując jednocześnie kursorem podejrzane miejsce w programie.

```
 File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
[■]                        NONAME00.PAS                         1=[↕]
 Error 10: Unexpected end of file.
  write(Pi*5*5)
end



 3:1
F1 Help  F2 Save  F3 Open  Alt+F9 Compile  F9 Make  Alt+F10 Local menu
```

Rysunek 6. *Kompilator sygnalizuje błąd kompilacji*

Powyżej zilustrowano reakcję kompilatora na brak kropki po kończącym program słowie **`end`**. Jak widać, kursor znalazł się w wierszu zawierającym błąd, zaś wyświetlony na ekranie komunikat oznacza, że treść programu nie została prawidłowo zakończona. Dodatkowe wyjaśnienia dotyczące błędu możesz uzyskać naciskając klawisz F1 (***Help***).

Jeśli w programie znajduje się kilka błędów, kompilator wykryje tylko pierwszy z nich (po jego poprawieniu — następny i tak dalej). W charakterze ćwiczenia możesz wprowadzić kilka błędów do programu (np. przestawić litery w słowie `write`) i przekonać się, jak reaguje na nie kompilator.



Musisz zdawać sobie sprawę, że bezbłędna kompilacja nie zawsze gwarantuje poprawne wykonanie programu. Błędy, o których powiedziano przed chwilą, nazywane są ***błędami kompilacji*** *(ang. compile-time error) i wynikają głównie z pomyłek podczas wpisywania programu (np. błędnie wpisanych słów kluczowych). Drugą, znacznie mniej przyjemną kategorię tworzą* ***błędy wykonania*** *(ang. runtime error), wynikające z niewłaściwej konstrukcji programu lub użycia nieprawidłowych danych. Błędy te nie są wykrywane przez kompilator i ujawniają się dopiero w trakcie pracy programu, prowadząc najczęściej do jej przerwania. Typowym przykładem jest próba dzielenia liczby przez zero lub odczytania danych z nieistniejącego pliku (kompilator nie może wiedzieć, czy plik o danej nazwie będzie istniał w chwili wykonania programu). Na szczęście programy, którymi będziemy się zajmować w najbliższym czasie, są na tyle proste, że trudno w nich o błędy wykonania.*

Tyle na temat błędów, których oczywiście życzę Ci jak najmniej. Jeśli już skompilowałeś program, możesz go wykonać, naciskając klawisze CTRL-F9.

Już? I jaki jest wynik? Hm... a właściwie *gdzie* on jest?

Nie ma się czego obawiać. Obliczone w programie pole koła zostało wypisane na ekranie, lecz jest przesłonięte przez okienko edytora. Aby się o tym przekonać, naciśnij klawisze ALT-F5: powinieneś zobaczyć znajdujący się „pod spodem" ekran DOS-owy, zawierający (oprócz innych rzeczy) liczbę

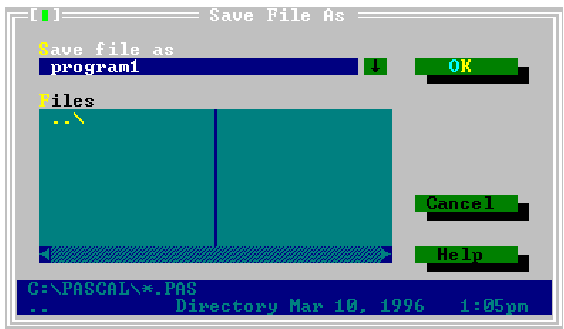
```
7.8539816340E+01
```

czyli z grubsza 78.5 (centymetrów kwadratowych). Aby wrócić do okienka edytora, wystarczy nacisnąć ENTER.

Uff. Na pierwszy raz to chyba wystarczy. Aby nie utracić swojego pierwszego programu, musisz go jeszcze zapamiętać, czyli zapisać na dysku. W tym celu wybierz polecenie ***Save*** (*Zapamiętaj*) z menu ***File*** lub po prostu naciśnij klawisz F2. Jeśli wcześniej nie nadałeś programowi nazwy (nazwa wyświetlana jest na górnej krawędzi ramki okienka edytora; jeśli znajduje się tam napis NONAME00.PAS, program nie został jeszcze nazwany), na ekranie pojawi się pole dialogowe ***Save File As***.

Aby zapisać program pod przykładową nazwą PROGRAM1.PAS, wystarczy wpisać ją w pole ***Save file as*** i nacisnąć ENTER. Ponieważ rozszerzenie .PAS jest dla programów pascalowych przyjmowane domyślnie, nie musisz go podawać.

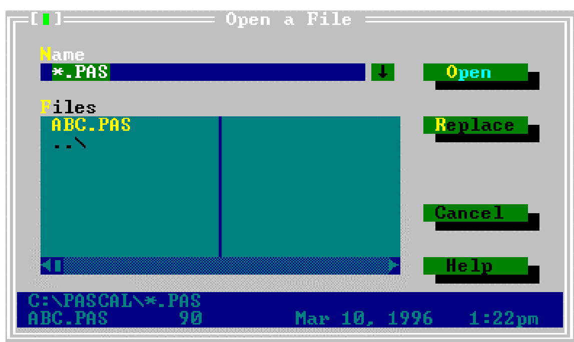
Zauważ, że po wykonaniu tej operacji nazwa programu wyświetlana w ramce okienka edytora zmieniła się z NONAME00.PAS na PROGRAM1.PAS. Jeśli teraz ponownie naciśniesz F2, program zostanie zachowany pod tą samą nazwą bez wyświetlania pola dialogowego.



Rysunek 7. *Okno dialogowe* ***Save File As***

Tworzenie nowego programu poleceniem ***File-New*** i późniejsza zmiana nazwy poleceniem ***File-Save/Save as*** nie są zbyt wygodne. Oto dwie metody utworzenia programu nie wymagające późniejszej zmiany nazwy.

Zamiast polecenia ***New*** możesz wykorzystać polecenie ***Open*** (*Otwórz*) z menu ***File***. Pozwala ono na odczytanie z dysku zapisanego wcześniej tekstu programu, wyświetlając okno dialogowe ***Open a File*** pokazane poniżej.



Rysunek 8. *Okno dialogowe* ***Open a File***

Nazwę żądanego programu możesz wpisać w pole ***Name*** lub wybrać z listy ***Files***, zawierającej zawartość katalogu roboczego; następnie należy nacisnąć ENTER. Jeśli plik o podanej nazwie nie istnieje, Turbo Pascal utworzy nowe, odpowiednio zatytułowane okienko edytora, którego zawartość będzie można w dowolnym momencie zapisać bez konieczności podawania nazwy.

Druga metoda polega na odpowiednim wywołaniu Turbo Pascala z poziomu DOS-u. Aby otworzyć istniejący plik z programem lub utworzyć nowy plik, wystarczy napisać

```
turbo nazwa↵
```

Po uruchomieniu system automatycznie odczyta plik o zadanej `nazwie` (i rozszerzeniu .PAS; podobnie jak poprzednio, rozszerzenia nie trzeba podawać) lub, jeśli taki plik nie istnieje, otworzy puste okienko edytora o odpowiednim tytule.

W ten sposób poznałeś absolutne podstawy posługiwania się środowiskiem Turbo Pascala i uruchomiłeś swój pierwszy program. A oto kilka ćwiczeń, które pomogą Ci nabrać wprawy i zdobyć dodatkowe wiadomości:

- wykorzystaj system pomocy do wyświetlenia informacji o poszczególnych elementach programu (słowach kluczowych, identyfikatorach, operatorach);
- spróbuj wprowadzić do programu kilka błędów. Jaka jest reakcja kompilatora?
- popraw program tak, by dało się za jego pomocą obliczyć pole koła o promieniu 2 cm, 10 cm itd. Spróbuj napisać program obliczający pole trójkąta o znanej długości podstawy i wysokości;
- zastanów się, co należałoby zmienić w przedstawionym programie, by uczynić go lepszym (bardziej czytelnym, uniwersalnym, praktycznym, łatwiejszym w obsłudze).

Zapamiętaj

- Program pascalowy składa się z ciągu rozdzielonych średnikami instrukcji położonych pomiędzy słowami kluczowymi **`begin`** i **`end`**.
- Instrukcje stanowią symboliczny zapis operacji, które ma wykonać komputer.
- Instrukcje mogą zawierać wyrażenia oraz wywołania funkcji i procedur.
- Wyrażenia składają się ze stałych, operatorów i identyfikatorów.
- Identyfikatory są nazwami obiektów składających się na program. Mogą one zawierać litery, cyfry i znaki podkreślenia, nie mogą jednak zaczynać się od cyfr.
- Aby skompilować program, musisz użyć polecenia ***Make*** (F9). Ewentualne błędy kompilacji są odpowiednio sygnalizowane.
- Chcąc uruchomić program, musisz wydać polecenie ***Run*** (CTRL-F9). Do obejrzenia wyników działania programu może okazać się niezbędne polecenie ***User Screen*** (ALT-F5).
- Aby zapisać program, użyj polecenia ***Save*** (F2). Do odczytania zapisanego programu z dysku służy polecenie ***Open*** (F3).

Nic nie jest doskonałe...

... a zwłaszcza pierwszy w życiu program w Pascalu. Jeżeli próbowałeś zastanowić się nad pytaniem zamykającym poprzedni rozdział, najprawdopodobniej widzisz już możliwości ulepszenia swego dzieła. Na początek spróbujemy zająć się czytelnością programu, a następnie zastanowimy się, jak uczynić go bardziej praktycznym.

Co właściwie robi nasz program? „Oblicza pole koła” — odpowiesz. Skąd to wiadomo? „Tak było napisane w książce”. A jeśli ktoś nie czytał książki? „Może się domyślić po przeczytaniu programu”.

Ale po co ma czytać program? Co prawda zrozumienie jednej instrukcji to żaden problem, ale jeśli instrukcji będzie sto...? Znacznie lepiej byłoby umieścić w programie informację opisującą jego przeznaczenie i (ewentualnie) metodę realizacji zadania. Można to zrobić na kilka sposobów: umieszczając odpowiednie zapisy w programie oraz używając stosownej nazwy pliku. Zróbmy to od razu:

```
program Pole_kola;
{ Program oblicza pole koła o promieniu 5 }

begin
  write(Pi*5*5);         (* oblicz i wypisz pole koła *)
end.
```

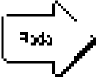
W porównaniu do poprzedniej wersji programu pojawiły się w nim trzy nowe elementy: *nagłówek*, *komentarze* oraz *wcięcia*.

Nagłówek (**program** `Pole_kola`) pełni wyłącznie funkcję informacyjną i nie ma wpływu na działanie programu, co nie zmienia faktu, że używanie go — chociaż nieobowiązkowe — należy do dobrej praktyki, ułatwia bowiem czytanie osobom trzecim. Składnia nagłówka to

```
program nazwa;
```

Zauważ przy okazji, że „Pole koła” to dwa oddzielne wyrazy: ponieważ `nazwa` jako identyfikator nie może zawierać znaku spacji, zastępuje się go podkreśleniem (_).

Znacznie ważniejsze dla czytelności programu są komentarze. Komentarzem w Pascalu jest dowolny ciąg znaków ujęty w nawiasy klamrowe `{` i `}`. Z oczywistych względów ciąg taki nie może zawierać nawiasów klamrowych, natomiast wszystkie inne znaki są dozwolone. Jeżeli koniecznie chcesz umieścić nawiasy klamrowe w komentarzu, musisz użyć jako ograniczniki dwuznakowych symboli `(*` i `*)` (co widać w drugim komentarzu). Podobnie jak w przypadku nagłówka, komentowanie programów nie jest obowiązkowe, należy jednak do dobrej praktyki programowania. Oczywiście, komentarze powinny być używane rozsądnie: nie ma sensu ich nadużywanie czy komentowanie rzeczy oczywistych, jak np. słowa **`begin`**. Warto jednak zdawać sobie sprawę, że komentarze nic nie kosztują (tj. nie mają wpływu na wielkość skompilowanego programu), a jednocześnie w znaczący sposób poprawiają czytelność programu.



Komentarze bywają również często stosowane do chwilowego wyłączenia fragmentu programu z kompilacji. Pozwala to uniknąć usuwania i powtórnego wpisywania tekstu: zamiast tego wystarczy ująć „niechciany” fragment programu w znaki komentarza, które potem można łatwo usunąć.

Ostatnim „kosmetycznym” elementem naszego programu są *wcięcia*. Nasz program jest co prawda tak mały, że używanie wcięć jest nieco dyskusyjne, jednak w większych programach, którymi zajmiemy się już wkrótce, wcięcia pozwalają na szybkie zlokalizowanie fragmentów tworzących logiczną całość i znajdujących się na tym samym poziomie w strukturze programu. Wcinanie, podobnie jak inne reguły pisania ładnych i czytelnych programów, jest w dużej mierze kwestią stylu (zwykle pojedyncze wcięcie ma głębokość $1 \div 3$ znaków), warto jednak przyjrzeć się sprawdzonym wzorcom (np. programom przykładowym firmy Borland), a następnie wypracować własny styl i stosować go konsekwentnie.

Tyle spraw „formalnych”; nie będziemy się nad nimi więcej rozwodzić, zostawiając resztę praktyce. Przejdziemy obecnie do znacznie bardziej interesującej części tego rozdziału: co zrobić, żeby nasz program stał się bardziej praktyczny?

Może inaczej: co program robi obecnie? Oblicza i wyświetla pole koła o promieniu 5 cm. Ba, ale skąd wiadomo, że to pole koła, a nie np. średnia opadów z ubiegłego półrocza? Żeby się o tym przekonać, musisz zajrzeć do książki (gdzie jest to napisane wprost) lub do programu (nad którym musisz trochę pomyśleć, o ile nie wpisałeś jeszcze odpowiednich komentarzy). Na dodatek wyświetlana liczba ma cokolwiek niestrawny format i niezbyt chętnie nam się pokazuje...

Chwileczkę, nie wszystko naraz. Na początek spróbujmy sformułować listę życzeń, czyli zapisać w uporządkowany sposób to, co nam się w programie nie podoba i co chcielibyśmy zmienić. A zatem:

- program powinien informować użytkownika o tym, co robi;

- wynik działania programu nie powinien znikać, lecz zostawać na ekranie tak długo, jak życzy sobie użytkownik;
- wyświetlana liczba powinna ładniej wyglądać.

Pierwszy problem można rozwiązać wyprowadzając na ekran dodatkową informację w postaci tekstu. Służy do tego znana Ci już procedura `write`, która tym razem będzie wypisywała na ekranie nie wyrażenie liczbowe, lecz *łańcuch* (a ściślej rzecz biorąc, tzw. stałą łańcuchową), czyli ciąg znaków. Stałą łańcuchową jest w Pascalu dowolny ciąg znaków ujety w apostrofy (`'`, nie `"`). Ciąg taki może zawierać dowolne znaki; znak apostrofu (normalnie kończący łańcuch) zapisuje się jako dwa apostrofy (tj. dwa oddzielne znaki `'`, a nie znak cudzysłowu `"`). Całość będzie wyglądała następująco:

```
write('Program oblicza pole kola o promieniu 5 cm.')
```

Spróbuj teraz przyjrzeć się opisowi procedury `write` z poprzedniego rozdziału i zastanowić się, jak powinna wyglądać instrukcja wypisująca na ekranie tekst

```
Pole kola = .....
```

(w miejscu kropek powinna znaleźć się obliczona przez nasz program liczba). Jeśli nie wiesz, jak to zrobić, rzuć okiem na tekst następnego programu. Zauważ przy okazji, że pojawiła się w nim drobna nowinka, mianowicie procedura `writeln`, różniąca się od `write` tylko tym, że po jej wykonaniu kursor przechodzi do nowego wiersza (i tam też wypisywane są kolejne informacje). Aby przekonać się o różnicy w działaniu obu procedur, zamień pierwszą instrukcję `writeln` na `write` i wykonaj program.

Sprawa druga, czyli zatrzymanie wyniku na ekranie monitora, również daje się prosto rozwiązać. Wystarczy mianowicie na końcu programu dopisać instrukcję

```
readln;
```

i gotowe. Co to takiego `readln` — to już inna sprawa: procedurą `readln` i pokrewną jej `read` zajmiemy się wkrótce, na razie zaś wystarczy Ci wiedzieć tyle, że `readln` odczytuje z klawiatury dowolny ciąg znaków zakończony znakiem nowego wiersza (kodem klawisza ENTER), toteż jej wywołanie powoduje zatrzymanie działania programu do momentu naciśnięcia tegoż klawisza.

Pozostała sprawa ostatnia, tj. zrobienie porządku z mało czytelną reprezentacją wyprowadzanego pola. Na początek rozszyfrujmy zapis

```
7.8539816340E+01
```

Jest to nic innego, jak $7.8539816340 \cdot 10^1$, czyli około 78.5. Zapis `x.xxxxEnn` oznacza więc liczbę `x.xxxx` pomnożoną przez 10 do potęgi `nn` i zwany jest *zapisem naukowym*. Zauważ przy okazji, że część ułamkowa liczby oddzielona jest od części rzeczywistej nie przecinkiem, lecz kropką (bardzo ważne!). Wykładnik `nn` jest zawsze dobierany tak, by znajdująca się „przed nim” liczba miała tylko jedną cyfrę przed kropką, tj. zawierała się w przedziale od 1 do 10. Wielkość litery E symbolizującej wykładnik nie ma znaczenia, zaś znak + można pominąć. Oto kilka przykładów:

3.14E+00	=	3.14	1.2345e4	=	12345
2.78E–3	=	0.00278	1,35E12	=	**błąd!** (jaki?)

Zapis naukowy pozwala na łatwe i jednolite reprezentowanie bardzo małych i bardzo dużych liczb, jest jednak niezbyt czytelny. Aby uczynić go bardziej strawnym, można użyć tzw. *formatowania*, polegającego na przekazaniu procedurze `write` (`writeln`) dodatkowych informacji opisujących sposób wyprowadzenia liczby. W przypadku liczb rzeczywistych informacje te określają całkowitą szerokość pola zawierającego liczbę oraz ilość pozycji po kropce dziesiętnej i mogą być symbolicznie zapisane jako

```
wartość:szerokość-pola:liczba-cyfr-po-kropce
```

Liczba zawierająca mniej cyfr niż określono w szerokości pola jest „dosuwana” do prawej strony i uzupełniana z lewej znakami spacji. Ustalając szerokość pola musisz również uwzględnić miejsce na kropkę dziesiętną i ewentualny znak liczby.

W ten sposób dotarliśmy do kolejnej wersji programu obliczającego pole koła:

```
program Pole_kola;
{ Program oblicza pole koła o promieniu 5 }

begin
  writeln('Program oblicza pole kola');
  write('Pole = '; Pi*5*5:8:3);       (* oblicz i wypisz *)
                                      (* pole kola *)
  readln;
end.
```

Skoro już mówimy o formatowaniu, warto jeszcze wspomnieć, że dla liczb całkowitych, nie posiadających części ułamkowej, wystarczy określenie szerokości pola, czyli zapis

```
wartość:szerokość-pola
```

W identyczny sposób można również formatować np. łańcuchy, co jednak stosowane jest rzadko. Za przykład niech posłuży kolejny program:

```
program Formatowanie;
{ Demonstracja możliwości formatowania wydruków }

begin
  writeln(1.234567:10:1); { za mało miejsca po kropce }
  writeln(1234567:3:0);   { za mało miejsca w ogóle }
  writeln(1.23:10);       { określamy tylko szerokość pola }
  writeln(123:10);        { formatowanie liczby całkowitej }
  writeln('!':20)         { formatowanie łańcucha }
end.
```

W ten sposób omówiliśmy podstawowe zabiegi kosmetyczne, którym możemy poddać nasz program. Prezentuje się on już nieco ładniej, jednak zmiany, których dokonaliśmy, mają charakter powierzchowny. Co bowiem zrobisz, żeby obliczyć pole koła o promieniu 3 cm? Poprawisz program i skompilujesz go jeszcze raz? Chyba jednak prościej wziąć kalkulator... a może nie...

Zapamiętaj

- Programy należy pisać czytelnie, gdyż ułatwia to ich zrozumienie i ewentualne wprowadzanie późniejszych zmian.
- Program można opisać nagłówkiem, zaczynającym się od słowa kluczowego **`program`**.
- Bardzo ważne dla czytelności programu są komentarze, zapisywane w nawiasach klamrowych, oraz odpowiednie wcinanie poszczególnych instrukcji.
- Aby wyprowadzać informacje w kolejno następujących po sobie wierszach, powinieneś użyć procedury `writeln`.
- Chcąc wstrzymać działanie programu, możesz użyć procedury `readln`.
- Aby uczynić wydruk bardziej czytelnym, zastosuj formatowanie wyjścia.

Ulepszamy dalej

Nie trzeba chyba nikogo przekonywać, że porządny program powinien umieć sam zapytać użytkownika o wartość promienia i obliczyć pole bez konieczności uciążliwego poprawiania i kompilowania tekstu. W tym celu należałoby wypisać na ekranie odpowiednie pytanie (co już potrafisz zrobić), a następnie wprowadzić z klawiatury wartość promienia i wykorzystać ją do obliczenia pola koła ze wzoru $S = \pi r^2$. Ba, łatwo powiedzieć: ale jak wprowadzić jakąś wartość do programu i gdzie ją przechować?

Rozwiązaniem tego problemu jest użycie *zmiennej*. Podobnie jak w matematyce, zmienna symbolizuje pewną wartość, czyli może być traktowana jako swego rodzaju pojemnik, do którego wkładamy np. liczbę (wartość promienia). „Fizycznie" wartość zmiennej umieszczana jest gdzieś w pamięci komputera i może być zlokalizowana poprzez *adres*, czyli numer odpowiedniej komórki pamięci. Ponieważ posługiwanie się adresami jest niezbyt wygodne, Pascal umożliwia odwoływanie się do zmiennej za pomocą *nazwy*, będącej zwykłym identyfikatorem (informacje na temat identyfikatorów znajdziesz na stronie 27). Dodatkową informacją, niezbędną do ustalenia sposobu posługiwania się zmienną i jej reprezentacji w pamięci, jest *typ* — w naszym przypadku będzie to typ zmiennoprzecinkowy, odpowiadający wartości rzeczywistej (`real`).

Ponieważ kompilator tłumaczy tekst źródłowy programu na bezpośrednio wykonywalną przez komputer postać wynikową, musi wiedzieć jak reprezentować daną zmienną w pamięci i jak się z nią obchodzić. Niezbędne jest zatem *zadeklarowanie* każdej używanej w programie zmiennej, czyli przekazanie kompilatorowi informacji o jej nazwie i typie. Do deklarowania zmiennych służy specjalne słowo kluczowe **`var`** (ang. ***var**iable* — zmienna), zaś składnia deklaracji ma postać

```
var
  lista-nazw : typ-1;
  lista-nazw : typ-2 ...
```

przy czym *lista-nazw* to ciąg nazw zmiennych tego samego typu rozdzielonych przecinkami. Aby zadeklarować kolejną grupę zmiennych (innego typu), nie musisz ponownie używać słowa **`var`**; wystarczy dopisać pod spodem nową listę i podać nazwę typu.

Ostatnim wymaganiem, o którym musisz pamiętać, jest umieszczanie wszystkich deklaracji przed częścią operacyjną programu: ponieważ bezpośrednio wykorzystuje ona zadeklarowane obiekty, muszą one zostać opisane przed jej rozpoczęciem.

Po zadeklarowaniu można już „normalnie" korzystać ze zmiennej, czyli nadawać jej wartość, wykorzystywać w wyrażeniach lub wyprowadzać na ekran monitora. Musisz jednak pamiętać, że sama deklaracja nie nadaje zmiennej sensownej wartości, a jedynie informuje kompilator o jej istnieniu i właściwościach. Przed użyciem zmiennej musisz **koniecznie** nadać jej wartość (*zainicjalizować* ją) przez wprowadzenie jej wartości z klawiatury lub za pomocą tzw. *instrukcji przypisania*.

Żeby dłużej nie teoretyzować, zademonstrujmy nową wersję naszego programu:

```
program Pole_kola;
{ Program oblicza pole koła o promieniu 5 }
{ Promień wprowadzany jest z klawiatury }

var  { deklaracja zmiennych }
  r : real;  { promień koła - zmienna rzeczywista }

begin
  writeln('Program oblicza pole kola');
  write('Podaj promien kola: ');
  readln(r);                    { odczytaj zmienną }
                                { z klawiatury }
  write('Pole = '; Pi*r*r:8:3); { oblicz i wypisz }
                                { pole koła }
  readln;
end.
```

Zapewne zauważyłeś już w programie znaną Ci z poprzedniego rozdziału procedurę `readln`. Tutaj pełni ona inną (właściwie swoją zasadniczą) funkcję, a mianowicie odczytuje wartość zmiennej *r* z klawiatury. Składnia procedury `readln` jest następująca:

```
readln[(zmienna, zmienna...)];
```

i wygląda podobnie, jak dla „odwrotnej" procedury `writeln`, z tą drobną różnicą, że argumentami `readln` mogą być jedynie zmienne. Dlaczego? Procedura `readln` pobiera z klawiatury (a ściślej rzecz biorąc, z tzw. *wejścia* programu) pewną wartość i musi ją gdzieś umieścić. Wartość może zostać umieszczona w zmiennej, czyli gdzieś w pamięci, nie może jednak zostać zapisana w wyrażeniu, które jest jedynie zapisem pewnej operacji i nie posiada lokalizacji w pamięci komputera. Podobnie niedopuszczalne będzie użycie jako argumentu stałej lub (o zgrozo!) operatora.

Pokrewna procedura `read` stosowana jest znacznie rzadziej; różnica między nią a `readln` sprowadza się (podobnie jak dla `write` i `writeln`) do tego, iż po odczytaniu swoich argumentów `readln` przechodzi do następnego wiersza danych wejściowych, a `read` nie. Procedura `read` używana jest głównie do czytania danych z plików (o których później), zaś jej użycie do wprowadzania danych z klawiatury może doprowadzić do niespodziewanych efektów, dlatego też na razie lepiej jej unikać.

Pora powiedzieć coś więcej na temat typów. Typ zmiennej określa jej wewnętrzną reprezentację w pamięci komputera (liczbę bajtów zajmowanych przez zmienną, zakres dopuszczalnych wartości i dokładność reprezentacji) oraz zestaw operacji, które można na niej wykonywać. Kilkanaście standardowych typów dostępnych w Turbo Pascalu opiszemy w poniższej tabeli. Z wyjątkiem typu **`string`**, wszystkie one należą do tzw. *typów prostych* (nie dających się rozbić na prostsze typy).

Tablica 1. *Najważniejsze typy dostępne w Turbo Pascalu*

Nazwa	*Znaczenie*	*Zakres wartości*	*Przykład*
`integer`	liczba całkowita ze znakiem	–32768..+32767	–14574
`real`	liczba rzeczywista	$2.9\cdot 10^{-39}..1.7\cdot 10^{38}$	1.23245e17
`char`	znak	znaki o kodach 0..255	'a'
`string`	napis (łańcuch, ciąg znaków)	ciąg do 255 znaków	'Napis'
`boolean`	wartość logiczna	prawda (*true*) lub fałsz (*false*)	*false*
`word`	s³owo	0..65535	56412
`byte`	bajt	0..255	127
`shortint`	krótka liczba całkowita ze znakiem	–128..+127	–13
`longint`	długa liczba całkowita ze znakiem	–2147483648..+2147483647	–1986734234
`single`	krótka liczba rzeczywista	$1.5\cdot 10^{-45}..3.4\cdot 10^{37}$	3.14e01
`double`	długa liczba rzeczywista	$5.0\cdot 10^{-324}..1.7\cdot 10^{308}$	–1.8e+234
`extended`	bardzo długa liczba rzeczywista	$3.4\cdot 10^{-4932}..1.1\cdot 10^{4932}$	4.5e2345
`comp`	bardzo długa liczba całkowita	$-9.2\cdot 10^{18}..9.2\cdot 10^{18}$	6e12
`pointer`	wskaźnik	0000h:0000h..FFFFh:FFFFh	$1234:ABCD

Cztery pierwsze typy wymienione w tabeli stanowią podstawowy zestaw, którym będziesz posługiwał się podczas pisania programów. Typy całkowite (`integer`, `word`, `shortint`, `byte`, `longint`), typ znakowy (`char`) oraz logiczny (`boolean`) tworzą z kolei grupę tzw. *typów porządkowych* (ang. *ordinal types*), w ramach których określona jest relacja porządku (czyli kto przed kim, a kto po kim).

Typy całkowite używane są w programach do reprezentacji liczników, adresów i indeksów tablic oraz zawartości komórek pamięci (`byte` i `word`). Pamiętaj, że ze względu na specyficzną reprezentację wewnętrzną liczb całkowitych przekroczenie zakresu dopuszczalnego dla danego typu powoduje „przeskoczenie” na drugi koniec zakresu (np. dodanie 1 do 32767 da w wyniku –32768). Efekt ten zwykle nie jest sygnalizowany przez kompilator i może prowadzić do dość dziwacznego zachowania programów. Aby go uniknąć, możesz zastosować pojemniejszy typ (np. `longint`).

Drugą obszerną grupę tworzą typy rzeczywiste, przeznaczone do przechowywania wartości niecałkowitych i realizacji większości bardziej złożonych obliczeń matematycznych. W większości przypadków zadowalające rezultaty osiąga się przy użyciu liczb typu `real`, jednak dla poprawy dokładności, powiększenia zakresu wartości i przyspieszenia obliczeń warto odwołać się do jednego z pozostałych typów. Ich wykorzystanie wymaga zadeklarowania użycia w programie tzw. *koprocesora arytmetycznego* (specjalnego układu przeznaczonego do operowania na liczbach rzeczy-

wistych, dostępnego jako oddzielny układ lub wbudowanego w procesory 486DX i „lepsze”) albo tzw. *emulatora*, czyli specjalnego podprogramu „udającego” koprocesor. Aby było możliwe odwołanie się do koprocesora, należy włączyć opcję ***Numeric Processing — 8087/80287*** w polu dialogowym ***Options-Compiler***, zaś włączenie emulacji realizuje znajdująca się poniżej opcja ***Emulation***. Alternatywą jest umieszczenie w programie tzw. *dyrektyw kompilatora*, mających odpowiednio postać `{$N+}` i `{$E+}`.

Typy znakowy i łańcuchowy wykorzystywane są do reprezentowania napisów i wszelkiej informacji „słownej”. Typ `boolean` pozwala na przechowywanie wartości logicznych, wykorzystywanych do sterowania działaniem programu, zaś typ wskaźnikowy (`pointer`) używany jest w operacjach na zawartości pamięci komputera (nieco dziwny zapis $1234:ABCD oznacza adres komórki pamięci w postaci szesnastkowej, sygnalizowanej znakiem $).

Na tym zakończymy ten krótki rozdział. Na kolejnych kilku stronach zajmiemy się wyrażeniami i ich elementami składowymi, zaś w charakterze ćwiczenia proponuję Ci zmodyfikowanie naszego programu tak, by obliczał on pole trójkąta na podstawie wprowadzonej z klawiatury długości podstawy i wysokości.

Zapamiętaj

- Do przechowywania wartości służą w programie zmienne.
- Cechami charakterystycznymi zmiennej są nazwa i typ. Nazwa pozwala na zidentyfikowanie zmiennej, zaś typ określa jej wewnętrzną reprezentację i zakres wartości oraz dopuszczalnych operacji.
- Przed użyciem zmienna musi zostać zadeklarowana (za pomocą słowa kluczowego **`var`**) oraz zaincjalizowana (przez przypisanie lub wprowadzenie wartości z zewnątrz, np. z klawiatury).
- Do wprowadzania wartości zmiennych z klawiatury służą procedury `readln` i `read`.
- Turbo Pascal oferuje kilkanaście standardowych typów, spośród których najczęściej stosowanymi są typ całkowity (`integer`), rzeczywisty (`real`), znakowy (`char`) i łańcuchowy (`string`).
- Typy całkowitoliczbowe, znakowe oraz typ `boolean` tworzą grupę typów porządkowych.

Wyrażenia

Temat wyrażenia został „przemycony” do książki już dość dawno temu i wypadałoby w końcu tę kwestię wyjaśnić. W niniejszym rozdziale spróbujemy wyjaśnić, czym są wyrażenia, do czego służą oraz jakie są ich składniki i reguły tworzenia.

Wyrażenia pozwalają na przekształcanie informacji w celu uzyskania odpowiednich wyników i stanowią jeden z podstawowych składników programów. Każde wyrażenie stanowi symboliczny zapis pewnej operacji na danych reprezentowanych przez zmienne (opisane identyfikatorami) i stałe (zapisane jawnie). Sama operacja realizowana jest za pomocą *operatorów* oraz *funkcji*. Pojęcie wyrażenia najłatwiej będzie nam zilustrować na przykładzie matematycznym: jak wiadomo, długość przeciwprostokątnej c trójkąta prostokątnego wyraża się wzorem

$$c = \sqrt{a^2 + b^2}$$

Przekładając to na Pascal otrzymamy

```
c := sqrt(a*a + b*b)
```

Zapis znajdujący się po prawej stronie znaku `:=` (tak zwanego *operatora przypisania*) jest właśnie wyrażeniem. W jego skład wchodzą cztery identyfikatory (`a` i `b`) symbolizujące zmienne przechowujące długości przyprostokątnych, trzy operatory (`*` i `+`) symbolizujące operacje mnożenia i dodawania oraz identyfikator `sqrt` reprezentujący funkcję — pierwiastek kwadratowy. Ponieważ o zmiennych i stałych już mówiliśmy, zajmiemy się obecnie operatorami.

Operator jest zastrzeżonym słowem języka, stanowiącym symboliczną reprezentację pewnego działania na danych reprezentowanych przez *argumenty* operatora. W większości przypadków operatory posiadają dwa argumenty, istnieją jednak również operatory jednoargumentowe. W Pascalu zdefiniowano kilka grup operatorów, z których najczęściej wykorzystywanymi są zapewne *operatory arytmetyczne*:

Tablica 2. *Operatory arytmetyczne*

Operator	*Znaczenie*	*Przykład*
`*`	mnożenie	2`*`2 = 4
`/`	dzielenie	2`/`3 = 0.66666...
`div`	dzielenie całkowite	2 `div` 3 = 0
`mod`	reszta z dzielenia	3 `mod` 2 = 1
`+`	dodawanie	2 `+` 3 = 5
`-`	odejmowanie	2 `–` 3 = –1
`–` (jednoargumentowy)	zmiana znaku	`–` 1 = –1

Z wyjątkiem operatorów **`div`** i **`mod`**, przeznaczonych wyłącznie do działań na liczbach całkowitych, wszystkie pozostałe operatory „współpracują” zarówno z liczbami całkowitymi, jak i rzeczywistymi. W przypadku wykonywania kilku działań w obrębie jednego wyrażenia istotny jest tzw. *priorytet operatorów*, określający pierwszeństwo pewnych działań przed innymi. Dla operatorów arytmetycznych priorytety wyglądają mniej więcej tak, jak w „zwykłej” matematyce: pierwszeństwo mają operatory mnożenia i dzielenia (`*`, `/`, **`div`** i **`mod`**), wykonywane zawsze przed dodawaniem i odejmowaniem. Działania reprezentowane przez operatory o tym samym priorytecie wykonywane są w kolejności od lewej do prawej.

Pamiętanie o priorytetach operatorów jest sprawą bardzo istotną, gdyż niewłaściwa kolejność wykonywania działań prowadzi często do uzyskania zupełnie innego wyniku, niż się spodziewaliśmy. Przykładowo, iloraz

$$\frac{1+2}{3}$$

nie zostanie poprawnie obliczony, jeśli zapiszesz go w postaci

```
1 + 2 / 3
```

bowiem zamiast wartości 1 otrzymasz 1 plus 2/3, czyli 1.6666... . Właściwą kolejność wykonywania działań możesz jednak wymusić za pomocą nawiasów, zapisując nasz iloraz jako

```
(1 + 2) / 3
```

Fragmenty wyrażenia ujęte w nawiasy są zawsze obliczane przed wykonaniem wszystkich pozostałych działań. Nawiasów warto (a nawet należy) używać również wówczas, gdy nie jest się pewnym co do kolejności wykonywania działań. Pamiętaj, że nawiasy nie powodują generowania dodatkowego kodu wynikowego, a jedynie zmieniają kolejność operacji, a więc „nic nie kosztują” (oprócz konieczności wpisania kilku dodatkowych znaków, co jednak jest lepsze od narażania się na trudno wykrywalne błędy).

Operatory arytmetyczne nie wyczerpują oczywiście arsenału dostępnego w Turbo Pascalu. Kolejną grupę tworzą *operatory bitowe i logiczne*, nazwane tak dlatego, iż

przeznaczone są do wykonywania działań na bitach liczb całkowitych lub do przekształcania wartości logicznych. A oto one:

Tablica 3. *Operatory bitowe i logiczne*

Operator	*Znaczenie*	*Przykład*	
		logiczny	*bitowy*
`not`	negacja	`not` *true = false*	`not` 15 = 240
`and`	iloczyn logiczny	true and false = false	7 `and` 15 = 7
`shl`	przesunięcie bitów w lewo		7 `shl` 2 = 28
`shr`	przesunięcie bitów w prawo		128 `shr` 4 = 8
`or`	suma logiczna	*true* `or` *false = true*	7 `or` 128 = 135
`xor`	suma modulo 2	*true* `xor` *true = false*	7 `xor` 15 = 8

Zrozumienie działania operatorów logicznych nie powinno nastręczać trudności. Jeśli chodzi o operatory bitowe, to ich działanie sprowadza się do manipulowania poszczególnymi bitami (mogącymi przyjmować wartości 0 lub 1) w komórkach pamięci zajmowanych przez liczbę. Jednoargumentowy operator `not` neguje poszczególne bity (zmienia ich wartości na przeciwne), operator `and` ustawia dany bit wyniku na 1 tylko wtedy, gdy odpowiadające sobie bity obu argumentów mają wartość 1, `or` — gdy co najmniej jeden z nich ma wartość 1, zaś `xor` ustawia bit na 1 tylko wtedy, gdy jeden z bitów ma wartość 1 a drugi 0. Operatory `shl` i `shr` mają charakter wyłącznie bitowy; dla liczb całkowitych (w pewnym uproszczeniu) przesunięcie bitów o *n* pozycji w lewo lub w prawo odpowiada pomnożeniu lub podzieleniu przez 2^n. W ramach ćwiczeń proponuję Ci sprawdzić poprawność podanych wyżej przykładów (potrzebna Ci będzie znajomość systemu dwójkowego, w którym np. całkowita liczba 7 zapisywana jest jako 00000111, czyli sekwencja ośmiu bitów, z których trzy najmniej znaczące mają wartość 1).

Ostatnią ważną grupę operatorów stanowią *operatory relacyjne*, służące do porównywania obiektów (nie tylko liczb, ale również znaków czy łańcuchów). Wszystkie operatory relacyjne są dwuargumentowe (oczywiście typy obu argumentów muszą być zgodne, tj. nie można porównywać łańcucha z liczbą) i dają w wyniku wartość logiczną.

Tablica 4. *Operatory relacyjne*

Operator	*Znaczenie*	*Przykład*
=	równy...	3 = 3.14 (*false*)
<>	różny od...	3 <> 3.14 (*true*)
<	mniejszy od...	3 < 3.14 (*true*)
<=	mniejszy lub równy...	3 <= 3.14 (*false*)
>	większy od...	3 > 3.14 (*false*)
>=	większy lub równy	3 >= 3 (*true*)

Musimy jeszcze ustosunkować się do wyrażeń „mieszanych”, zawierających operatory należące do kilku grup. Przykładowo, obliczając wysokość stypendium studenckiego należy określić, czy średnia ocen delikwenta jest wystarczająco wysoka, a dochód na jednego członka rodziny wystarczająco niski. Aby zatem ustalić, czy Kowalskiemu należy się stypendium, użyjemy przykładowego wyrażenia

```
(Dochod/LiczbaOsob < 150) and (Srednia > 3.75)
```

Wyrażenie to zawiera doskonałą mieszankę operatorów arytmetycznych, logicznych i relacyjnych. W takich sytuacjach kolejność wykonywania działań jest następująca:

Tablica 5. *Priorytety operatorów*

Operatory	*Priorytet*
`not`	1 (najwyższy)
`* / div mod and shl shr`	2 (niższy)
`+ - or xor`	3 (jeszcze niższy)
`= <> < <= > >=`	4 (najniższy)

Oczywiście, jeśli wyrażenie zawiera nawiasy, ich zawartość zostanie wyliczona przed wykonaniem pozostałych działań. Pokazany wyżej przykład ilustruje jednocześnie dość typową sytuację, w której musimy połączyć ze sobą kilka warunków (wyrażonych nierównościami, czyli operatorami relacyjnymi). Ponieważ do łączenia warunków służą operatory logiczne (zwykle **`or`** lub **`and`**), mające wyższy priorytet, niezbędne jest użycie nawiasów (ich pominięcie jest bardzo pospolitym błędem; o reakcji kompilatora najlepiej przekonaj się sam).

Jak już powiedzieliśmy, wyrażenia mogą zawierać elementy najróżniejszych typów (liczby, znaki, łańcuchy itp.), jednak najczęściej będziesz spotykał się z wyrażeniami reprezentującymi operacje arytmetyczne. Aby rozdział ten nie ograniczał się do pustego teroretyzowania, spróbujemy zastosować wyrażenia w praktyce, tworząc program rozwiązujący równanie kwadratowe znaną Ci zapewne metodą wyznaczników.

Zapisując nasze równanie w postaci

$$a \cdot x^2 + b \cdot x + c$$

możemy obliczyć pierwiastki (tj. wartości *x*, dla których równanie przyjmuje wartość zero) jako

$$\frac{-b \pm \sqrt{\Delta}}{2a}$$

gdzie * jest tzw. *wyznacznikiem* równania, obliczanym jako

$$\Delta = b^2 - 4 \cdot a \cdot c$$

Znak • w poprzednim wzorze oznacza, że równanie ma w ogólności dwa pierwiastki, z których jeden oblicza się przez dodanie wartości w liczniku ułamka, zaś drugi — przez ich odjęcie.

Sam program powinien wyglądać następująco:

początek
odczytaj wartości współczynników a, b i c,
oblicz wartość wyznacznika
oblicz pierwiastki i wypisz je na ekranie
koniec

W postaci pascalowej będzie to nieco bardziej skomplikowane:

```
program Rownanie_Kwadratowe;
{ Program rozwiązuje równanie kwadratowe metodą wyznaczników }

var
  a, b, c : real;         { współczynniki }
  delta : real;           { wyznacznik }
  x1, x2 : real;          { pierwiastki }

begin
  writeln('Program rozwiazuje rownanie kwadratowe')
  writeln('a*x^2 + b*x + c');
  write('Podaj wspolczynnik a: '); { wprowadź współczynniki }
  readln(a);
  write('Podaj wspolczynnik b: ');
  readln(b);
  write('Podaj wspolczynnik c: ');
  readln(c);
  delta := sqr(b) - 4*a*c;          { oblicz wyznacznik }
  x1 := (-b + sqrt(delta))/(2*a);   { oblicz pierwiastki }
  x2 := (-b - sqrt(delta))/(2*a);   { znaczek := to tzw. }
                                    { przypisanie }
  writeln('Pierwiastki:');          { wyświetl pierwiastki }
  writeln('x1 = ', x1:12:4);
  writeln('x2 = ', x2:12:4);
  readln;
end.
```

Wyrażenia wykorzystywane do obliczania poszczególnych wartości są tu nieco bardziej skomplikowane, ale nadal nie powinieneś mieć problemów z ich analizą. Dwoma nowymi (no, nie do końca) elementami są *operator przypisania* oraz *funkcje*.

Operator przypisania jest jednym z najpospoliciej wykorzystywanych operatorów pascalowych. Znaczek **:=** (dwukropek i znak równości) czyta się „staje się” lub „przypisz”, zaś jego efekt sprowadza się do umieszczenia w obiekcie znajdującym się po lewej stronie wartości znajdującej się po prawej stronie. „Obiekt” umieszczony po lewej stronie operatora przypisania określany jest mianem *l*-wartości (ang. *lvalue*). Jest to na ogół zmienna, możliwe jest jednak przypisanie wartości do identyfikatora funkcji (o tym będziemy jeszcze mówić). Ponieważ przypisanie polega na umieszczeniu

wartości gdzieś w pamięci komputera (w miejscu określonym przez nazwę obiektu), **nie możesz** po lewej stronie operatora przypisania umieścić stałej czy wyrażenia. Po prawej stronie operatora przypisania panuje znacznie większa demokracja: dopuszczalne są tam praktycznie dowolne wyrażenia (a więc w szczególności również pojedyncze stałe i zmienne), byleby typ wartości uzyskanej w wyniku obliczenia wyrażenia był zgodny z typem obiektu znajdującego się po drugiej stronie operatora `:=`.

Zagadnienie zgodności typów w sensie przypisania (ang. *assignment compatibility*) jest tyleż istotne, co złożone. Z dość dobrym przybliżeniem można powiedzieć, że typ wyrażenia znajdującego się po prawej stronie operatora `:=` powinien być identyczny z typem obiektu znajdującego się po lewej stronie, z kilkoma wyjątkami. Do najważniejszych wyjątków należą przypisania:

- wartości typu całkowitego do obiektu typu rzeczywistego (ale nie odwrotnie);
- wartości typu rzeczywistego do obiektu innego typu rzeczywistego (w dowolną stronę);
- wartości typu znakowego do obiektu typu łańcuchowego;
- wartości typu „wskaźnik do typu” do obiektu typu `pointer`.

Przy tej okazji warto również wspomnieć, że niektóre przypisania pomiędzy typami rzeczywistymi mogą spowodować utratę dokładności lub przekroczenie zakresu (np. próba „wciśnięcia” wartości typu `extended` do zmiennej typu `single`).

Jeśli idzie o drugą nowość, czyli funkcje, to w naszym programie pojawiły się wywołania dwóch tzw. bibliotecznych funkcji arytmetycznych: `sqr` i `sqrt`. Temat funkcji będzie dyskutowany nieco później; na razie wystarczy Ci wiedzieć, że funkcja (i pokrewna jej procedura) jest pewnym narzędziem pozwalającym na przekształcenie podanych jej wartości (tzw. *argumentów*) do żądanej postaci. Praktycznie każdy język programowania dysponuje mniejszym lub większym zestawem (biblioteką) funkcji realizujących operacje arytmetyczne, znakowe, graficzne i inne. Biblioteka funkcji i procedur dostępnych w Turbo Pascalu jest ogromna i omawianie jej mijałoby się z celem; wśród często wykorzystywanych funkcji arytmetycznych, oprócz nierzadko mylonych ze sobą `sqr` (podniesienie do kwadratu) i `sqrt` (pierwiastek kwadratowy), znajdują się również funkcje trygonometryczne (`sin`, `cos`, `arctan`), logarytmiczne (`ln`, `exp`) i wiele innych.

W ten sposób omówiliśmy zasadnicze tematy związane z wyrażeniami i realizacją obliczeń arytmetycznych. Wykorzystując poznane wiadomości możesz pokusić się o napisanie programu obliczającego wartości bardziej skomplikowanych wzorów (jakich? to już pewnie sam wiesz najlepiej). Niestety, o czym być może już się przekonałeś, Turbo Pascal nie jest na tyle inteligentny, by obronić się przed próbą wykonania operacji nielegalnej z matematycznego punktu widzenia (np. dzielenia przez zero). Jeśli jeszcze nie miałeś okazji potknąć się o ten problem, spróbuj za pomocą naszego programiku znaleźć miejsca zerowe funkcji

$$x^2 + 2x + 3$$

Nasze równanie nie ma pierwiastków rzeczywistych: wartość wyznacznika wynosi –8, a zatem jego pierwiastek kwadratowy nie daje się obliczyć. Ponieważ jednak Turbo Pascal o tym nie wie, efektem podjętej próby jest błąd wykonania (*Invalid floating point operation* — nielegalna operacja zmiennoprzecinkowa) i przerwanie realizacji programu. Co prawda żadna to tragedia, bo funkcja i tak nie ma pierwiastków, ale program mógłby zachowywać się przyzwoiciej, tj. zamiast „padać" — wyświetlać odpowiednią informację.

Nie ma problemu... ale to już może w następnym rozdziale.

Zapamiętaj

- Do przekształcania informacji w programie służą wyrażenia.
- W skład wyrażeń wchodzą stałe i zmienne (reprezentujące dane) oraz operatory i funkcje (reprezentujące sposoby przekształcania informacji).
- Najważniejsze grupy operatorów pascalowych to operatory arytmetyczne, bitowe, logiczne oraz relacyjne (operatory porównania).
- Konstruując wyrażenia musisz pamiętać o kolejności wykonywania działań, czyli priorytetach operatorów.
- Do zmiany kolejności działań służą nawiasy.
- Funkcje reprezentują bardziej złożony sposób przekształcania informacji niż operatory.
- Do nadawania wartości zmiennym służy operator przypisania `:=`.
- Po lewej stronie operatora przypisania musi znajdować się *l*-wartość. Po jego prawej stronie może znajdować się dowolne wyrażenie.

Instrukcja warunkowa

Aby uniknąć błędu wynikającego z próby pierwiastkowania liczby ujemnej, wypadałoby umieścić w naszym programie zastrzeżenie w postaci „jeżeli wyznacznik jest większy od zera to wykonaj dalsze obliczenia, w przeciwnym przypadku wypisz odpowiedni komunikat”. Przekładając to na język angielski i nieco uogólniając, otrzymujemy strukturę *instrukcji warunkowej*:

```
if warunek
then instrukcja-1
else instrukcja-2
```

W powyższym zapisie `warunek` jest niczym innym, jak wyrażeniem dającym w wyniku wartość *true* lub *false*, na ogół wykorzystującym znane Ci już operatory relacyjne (mówiąc po ludzku, jest to zwykłe porównanie). `Instrukcje` opisują czynności, które powinny być wykonane po spełnieniu lub niespełnieniu warunku, przy czym część zaczynająca się od słowa kluczowego **`else`** nie jest obowiązkowa (tj. nie musisz określać „wariantu awaryjnego”).

Instrukcja warunkowa, dostępna praktycznie w każdym języku programowania, stanowi podstawowe narzędzie umożliwiające podejmowanie decyzji w zależności od aktualnych warunków, czyli wbudowywanie w program „inteligencji”, pozwalającej mu zachować się odpowiednio do sytuacji. Spróbujmy wkomponować instrukcję warunkową w nasz poprzedni program:

```
program Nowe_Rownanie_Kwadratowe;
{ Program rozwiązuje równanie kwadratowe metodą wyznaczników }
{ uwzględniając możliwość ujemnego znaku wyznacznika }

var
  a, b, c : real;       { współczynniki }
  delta : real;         { wyznacznik }
  x1, x2 : real;        { pierwiastki }

begin
  writeln('Program rozwiazuje rownanie kwadratowe')
  writeln('a*x^2 + b*x + c');
  write('Podaj wspolczynnik a: '); { wprowadź współczynniki }
  readln(a);
```

```
    write('Podaj wspolczynnik b: ');
    readln(b);
    write('Podaj wspolczynnik c: ');
    readln(c);
    delta := sqr(b) - 4*a*c;          { oblicz wyznacznik }
    if delta >= 0 then                { wyznacznik OK }
      begin                           { wykonujemy obliczenia}
        x1 := (-b + sqrt(delta))/(2*a); { oblicz pierwiastki }
        x2 := (-b - sqrt(delta))/(2*a);
        writeln('Pierwiastki:');      { wyświetl pierwiastki }
        writeln('x1 = ', x1:12:4);
        writeln('x2 = ', x2:12:4);
      end                             { tu NIE MA średnika! }
    else                              { wyznacznik ujemny }
      writeln('Rownanie nie ma pierwiastkow rzeczywistych.');
    readln;
  end.
```

Postać warunku jest w naszym przypadku oczywista:

```
if delta >=0
```

czyli „jeżeli wyznacznik nie jest mniejszy od zera...”. Zauważ, że jeśli warunek jest spełniony, należy wykonać więcej niż jedną instrukcję, a zatem cała grupa instrukcji zajmujących się obliczaniem i wyświetlaniem pierwiastków musi zostać otoczona słowami kluczowymi **begin** i **end**, oznaczającymi tzw. *instrukcję złożoną*, czyli zestaw instrukcji wykonywanych jako jedna całość. Zapominanie o tym wymaganiu jest jednym z pospolitszych błędów popełnianych przez programistów pascalowych (nie tylko początkujących) i może prowadzić do niezbyt przyjemnych błędów wykonania. Oto przykład:

```
if n <> 0 then
  LiczbaProbek := n;
  srednia := suma/LiczbaProbek;
```

Powyższy fragment powinien przypisywać zmiennej `LiczbaProbek` wartość `n`, a następnie obliczać `srednia` jako `suma` podzielona przez `LiczbaProbek`, gwarantując, że w przypadku `n` równego zero dzielenie nie będzie wykonywane. Ponieważ jednak przypisania nie zostały połączone w instrukcję złożoną, dla `n` równego zeru pierwsze przypisanie nie wykona się, a zatem zmienna `LiczbaProbek` będzie miała wartość nieokreśloną. Może to być zero (wówczas wykonanie programu zostanie przerwane) lub dowolna wartość, co oczywiście doprowadzi do zupełnie bezsensownego wyniku. Prawidłowa postać powyższej instrukcji będzie wyglądała następująco:

```
if n <> 0 then
  begin
    LiczbaProbek := n;
    srednia := suma/LiczbaProbek;
  end;
```

Wróćmy do naszego równania. Zauważ, że w pewnym szczególnym przypadku — gdy wyznacznik jest równy zeru — ma ono tylko jeden pierwiastek: jest tak dla równania

$$x^2 + 2x + 1$$

którego podwójnym pierwiastkiem jest liczba –1. W tym przypadku musimy uwzględnić trzy możliwości, w zależności od wartości wyznacznika:

Wyznacznik	*Pierwiastki*
większy od zera	$\frac{-b+\sqrt{\Delta}}{2a}$ i $\frac{-b-\sqrt{\Delta}}{2a}$
równy zeru	$-\frac{b}{2a}$
mniejszy od zera	brak pierwiastków rzeczywistych

Ponieważ pojedyncza instrukcja warunkowa potrafi obsłużyć tylko dwie wzajemnie się wykluczające sytuacje, aby uwzględnić wszystkie warianty, niezbędne jest *zagnieżdżenie* dwóch takich instrukcji. Schemat postępowania pokazany jest poniżej; w ramach ćwiczeń proponuję Ci samodzielne dokonanie odpowiednich modyfikacji w programie.

```
if delta < 0 then
  { brak pierwiastków rzeczywistych }
else
  if delta = 0 then
    { oblicz i wypisz pojedynczy pierwiastek }
  else { odnosi się do drugiego if-a }
    { oblicz i wypisz obydwa pierwiastki }
```

W przypadku większej liczby wariantów takie „piętrowe" konstrukcje stają się nieporęczne, trudne w interpretacji i podatne na błędy. Zastanów się na przykład, jak wyglądałaby instrukcja warunkowa wypisująca na ekranie nazwę dnia tygodnia danego numerem (niech 1 oznacza poniedziałek). Może tak:

```
if dzien = 1 then writeln('Poniedzialek')
else if dzien = 2 then writeln('Wtorek')
else if ...
```

Co prawda używanie pełnej formy instrukcji warunkowej (**`if-then-else`**) nie jest tu konieczne i słowa **`else`** można usunąć (pamiętając o średnikach), jednak w dalszym ciągu powyższy twór będzie co najmniej nieelegancki. Skutecznym sposobem obsługi wyborów wielowariantowych jest *instrukcja wyboru* **`case`**, odpowiadająca z grubsza „stercie" zagnieżdżonych instrukcji warunkowych:

```
case przełącznik of
  wartość-1 : akcja-1;
  wartość-2 : akcja-2;
  ...
  else akcja-domyślna
end;
```

Powyższa struktura jest swoistą tablicą akcji odpowiadających poszczególnym wartościom zmiennej `przełącznik` i doskonale nadaje się do realizacji wielowariantowych wyborów, których klasycznym przykładem jest menu. Zauważ, że oprócz akcji skojarzonych z poszczególnymi wartościami przełącznika masz możliwość określenia akcji domyślnej, realizowanej w przypadku wystąpienia wartości nie znajdującej się w naszej „tablicy". Odpowiada za nią słowo kluczowe **`else`** umieszczone na końcu instrukcji. Pewnym ograniczeniem narzucanym przez instrukcję **`case`** jest wymaganie, by zmienna `przełącznik` była typu porządkowego (tak więc nie będziemy mogli zastosować jej w naszym programie, w którym wyznacznik jest liczbą rzeczywistą). Ułatwieniem natomiast jest możliwość „grupowego" definiowania wartości odpowiadających poszczególnym akcjom, np.:

```
case liczba_interesantow of
  0    : mozna_wyjsc_do_fryzjera;
  1, 2 : niech_czekaja;
  3..6 : trzeba_obsluzyc;
  else wypisz('INWENTARYZACJA')
end;
```

Wartości odpowiadające poszczególnym akcjom mogą być wyliczane jawnie lub podawane w postaci zakresów (od..do), o ile tylko tworzą ciągłą grupę. Musisz również pamiętać, że — w odróżnieniu od instrukcji warunkowej — przed słowem **`else`** możesz postawić średnik.

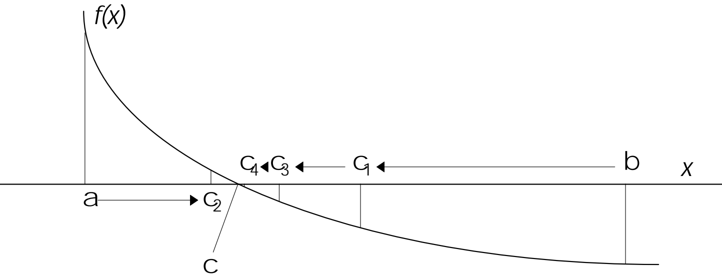
I to by było na tyle, co oczywiście nie znaczy, że możesz spocząć na laurach. Miło jest mieć program potrafiący rozwiązać równanie kwadratowe, jednak znacznie przyjemniejsze byłoby posiadanie narzędzia do rozwiązywania bardziej skomplikowanych równań. Ale czy to aby nie za trudne? Bynajmniej...

Zapamiętaj

- Do warunkowego wykonywania operacji służy w programach pascalowych instrukcja warunkowa **`if-then-else`**.
- Używając instrukcji warunkowych musisz pamiętać o właściwym stosowaniu instrukcji złożonej oraz o braku średnika przed słowem **`else`**.
- Aby obsłużyć więcej niż dwa warianty, możesz użyć zagnieżdżonych instrukcji warunkowych.
- Jeśli wyborem steruje wartość typu porządkowego, lepszym rozwiązaniem jest użycie instrukcji wyboru **`case`**.

Rozwiązujemy dowolne równanie

Problem rozwiązania dowolnego równania nie należy bynajmniej do zadań typu kamienia filozoficznego czy uniwersalnego rozpuszczalnika. Chociaż dla pewnych klas równań istnieją dawno opracowane wzory i metody, spora grupa równań nieliniowych nie daje się rozwiązać „na papierze" w sposób dokładny. Nie oznacza to jednak, że nie można ich w ogóle rozwiązać: kwestii przybliżonego rozwiązywania równań nieliniowych poświęcono z niezłym skutkiem spory dział matematyki znany jako *metody numeryczne*. W niniejszym rozdziale spróbujemy zademonstrować jedną z najprostszych metod rozwiązywania równań nieliniowych — tak zwaną *metodę bisekcji*. Jest ona na tyle prosta, że praktycznie nie wymaga znajomości skomplikowanej matematyki (a jedynie logicznego myślenia), a jednocześnie całkiem skuteczna.



Rysunek 9. *Zasada działania metody bisekcji*

Załóżmy, że chcemy znaleźć miejsce zerowe funkcji *f(x)* przedstawionej powyżej. Szukamy więc takiego punktu *c* na osi *x*, dla którego $f(c) = 0$. Wybierając na osi *x* dwa punkty *a* i *b* takie, by wartości *f(a)* i *f(b)* miały przeciwne znaki, możemy z pewnością stwierdzić, że funkcja ma pomiędzy *a* i *b* co najmniej jedno miejsce zerowe (gdyż musi gdzieś przeciąć oś *x*). Podzielmy przedział pomiędzy *a* i *b* na dwie równe części. Wartość funkcji w nowo uzyskanym punkcie c_1 ma nadal znak przeciwny do *f(a)*, a zatem miejsce zerowe leży gdzieś pomiędzy *a* i c_1. Podzielmy nowo uzyskany przedział na połowy: tym razem okazuje się, że wartość funkcji w punkcie c_2 ma znak przeciwny do

$f(c_1)$, a więc miejsce zerowe leży pomiędzy c_1 a c_2. Wykonując kolejne podziały dojdziemy w końcu do punktu... no, właśnie. Zauważ, że z każdym kolejnym podziałem zbliżamy się do miejsca zerowego coraz wolniej: za każdym razem przedział poszukiwania zawężany jest dwukrotnie, tak więc (teoretycznie) dokładne położenie miejsca zerowego (odpowiadające przedziałowi o zerowej szerokości, czyli pojedynczemu punktowi) osiągniemy po nieskończonej liczbie podziałów. Chyba nie masz zamiaru czekać *aż tak długo!*

Rozwiązaniem tego problemu jest właściwe sformułowanie tzw. *kryterium zakończenia obliczeń* lub *kryterium stopu*. W naszym przypadku kolejne cykle podziałów (zwane fachowo *iteracjami*) będziemy prowadzili tak długo, aż bezwzględna wartość funkcji w punkcie c_n zmaleje poniżej zadanego progu.

Spróbujmy zapisać w bardziej formalny sposób pojedynczy podział:

początek
wyznacz punkt c w połowie odległości pomiędzy a i b
jeżeli f(a) ma znak różny od f(c), to przenieś punkt b do punktu c
w przeciwnym przypadku przenieś punkt a do punktu c
koniec

Ale jak zabrać się za wykonywanie kolejnych podziałów? Oczywiście nie będziesz musiał wielokrotnie wpisywać odpowiednich instrukcji: zamiast tego wykorzystasz do ich cyklicznego wykonywania *instrukcję pętli*.

Spośród trzech dostępnych w Pascalu struktur pętli do realizacji naszego zadania odpowiednie są dwie — **`while`** i **`repeat`** (trzecią pętlą, **`for`**, zajmiemy się nieco później). Należą one do grupy *pętli sterowanych warunkiem*, co oznacza, że wykonują się tak długo, jak długo spełnione będzie odpowiednie kryterium (**`while`** — powtarzaj, dopóki...) lub do momentu, kiedy zostanie ono spełnione (**`repeat`** — powtarzaj, aż...). Nie trzeba chyba dodawać, że owym kryterium będzie właśnie nasze kryterium stopu:

zakończ iteracje, jeśli wartość bezwzględna f(c) jest mniejsza od zadanego progu.

Której instrukcji użyć? W zasadzie w naszym przypadku jest to obojętne. Działanie pętli **`while`** i **`repeat`** jest bardzo zbliżone, zaś różnica sprowadza się do faktu, że w pierwszej z nich warunek przerwania pętli sprawdzany jest na początku:

```
while warunek
do instrukcja
```

zaś w drugiej — na końcu pętli:

```
repeat instrukcja
until warunek
```

W konsekwencji instrukcje tworzące zawartość pętli **`repeat`** muszą wykonać się co najmniej raz, zaś w przypadku pętli **`while`** mogą nie wykonać się ani razu (jeśli `warunek` będzie od razu spełniony). Ponadto musisz pamiętać, że sformułowanie warunku jest dla obu pętli dokładnie przeciwne (pętla **`while`** wykonuje się tak długo, jak długo

`warunek` jest spełniony, **`repeat`** — tak długo, jak długo jest niespełniony). Zapamiętaj również, że umieszczając kilka instrukcji w pętli **`while`** musisz połączyć je w instrukcję złożoną słowami **`begin`** i **`end`**, zaś w przypadku pętli **`repeat`** konieczność ta nie występuje (ogranicznikami instrukcji złożonej są tu słowa **`repeat`** i **`until`**).

Spróbujmy zapisać nasz algorytm z wykorzystaniem pętli **`repeat`**. Będzie on wyglądał mniej więcej tak:

początek
wprowadź wartości krańców przedziału i dokładności
powtarzaj
wyznacz punkt c w połowie odległości pomiędzy a i b
jeżeli f(a) ma znak różny od f(c), to przenieś punkt b do punktu c
w przeciwnym przypadku przenieś punkt a do punktu c
***aż do momentu, gdy** wartość bezwzględna f(c) jest mniejsza od zadanego progu*
wypisz znalezione miejsce zerowe
koniec

Załóżmy, że chcemy rozwiązać równanie

$$1 - e^{\sin x \cdot \cos x} = 0$$

posiadające (jak nietrudno obliczyć na kartce) pierwiastek w punkcie $x = 0$. Tłumacząc powyższy schemat na angielski (i Pascal) otrzymamy następujący program:

```
program Bisekcja;
{ Program rozwiązuje równania nieliniowe metodą bisekcji }

var
  a, b, c : real;  { granice przedziału i punkt podziału }
  eps : real;      { dokładność }

begin
  writeln('Program znajduje miejsce zerowe funkcji')
  writeln('w przedziale [a;b]');
  write('Podaj wartosc a: ');  { wprowadź granice przedziału }
  readln(a);
  write('Podaj wartosc b: ');
  readln(b);
  write('Podaj dokladnosc: ');
  readln(eps);
  repeat
    c := (a + b)/2;  { podziel przedział na pół }
    if (1 - exp(sin(a)*cos(a)))*(1 - exp(sin(c)*cos(c))) < 0
    then
      b := c   { funkcja ma przeciwne znaki w a i c }
    else
      a := c;  { funkcja ma przeciwne znaki w b i c }
  writeln(c);
  until abs(1 - exp(sin(c)*cos(c))) < eps;
  { badamy wartość bezwzględną! }
  writeln('Miejsce zerowe: c = ',c:12:8);
  readln;
```

```
end.
```

Nasz program dość dokładnie odpowiada schematowi podanemu powyżej i w zasadzie nie wymaga dodatkowych komentarzy (mimochodem poznałeś kilka nowych funkcji arytmetycznych — `exp`, `sin`, `cos` i `abs`). Warto jednak zwrócić uwagę na kryterium stopu:

```
until abs(1 - exp(sin(c)*cos(c))) < eps;
```

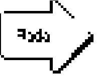
Jak wspomnieliśmy, zapis ten pozwala uniknąć nieskończenie długiego dochodzenia kolejnych podziałów do punktu stanowiącego właściwe rozwiązanie. Ale czy tylko? Spróbuj wykonać poniższy program:

```
program Dodawanie;
{ ilustracja kumulacji błędów zaokrągleń }
{ zmiennoprzecinkowych }

var x : real;

begin
  x := 0.0
  repeat
    x := x + 0.1;     { po 100 dodawaniach powinniśmy... }
    writeln(x:16:12);
  until x = 10.0;  { ... otrzymać dokładnie 10 }
end.
```

Jak łatwo sprawdzić, nasza pętla wcale nie ma zamiaru zakończyć działania po osiągnięciu wartości 10. Czyżby więc dziesięciokrotne dodanie 0.1 do 0.0 nie dawało 1.0? Istotnie: liczby rzeczywiste reprezentowane są w komputerze ze skończoną dokładnością i operacje na większości z nich obarczone są pewnym błędem, który kumuluje się w przypadku wykonywania większej ilości obliczeń. Dlatego też porównania zmiennoprzecinkowe wykorzystujące operator **=** (sprawdzający dokładną równość) zwykle okazują się nieskuteczne. Rozwiązaniem jest użycie operatorów **>=** i **<=**, czyli nierówności nieostrych. Poprawne kryterium zakończenia pętli miałoby u nas postać

```
until x >= 10.0;
```

Wielu początkujących programistów zapomina o niedoskonałości komputera, próbując przełożyć „idealny” zapis matematyczny wprost na język programowania. Efekty mogą być niezbyt przyjemne, zwłaszcza w przypadku wykorzystania takich zapisów w pętlach (dodatkowo występuje wówczas kumulacja błędów) i instrukcjach warunkowych.



Programując operacje na liczbach rzeczywistych należy zawsze pamiętać o ich niedokładnej reprezentacji.

Omówione wyżej pętle **while** i **repeat** należą do grupy pętli sterowanych warunkiem, co oznacza, że liczba przebiegów pętli nie jest z góry określona. Typowym zastosowaniem pętli **while** i **repeat** jest wykonywanie obliczeń (lub innych działań) aż do spełnienia określonego kryterium, wprowadzanie danych z klawiatury lub pliku i wszel-

kie inne operacje, w których nie jesteśmy w stanie wcześniej ustalić, jak wiele cykli trzeba będzie wykonać. Czasem jednak okazuje się, że musimy zrealizować ściśle określoną liczbę operacji, np. wypisać na ekranie wszystkie litery od „A” do „Z” wraz z ich kodami ASCII. Oczywiście i to można zrobić za pomocą pętli **while** lub **repeat**, jednak znacznie poręczniejsza okazuje się tu pętla **for**, sterowana nie warunkiem, lecz *licznikiem*. A oto i sam program:

```
program ASCII;
{ wyprowadza znaki ASCII od 'A' do 'Z' i ich kody }

var c : char;  { wypisywany znak }

begin
  for c := 'A' to 'Z' do
    write(c:4, ord(c):4)
end.
```

Jak widać, składnia pętli **for** jest następująca:

```
for licznik := początek to koniec do
  instrukcja
```

co tłumaczy się „na nasze” jako „dla `licznika` przebiegającego wartości od `początek` do `koniec` wykonuj `instrukcję`”. Pętlę „w dół” (wartość końcowa `licznika` jest mniejsza od początkowej) zapisuje się nieco inaczej:

```
for licznik := początek downto koniec do
  instrukcja
```

Warto o tym pamiętać, gdyż użycie do tego celu poprzedniej postaci spowoduje, że pętla nie wykona się ani razu. Jeśli wreszcie wartość początkowa `licznika` jest równa końcowej, pętla wykona się dokładnie jeden raz (co ma niewielki sens) niezależnie od tego, której z powyższych form użyjemy.

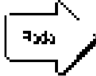
Programując pętlę **for** musisz dokładnie wiedzieć, ile razy będzie się ona wykonywała, czyli znać wartość początkową i końcową `licznika`. Nie jest oczywiście powiedziane, że wartości te musisz podać w postaci stałych: równie dobrze możesz użyć do tego celu zmiennych lub wyrażeń, byle tylko dawały one w wyniku wartości typu porządkowego, np:

```
for i := 1 to MaxIndex div 2 do { ... }
```

Sam `licznik` również musi być zmienną typu porządkowego (zwykle `integer` lub `char`) i w trakcie wykonywania pętli zmienia się z krokiem 1 od wartości `początek` do `koniec` włącznie. Zaprogramowanie w Pascalu pętli z krokiem różnym od 1 (w szczególności niecałkowitym) jest nieco bardziej skomplikowane; zwykle lepiej w tym celu użyć pętli **while** lub **repeat**, chociaż można to zrobić tak:

```
for i := 1 to 20 do
  begin
    writeln(i);
    Inc(i)  { dodaj 1 do licznika }
  end;
```

Powyższa konstrukcja wypisze na ekranie wszystkie liczby nieparzyste zawarte między 0 a 20. Sztuczka, której tu użyliśmy, polega na zmianie licznika w trakcie wykonywania pętli (normalnie zajmuje się tym sama pętla, a my nie powinniśmy się do tego wtrącać) i — chociaż legalna i poprawna — jest niezbyt bezpieczna. Spróbuj np. zmienić wartość końcową licznika na 21 (program „przeskoczy” zakończenie pętli) lub instrukcję `Inc(i)` na `Dec(i)` (zmniejszenie licznika o 1 zamiast zwiększenia — pętla „zapętli się” na wartości 1).



Programując pętlę **`for`** *należy stanowczo unikać jawnego modyfikowania licznika wewnątrz pętli. Aby zaprogramować pętlę z krokiem różnym od 1, lepiej użyć konstrukcji* **`while`** *lub* **`repeat`**.

Wzmiankowane wyżej instrukcje `Inc` i `Dec` pozwalają odpowiednio na zwiększenie (inkrementację) lub zmniejszenie (dekrementację) wartości argumentu typu porządkowego o zadaną wartość (również typu porządkowego) i odpowiadają instrukcjom dodania lub odjęcia wartości, np.

```
Dec(k, 30);  { to samo, co k := k - 30; }
```

Na tym zakończymy rozważania na temat pętli w Pascalu (i kilku drobnych dodatków), choć do zastosowań pętli będziemy wracać jeszcze wielokrotnie. W kolejnym rozdziale zajmiemy się pewną niedogodnością programu `Bisekcja`, polegającą na kłopotliwym zadawaniu postaci rozwiązywanego równania. Zauważ, że nasza funkcja nieliniowa

$$f(x) = 1 - e^{\sin x \cdot \cos x}$$

jest w programie wykorzystywana dwa razy: raz podczas właściwej iteracji, drugi raz w kryterium stopu. Jest to niewygodne i nieeleganckie, gdyż chcąc znaleźć miejsce zerowe innej funkcji musimy wpisywać ją dwa razy. Jednocześnie sam algorytm pozwala nam znaleźć miejsce zerowe dowolnej funkcji *f(x)*. Czy zatem nie dałoby się zamknąć naszej funkcji w jakieś pudełko, do którego wkładalibyśmy tylko wartość argumentu i odbierali wynik? Oczywiście, że tak.

W ramach ćwiczeń praktycznych proponuję Ci następujące zadania:

- zmodyfikuj program *Bisekcja* tak, by wykorzystywał pętlę **`while`**;
- napisz program wypisujący wartości funkcji $f(x) = 1 - e^{\sin x \cdot \cos x}$ w przedziale od $x = 0$ do $x = 360$ stopni (2π) z krokiem 10 stopni. Pamiętaj, że argumenty funkcji trygonometrycznych podawane są w radianach. Spróbuj wykorzystać wszystkie poznane pętle.

Zapamiętaj

- Do cyklicznego (iteracyjnego) wykonywania instrukcji służą w Pascalu pętle.
- Instrukcje pętli mogą być sterowane warunkiem (**`while-do`**, **`repeat-until`**) lub licznikiem (**`for-to/downto`**).

- Pętle **`while`** i **`repeat`** używane są wówczas, gdy nie znamy z góry liczby przebiegów pętli, możemy natomiast określić warunek jej zakończenia.
- Programując kryterium zakończenia pętli **`while`** i **`repeat`** (i nie tylko) należy pamiętać o niedokładności reprezentacji liczb rzeczywistych i kumulacji błędów obliczeń.
- Pętla **`for`** wykorzystywana jest w sytuacjach, gdy możemy dokładnie określić liczbę przebiegów.
- Licznik pętli **`for`** jest zawsze typu porządkowego i zmienia się z krokiem • 1.
- Modyfikacja licznika wewnątrz pętli **`for`** jest dopuszczalna, ale może prowadzić do trudnych do wykrycia błędów i należy jej unikać.
- Do inkrementacji lub dekrementacji zmiennych typu porządkowego można wykorzystać instrukcje `Inc` i `Dec`.

Funkcje i procedury

Jak w życiu, tak i w programowaniu obowiązuje zasada, że nie należy robić tego samego więcej niż raz. Jeśli np. musisz kilkakrotnie obliczyć w programie wartość jakiegoś skomplikowanego wyrażenia, to oczywiście możesz to zrobić przepisując (lub powielając za pomocą operacji blokowych) odpowiednie fragmenty programu, ale trudno powiedzieć, by było to najlepsze rozwiązanie. Po pierwsze, wymaga to trochę pracy. Po drugie, program robi się dłuższy, mniej elastyczny i mniej czytelny (o elegancji nie mówiąc). Po trzecie, jakakolwiek poprawka w treści wyrażenia musi zostać wprowadzona do wszystkich powielonych fragmentów; jeśli zapomnisz choćby o jednym, wyniki mogą być opłakane.

Powyższe argumenty w wystarczającym stopniu uzasadniają potrzebę wprowadzenia rozwiązania umożliwiającego „ukrycie" wybranej operacji (lub grupy operacji) w pojemniku, do którego dostarczalibyśmy dane i odbierali wynik bez przejmowania się sposobem przetwarzania informacji wewnątrz. Rozwiązaniem takim są *funkcje* i *procedury*. Obydwa pojęcia są Ci już znane — wielokrotnie wykorzystywałeś procedury i funkcje biblioteczne Pascala (np. `writeln` czy `exp`) bez zastanawiania się nad zasadą ich działania. Okazuje się jednak, że w nieskomplikowany sposób możesz również tworzyć własne procedury i funkcje, co pozwoli Ci uniknąć prowizorki, jaką zaprezentowaliśmy w poprzednim rozdziale.

Czym jest funkcja? Z punktu widzenia programu, który ją wykorzystuje, jest to rodzaj „czarnej skrzynki", przetwarzającej włożone do niej informacje i zwracającej odpowiedni wynik. Z punktu widzenia programisty natomiast funkcja jest swoistym „miniprogramem" — zestawem zamkniętych w logiczną całość instrukcji. Instrukcje te mogą zajmować się wykonywaniem dowolnych operacji, chociaż na ogół ich celem jest przekształcenie wprowadzonych do funkcji informacji (tzw. *parametrów*) do żądanej postaci, zwracanej następnie poprzez nazwę funkcji. Procedura różni się od funkcji jedynie sposobem zwracania wyniku, o czym powiemy za chwilę.

Co należy zrobić, żeby wykorzystać w programie funkcję lub procedurę? Po pierwsze trzeba ją *zdefiniować*, a następnie *wywołać*. Zajmijmy się na początek definicją, która polega na:

- określeniu sposobu kontaktowania się jej z otoczeniem (ustaleniu nazwy, zestawu parametrów i typu zwracanego wyniku), oraz

* określeniu jej zawartości (czyli zestawu tworzących ją instrukcji i pomocniczych obiektów lokalnych, o których później).

A oto składnia definicji funkcji[3]:

```
function nazwa-funkcji(lista-parametrów):typ-wyniku;
deklaracje-i-definicje-obiektów lokalnych
begin
  instrukcje-realizujące-treść funkcji
end;

procedure nazwa-procedury(lista-parametrów);
deklaracje-i-definicje-obiektów-lokalnych
begin
  instrukcje-realizujące-treść-procedury
end;
```

Definicja rozpoczyna się tak zwanym *nagłówkiem*, sygnalizowanym słowem kluczowym **`function`** lub **`procedure`**, po którym następuje umieszczona w nawiasach okrągłych lista *parametrów (argumentów) formalnych*. Parametry te symbolizują wartości, które zostaną przekazane do funkcji i przetworzone, zaś sama lista ma postać:

```
lista-nazw : typ; lista-nazw : typ; ...
```

czyli zawiera zestawy nazw parametrów tego samego typu rozdzielone średnikami (w obrębie zestawu nazwy parametrów rozdzielone są przecinkami). Warto zauważyć, że lista parametrów nie jest obowiązkowym elementem definicji, tak więc istnieje możliwość zdefiniowania funkcji bezparametrowej, nie pobierającej informacji z otoczenia. Ostatnim elementem nagłówka jest określenie typu zwracanego wyniku, wymagane wyłącznie dla funkcji.

Sama treść funkcji zdefiniowana jest pomiędzy słowami kluczowymi **`begin`** i **`end`** i może być poprzedzona deklaracjami i definicjami tak zwanych obiektów lokalnych, wykorzystywanych przez funkcję do użytku wewnętrznego. Obiektami lokalnymi zajmiemy się szerzej w dalszej części książki. Same instrukcje składające się na treść definicji funkcji nie różnią się niczym od używanych w „zwykłym" programie, z wyjątkiem tego, że w przypadku funkcji należy umieścić w definicji instrukcję przypisania

```
nazwa-funkcji := wartość-wyniku
```

umożliwiającą przekazanie obliczonego wyniku „na zewnątrz". Co ciekawe, obecność tej instrukcji w treści funkcji nie jest sprawdzana przez kompilator (jak ma to miejsce np. dla kompilatorów języka C), toteż zapominalskim programistom może się przytrafić zdefiniowanie funkcji kompletnie nie potrafiącej skomunikować się z otoczeniem (nie pobierającej żadnych parametrów i nie zwracającej wyniku), a więc praktycznie bezużytecznej. Warto zdawać sobie sprawę, że wywołanie funkcji pozbawionej instrukcji zwrócenia wyniku da w rezultacie wartość nieokreśloną (przypadkową), co raczej nie może być uznane za efekt pozytywny.

---

[3] Dla uproszczenia, w dalszym ciągu naszych rozważań będziemy używali słowa „funkcja" zarówno na określenie funkcji, jak i procedury. Ewentualne różnice będą wyraźnie zaznaczane.

Jak widać z powyższych zapisów, definicja funkcji lub procedury przypomina mały program. Definicje wszystkich funkcji i procedur wykorzystywanych w programie muszą być umieszczone przed miejscem ich wywołania. To ostatnie może nastąpić w części operacyjnej programu (a zatem definicje muszą poprzedzać rozpoczynające właściwy program słowo **begin**) lub w treści innej funkcji lub procedury (co wymusza odpowiednie ułożenie definicji względem siebie[4]). Struktura programu wykorzystującego funkcje i procedury powinna więc wyglądać następująco:

```
nagłówek-programu
deklaracje-i-definicje-obiektów-globalnych
definicje-funkcji-i-procedur
begin
  część-operacyjna-programu
end.
```

Umieszczanie deklaracji i definicji obiektów globalnych (o których również powiemy później) przed definicjami funkcji nie jest co prawda obowiązkowe (chyba, że obiekty te są wykorzystywane w funkcjach), stanowi jednak element dobrej praktyki programistycznej. Natomiast jednym z pospolitych błędów popełnianych przez początkujących programistów jest próba umieszczania definicji funkcji i procedur w części operacyjnej programu, co jest oczywiście nielegalne.

Samo zdefiniowanie funkcji nic nam jeszcze nie daje (jego efektem jest wyłącznie utworzenie odpowiedniego fragmentu kodu wynikowego). Dopiero *wywołanie* funkcji powoduje wykonanie składających się na jej treść instrukcji i rzeczywiste przekształcenie informacji podanej w postaci parametrów. Aby wywołać procedurę lub funkcję, wystarczy umieścić w programie jej nazwę wraz z listą *parametrów aktualnych*, czyli wyrażeń zawierających informację, która ma być przetworzona. Zawartość listy parametrów aktualnych musi odpowiadać liście podanej w definicji funkcji. W szczególności, jeśli w definicji pominięto listę parametrów, nie podaje się ich podczas wywołania (mamy wówczas do czynienia z funkcją lub procedurą bezparametrową).

Wywołanie procedury ma postać

```
nazwa-procedury(lista-parametrów-aktualnych);
```

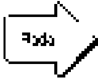
zaś w przypadku funkcji, która zwraca obliczoną wartość poprzez swoją nazwę, musisz dodatkowo zadbać o umieszczenie gdzieś zwróconego wyniku, np. tak:

```
zmienna := nazwa-funkcji(lista-parametrów-aktualnych);
```

Funkcję możesz również wykorzystać jako element wyrażenia lub instrukcji (np. `writeln`); ogólnie, możesz ją zastosować wszędzie tam, gdzie możliwe jest użycie wyrażenia[5].

[4] Ograniczenie to można obejść, wykorzystując tzw. deklaracje zapowiadające, rozpoczynane słowem kluczowym **`forward`**.

[5] Pamiętaj, że identyfikator funkcji użyty w programie nie jest *l*-wartością, a więc nie możesz używać funkcji tak samo, jak zmiennej — nie możesz np. przypisywać jej wartości (możliwe to jest wyłącznie w obrębie definicji funkcji i ma wówczas nieco inne znaczenie).



*Począwszy od wersji 6.0 Turbo Pascal umożliwia Ci zignorowanie wyniku zwracanego przez funkcję, czyli wykorzystanie jej tak samo, jak procedury. Jest to możliwe po włączeniu opcji kompilatora **Extended Syntax** lub użyciu odpowiadającej jej dyrektywy* `{$X+}`.

Uff. Po tej solidnej (i nużącej) dawce teorii czas na nieco praktyki. Na początek spróbujmy zaradzić problemowi przedstawionemu pod koniec poprzedniego rozdziału. Oto poprawiona wersja programu `Bisekcja`, wykorzystująca funkcję:

```
program Bisekcja;
{ Program rozwiązuje równania nieliniowe metodą bisekcji }

var
  a, b, c : real;  { granice przedziału i punkt podziału }
  eps : real;      { dokładność }

function f(x:real):real;        { badana funkcja }

begin
  f := 1 - exp(sin(x)*cos(x));  { treść funkcji }
end;

begin
  writeln('Program znajduje miejsce zerowe funkcji')
  writeln('w przedziale [a; b]');
  write('Podaj wartosc a: ');  { wprowadź granice przedzialu }
  readln(a);
  write('Podaj wartosc b: ');
  readln(b);
  write('Podaj dokladnosc: ');
  readln(eps);
  repeat
    c := (a + b)/2;  { podziel przedział na pół }
    if (f(a)*f(c)) < 0 then  { wygląda ładniej, prawda? }
      b := c   { funkcja ma przeciwne znaki w a i c }
    else
      a := c;  { funkcja ma przeciwne znaki w b i c }
  writeln(c);
  until abs(f(c)) < eps; { badamy wartość bezwzględną! }
  writeln('Miejsce zerowe: c = ',c:12:8);
  readln;
end.
```

Czym różni się nasz program od wersji poprzedniej? Po pierwsze, definicja badanej funkcji została wyodrębniona jako oddzielny fragment, który łatwo znaleźć, zinterpretować i poprawić. Po drugie, treść samego programu jest czytelniejsza (program zajmuje się teraz wyłącznie realizacją właściwego algorytmu, czyli szukaniem pierwiastka *pewnej funkcji f(x)*. Po trzecie, zmniejszyła się podatność programu na błędy (zmiana definicji funkcji wymaga poprawienia tylko jednej linijki; zastanów się, jak wyglądałaby poprzednia wersja programu, gdyby obliczenia wymagały użycia kilku oddzielnych instrukcji). Wreszcie po czwarte, program wygląda ładniej, jest bardziej elegancki i wymaga mniej pisania.

Korzystanie z procedur wygląda bardzo podobnie, z wyjątkiem tego, że nie musisz się troszczyć o zagospodarowanie zwracanej wartości (ponieważ jej nie ma). Ogólnie rzecz biorąc, funkcje wykorzystywane są głównie tam, gdzie zachodzi potrzeba obliczenia jakiejś pojedynczej wartości i przekazania jej do kolejnych instrukcji programu, natomiast procedury pozwalają na efektywne wykonywanie powtarzających się, rutynowych czynności. Aby na przykład uatrakcyjnić szatę graficzną programu, możesz wykorzystać w nim następującą procedurę:

```
procedure Szlaczek;

begin
  for i := 1 to 60 do  { wypisz szlaczek }
    write('*');  { złożony z 60 gwiazdek }
  writeln;  { przejdź do nowego wiersza }
end;
```

Gdzie umieścić definicję procedury — już wiesz. Pozostaje ją wywołać, co robi się bardzo prosto:

```
Szlaczek;
```

Każde takie wywołanie umieści na ekranie „szlaczek" składający się z 60 gwiazdek. Oczywiście, powyższa procedura nie jest idealna, bo jeśli np. zachce Ci się wypisać szlaczek złożony z kresek, będziesz musiał zmienić jej treść. Znacznie lepszym rozwiązaniem jest coś takiego:

```
procedure Szlaczek(Znak:char; Ile:byte);

begin
  for i := 1 to Ile do  { wypisz szlaczek }
    write(Znak);  { złożony z iluś znaków }
  writeln;
end;
```

Spróbujmy w ramach zabawy wykorzystać naszą procedurę do wyświetlenia na ekranie trójkąta złożonego z gwiazdek, czyli czegoś takiego:

```
*
**
***
****
```

itd. Odpowiedni program będzie wyglądał tak:

```
program Szlaczki;

var
  i : integer;  { licznik wierszy }

procedure Szlaczek(Znak : char; Ile : integer);

begin
  for i := 1 to Ile do  { wypisz szlaczek }
      write(Znak);      { złożony z iluś znaków }
```

```
  writeln
end;

begin
  for i := 1 to 20 do   { wypisz 20 szlaczków }
    Szlaczek('*', i);
end.
```

Jeśli uważasz, że wszystko jest OK, spróbuj teraz zmienić program tak, by każdy szlaczek miał długość równą kolejnej liczbie nieparzystej, tj. pierwszy 1, drugi 3 itd. Ogólnie długość szlaczka wyrazi się wzorem $Ile = 2 \cdot numer - 1$, gdzie *numer* jest numerem wiersza, tak więc wystarczy zmienić główną pętlę programu na

```
for i := 1 to 20 do { wypisz 20 szlaczków }
  Szlaczek('*', 2*i - 1);
```

Hm... no, chyba nie całkiem to chciałeś uzyskać, prawda? Efekt, którego doświadczyłeś uruchamiając program wynika z użycia w procedurze i wywołującej ją pętli tej samej zmiennej `i`, a wygląda następująco:

Wartość	***Przebieg pętli***		
	1	***2***	***3***
ma być	2	1	3
jest przed procedurą	2	1	3
jest po procedurze	3	1	7
jest po inkrementacji licznika	4	2	8

Każde wywołanie procedury powoduje modyfikację zmiennej `i`, która jest jednocześnie licznikiem pętli w programie głównym. Aby tego uniknąć, należałoby w procedurze użyć jako licznika pętli innej zmiennej. To z kolei nakłada na programistę obowiązek przejrzenia wszystkich procedur znajdujących się w programie i wcześniejszego zadeklarowania odpowiednich zmiennych tak, by się nie „nakładały". W efekcie otrzymujemy kolejny zestaw łatwych do przeoczenia pułapek. Rozwiązaniem tego problemu są *zmienne lokalne*, którymi jednak zajmiemy się już w następnym rozdziale. Tam również dokończymy omawianie przekazywania parametrów i wartości z i do procedur i funkcji.

Zapamietaj

- Procedury i funkcje pascalowe pozwalają zamknąć dany zestaw operacji w logiczną całość, pobierającą z otoczenia odpowiednie informacje i zwracającą żądany wynik.
- Przed wykorzystaniem (wywołaniem) funkcji lub procedury należy ją zdefiniować, czyli określić jej treść i sposób komunikowania się z otoczeniem.

- Do przekazywania informacji do funkcji (procedury) służą parametry (argumenty). W trakcie definiowania informacja przekazywana do funkcji (procedury) symbolizowana jest parametrami formalnymi.
- Definicje funkcji (procedur) muszą być umieszczone w programie przed częścią operacyjną i muszą występować w odpowiedniej kolejności.
- Samo wywołanie funkcji (procedury) odbywa się przez podanie w programie jej nazwy uzupełnionej odpowiednią listą parametrów aktualnych.
- Stosowanie funkcji i procedur jest korzystne ze względu na poprawę czytelności i efektywności programu, zmniejszenie podatności na błędy i skrócenie czasu potrzebnego na jego pisanie.

Jak program porozumiewa się z funkcją?

Jak powiedzieliśmy w poprzednim rozdziale, funkcja (procedura) pozwala zamknąć zestaw złożonych operacji w swoisty pojemnik, wykorzystywany przez wywołujący program na zasadzie czarnej skrzynki: po włożeniu do funkcji zestawu informacji, program odbiera wynik nie przejmując się sposobem jego uzyskania. Niestety, jak mogłeś się już przekonać, nasza skrzynka nie jest szczelna i stosunkowo łatwo doprowadzić do sytuacji, w której funkcja w sposób niekontrolowany „uszkodzi" zmienne programu. Zjawiska tego typu (zarówno kontrolowane, jak i niekontrolowane) zwane są *efektami ubocznymi* (ang. *side effects*) i dość często utrudniają życie mniej doświadczonym programistom. Na szczęście efektów ubocznych można łatwo uniknąć: pozwalają na to *zmienne lokalne*[6].

Jak sama nazwa wskazuje, zmienna lokalna jest „prywatną własnością" funkcji lub procedury, wewnątrz której została zdefiniowana. Zmienna taka nie jest widoczna na zewnątrz funkcji (czyli np. w wywołującym ją programie), natomiast jest dostępna dla wszelkich funkcji i procedur zdefiniowanych w obrębie danej funkcji lub procedury (o ile ich definicje umieszczone są po deklaracji zmiennej). Przeciwieństwem zmiennej lokalnej jest oczywiście zmienna *globalna*, „widoczna" w całym obszarze programu od momentu jej zadeklarowania. Każda zdefiniowana w programie funkcja i procedura ma pełny dostęp do wszystkich zmiennych globalnych, czego jednak nie należy nadużywać (a wręcz nie należy używać w ogóle). „Zakres widzialności" zmiennej nazywany jest jej *zasięgiem* (ang. *scope*).

Zarówno deklaracja, jak i sposób wykorzystania zmiennych lokalnych są „na oko" takie same, jak dla zmiennych globalnych. Różnica kryje się w lokalizacji zmiennych obu

[6] Ściślej, *obiekty* lokalne, bowiem oprócz zmiennych lokalne mogą być również stałe i typy, o których jednak na razie nie mówiliśmy.

typów: zmienne globalne przechowywane są w tzw. *segmencie danych* i istnieją przez cały czas wykonywania programu. Dzięki temu są zawsze dostępne dla każdej ze zdefiniowanych w programie funkcji, o ile nie zachodzi tzw. zjawisko przysłaniania, o którym za chwilę. Zmienne lokalne umieszczane są z kolei na *stosie*, czyli w specjalnym obszarze pamięci przeznaczonym do tymczasowego przechowywania informacji. W odróżnieniu od zmiennych globalnych, czas życia zmiennych lokalnych ograniczony jest do czasu wykonywania posiadającej je funkcji, tak więc „lokalność” ma charakter nie tylko przestrzenny, ale i czasowy — zmienne lokalne są tworzone w momencie rozpoczęcia wykonywania funkcji i znikają po jej zakończeniu. Oczywiste jest więc, że próba odwołania się do zmiennej lokalnej na zewnątrz deklarującej ją funkcji jest nielegalna, gdyż zmienna taka po prostu nie istnieje[7].

Jak wcześniej wspomnieliśmy, funkcja (procedura) przypomina strukturą mały program, toteż deklaracje zmiennych lokalnych umieszcza się zwykle na jej początku, po nagłówku. Użycie zmiennych lokalnych zademonstrujemy na przykładzie programu `Szlaczki`, doprowadzając go przy okazji do stanu pełnej używalności. Aby to zrobić, wystarczy poprawić procedurę `Szlaczek`:

```
procedure Szlaczek(Znak : char; Ile : integer);

var  { deklaracja zmiennej lokalnej, }
  i : integer;  { czyli licznika znaków }

begin
  for i := 1 to Ile do  { wypisz szlaczek }
    write(Znak);        { złożony z iluś znaków }
  writeln
end;
```

Tak poprawiony program powinien działać poprawnie niezależnie od żądanej długości ciągu znaków. Wprowadzając do procedury `Szlaczek` deklarację zmiennej `i` spowodowałeś, że podczas każdego wywołania korzysta ona nie ze zmiennej globalnej, ale ze swojej „prywatnej” zmiennej lokalnej. Zauważ przy okazji, że obie zmienne nazywają się tak samo, a mimo to program działa poprawnie. Zjawisko „maskowania” zmiennej globalnej przez identycznie nazwaną zmienną lokalną nazywa się *przysłanianiem* i daje Ci gwarancję, że nawet jeśli zadeklarujesz w funkcji zmienną lokalną o nazwie takiej samej jak zmienna zewnętrzna (w szczególności globalna), wszelkie odwołania wewnątrz funkcji będą zawsze kierowane do zmiennej lokalnej, zaś odpowiednia zmienna zewnętrzna pozostanie nienaruszona. Dzięki temu nie musisz dbać o to, by zmienne deklarowane w ramach poszczególnych funkcji miały różne nazwy i by któraś przypadkiem nie nazywała się tak samo, jak odpowiednia zmienna globalna.

Aby zobrazować pojęcia zasięgu i przysłaniania, wyobraźmy sobie pewien abstrakcyjny program, w którym zdefiniowano kilka zmiennych globalnych oraz procedur i funkcji

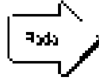
---

[7] Wyjątkiem są tutaj lokalne zmienne predefiniowane (zmienne z wartością początkową), które lokowane są w segmencie danych, a więc istnieją przez cały czas wykonywania programu. Nie zmienia to faktu, że i one nie są widoczne na zewnątrz posiadających je funkcji i procedur.

posiadających swoje zmienne lokalne. Na rysunku obok przedstawiony jest schemat takiego programu. W tabelce pokazany jest zasięg poszczególnych zmiennych (widzialność w programie i procedurach), znak gwiazdki oznacza przysłanianie.

Kiedy używać zmiennych lokalnych, a kiedy globalnych? Zasadniczo, zmiennych globalnych w procedurach i funkcjach nie należy używać w ogóle. Zdarzają się jednak sytuacje, kiedy **świadome** użycie zmiennej globalnej (czyli świadome wykorzystanie efektów ubocznych) upraszcza program. Efekty uboczne znajdują na ogół zastosowanie do przekazywania informacji z i do funkcji (procedur), pozwalając na zredukowanie liczby parametrów. Jeśli jednak chciałbyś zaoszczędzić sobie pisania deklarując w programie jedną zmienną globalną `i` i używając jej jako licznika we wszystkich procedurach, które takowego potrzebują, szukasz guza. Prędzej czy później któraś z przyszłych modyfikacji programu spowoduje ujawnienie się efektów ubocznych w formie prowadzącej do błędu wykonania.

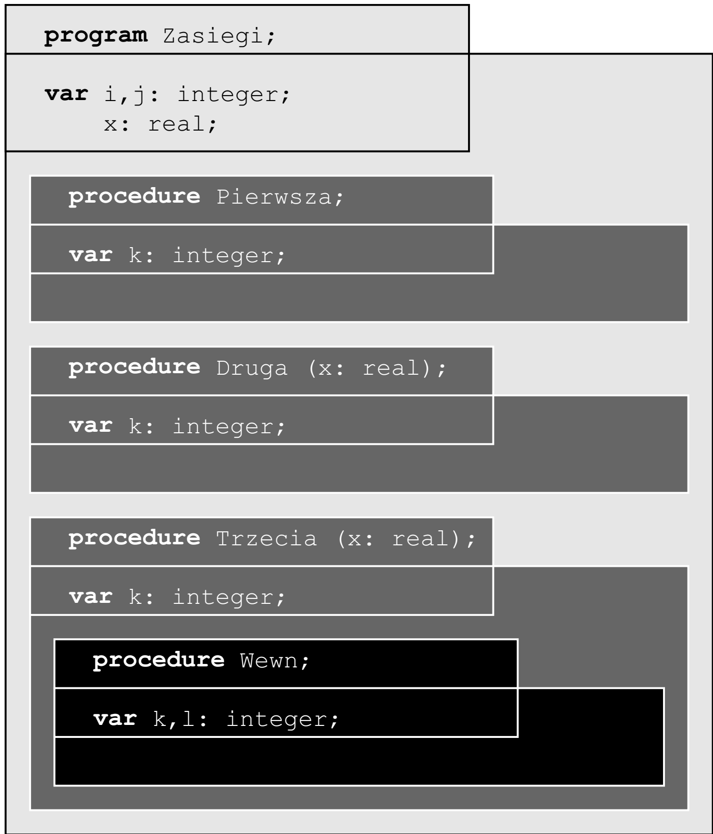
Ponieważ wszelkie czynności wykonywane przez funkcję lub procedurę są niedostrzegalne dla wywołującego ją programu, zmienne wykorzystywane w trakcie wykonywania tych czynności powinny być dla niego niedostrzegalne, a zatem powinny być deklarowane jako zmienne lokalne. W szczególności dotyczy to zmiennych tymczasowych i pomocniczych, jak liczniki pętli, tymczasowe wartości maksimów i minimów, zmienne tekstowe używane przy wyprowadzaniu komunikatów itp.



Wszelkie zmienne przeznaczone wyłącznie do użytku wewnętrznego funkcji i procedur powinny być bezwzględnie deklarowane jako zmienne lokalne. Zmienne globalne powinny być wykorzystywane tylko do przechowywania danych istniejących przez cały czas trwania programu.

W ostatniej części tego rozdziału zajmiemy się sprawą przekazywania informacji pomiędzy funkcjami i procedurami a wywołującym je programem. Jak już wiesz, wymiana taka może być zrealizowana z wykorzystaniem efektów ubocznych (co nie jest wskazane) lub za pomocą parametrów (argumentów). Deklarowanie i wykorzystanie parametrów było już omawiane poprzednio, przypomnijmy więc w skrócie:

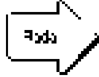
- parametry deklarowane są w nagłówku funkcji (procedury) jako tzw. *parametry formalne*, czyli identyfikatory (nazwy) symbolizujące dane przekazywane do funkcji;
- podczas wywołania w miejsce parametrów formalnych podstawiane są rzeczywiste wartości zmiennych, stałych i wyrażeń, tworzące tzw. *parametry aktualne*;
- parametry formalne opisują jedynie postać informacji przekazywanej do funkcji (procedury); rzeczywistą informację niosą ze sobą dopiero parametry aktualne.



Zmienna (właściciel)	*Zasięg*				
	Program	Pierwsza	Druga	Trzecia	Wewn
i (program)	tak	tak	tak	tak	tak
j (program)	tak	nie (*)	tak	tak	tak
x (program)	tak	tak	nie (*)	nie (*)	nie
j (Pierwsza)	nie	**tak**	nie	nie	nie
k (Pierwsza)	nie	**tak**	nie	nie	nie
x (Druga)	nie	nie	**tak**	nie	nie
k (Druga)	nie	nie	**tak**	nie	nie
x (Trzecia)	nie	nie	nie	**tak**	**tak**
k (Trzecia)	nie	nie	nie	**tak**	nie (*)
k (Wewn)	nie	nie	nie	nie	**tak**
l (Wewn)	nie	nie	nie	nie	**tak**

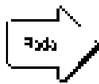
Rysunek 10. *Widzialność zmiennych lokalnych i globalnych w programie*

Wewnątrz funkcji parametry zachowują się jak normalne zmienne (można wykonywać na nich wszelkie operacje tak samo, jak na zmiennych lokalnych). Możliwe jest zadeklarowanie parametru o nazwie takiej samej, jak zmienna globalna (w ogólności — zmienna zadeklarowana w bloku nadrzędnym); wystąpi wówczas efekt przysłaniania, tj. wszelkie odwołania wewnątrz funkcji korzystającej z parametru będą odnosiły się do niego, a nie do przysłoniętej przezeń zmiennej. Nielegalne jest natomiast deklarowanie zmiennych lokalnych i parametrów o identycznych nazwach (nie mówiąc już o zadeklarowaniu parametru o nazwie identycznej z nazwą funkcji, która go wykorzystuje).



Wszystkie identyfikatory (tj. nazwy zmiennych, stałych, typów, parametrów, procedur i funkcji) deklarowane w obrębie danego bloku (z wyjątkiem bloków podrzędnych) muszą być unikalne. Możliwe jest natomiast „zdublowanie” identyfikatora w bloku podrzędnym, co spowoduje efekt przysłaniania.

Warto wspomnieć, że początkujący programiści często nadużywają parametrów, wykorzystując je jako zmienne lokalne lub (co jest już zbrodnią) deklarują dodatkowe parametry tylko i wyłącznie w tym celu. Praktyka ta, chociaż legalna, jest sprzeczna z zasadami poprawnego programowania: parametry przeznaczone są do wymiany informacji pomiędzy funkcją, nie zaś do „podpierania się” w wykonywanych przez nią operacjach o charakterze wewnętrznym.



Tworząc program musisz pamiętać o właściwym podziale kompetencji pomiędzy składającymi się nań obiektami. Jeśli obiekt przeznaczony jest do przekazywania informacji, nie powinieneś używać go jako zmiennej lokalnej. Poczta nie służy do przesyłania wiadomości do sąsiedniego pokoju.

Do tej pory koncentrowaliśmy się na przekazywaniu informacji z wywołującego programu do funkcji lub procedury. Po przetworzeniu tej informacji na ogół zachodzi konieczność zwrócenia wyniku do wywołującego. Jeśli wynik jest pojedynczą wartością (jest typu prostego), najlepiej użyć funkcji, która pozwala na zwrócenie go przez nazwę. Rozwiązanie to bywa jednak czasami niewygodne, nie mówiąc już o sytuacjach, gdy trzeba zwrócić do wywołującego dwie lub więcej wartości. Ponieważ trzymamy się z daleka od efektów ubocznych, pozostaje wykorzystanie do tego celu parametrów. Wyobraź sobie na przykład, że jest Ci potrzebna funkcja obliczająca jednocześnie sinus i cosinus danego kąta. Można to zrobić np. tak:

```
program Trygonometria;
{ demonstracja przekazywania parametrów }
var
  i : integer;
  s, c : real;

procedure SinCos(x : real; SinX, CosX : real);
{ oblicza jednocześnie sinus i cosinus argumentu }

begin
 SinX := sin(x);
 CosX := cos(x);
end;
```

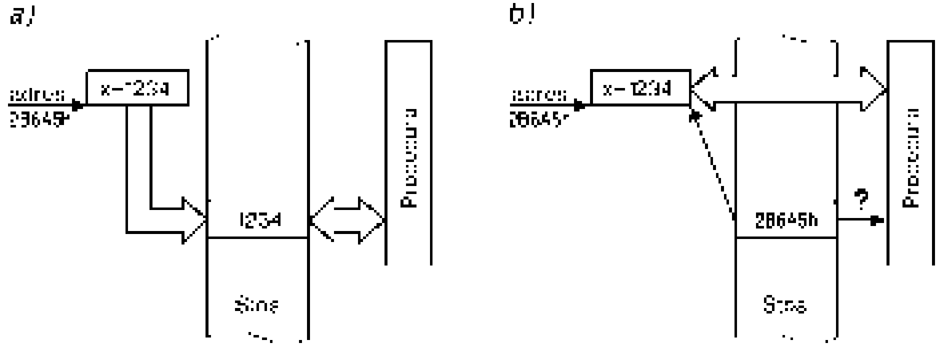
```
begin
  SinCos(Pi/3,s,c);      { oblicz wartości }
  writeln(s:8:3, c:8:3); { i wypisz je na ekranie }
end.
```

Niestety, wynik bynajmniej nie jest satysfakcjonujący: zamiast obliczonych wartości program wyświetla zera lub inne pozbawione sensu liczby. Dzieje się tak dlatego, iż w swojej obecnej formie parametry `SinX` i `CosX` pozwalają jedynie na przekazanie informacji do procedury, nie zaś odwrotnie. Na szczęście wystarczy jedna drobna modyfikacja w programie:

```
procedure SinCos(x : real; var SinX, CosX : real);
```

by wszystko zaczęło działać poprawnie. Co takiego się zmieniło?

Otóż dodanie słowa kluczowego **var** przed nazwą parametru (parametrów) zasadniczo zmienia sposób ich przekazywania. Poprzednio mieliśmy do czynienia z tak zwanym *przekazywaniem przez wartość*, umożliwiającym wyłącznie włożenie danych do funkcji (procedury). Działanie przekazywania przez nazwę sprowadzało się do umieszczenia w odpowiednim miejscu (tj. na stosie) *kopii wartości parametru aktualnego*, która następnie mogła być przetwarzana wewnątrz funkcji lub procedury. Obecnie wykorzystywany mechanizm nosi nazwę *przekazywania przez nazwę* (inne mądre określenie to przekazywanie przez wskaźnik lub przez referencję) i różni się od poprzedniego tym, że procedura otrzymuje nie kopię wartości parametru aktualnego, lecz *informację o jego położeniu*. Można to przedstawić następująco:



Rysunek 11. *Mechanizm przekazywania parametrów: a) przez wartość, b) przez nazwę*

W przypadku przekazywania przez wartość funkcja otrzymuje jedynie kopię rzeczywistego parametru. Zasięg operacji dokonywanych na takiej kopii ogranicza się do wnętrza funkcji, zaś gdy ta kończy działanie, wartość parametru jest tracona. Parametry przekazywane przez wartość zachowują się więc identycznie jak zmienne lokalne, różniąc się od nich jedynie przeznaczeniem. Inaczej wygląda sprawa w przypadku przekazywania przez nazwę: funkcja otrzymuje tu nie kopię parametru, lecz tzw. *wskaźnik* opisujący jego rzeczywiste położenie. Wszelkie operacje dokonywane przez funkcję na parametrze przekazywanym przez nazwę dotyczą nie kopii, ale rzeczywistego obiektu będącego własnością programu. Można więc powiedzieć, że funkcja „sięga na

zewnątrz” do zasobów programu i modyfikuje je ze skutkiem natychmiastowym, a efekty wszystkich operacji pozostają „utrwalone” w treści obiektu będącego parametrem. Zauważ, że ponieważ funkcja otrzymuje tu wskaźnik do rzeczywistego parametru, nie możesz przekazać przez nazwę obiektu nie posiadającego lokalizacji w pamięci (np. stałej czy wyrażenia), czyli nie będącego *l*-wartością. Przekazanie takiego parametru przez wartość jest natomiast najzupełniej legalne.

Kiedy stosować przekazywanie parametrów przez wartość, a kiedy przez nazwę? Przekazywanie przez wartość używane jest w „komunikacji jednokierunkowej”, czyli wówczas, gdy zwrócenie wartości do wywołującego programu nie jest wymagane lub wręcz jest niewskazane (tj. chcemy zabezpieczyć się przed modyfikacją obiektu będącego parametrem). Jeżeli zachodzi potrzeba przekazania wartości z powrotem do wywołującego, konieczne jest użycie przekazywania przez nazwę, czyli poprzedzenie nazwy parametru słowem **`var`**. Przekazywanie przez nazwę stosuje się również dla argumentów o większych rozmiarach (np. tablic, łańcuchów i innych zmiennych strukturalnych, o których powiemy wkrótce), nawet wówczas, gdy nie chcemy zwracać ich wartości. Jak powiedzieliśmy przed chwilą, przekazanie przez wartość powoduje umieszczenie na stosie kopii parametru, co przy większych rozmiarach parametrów zajmuje sporo czasu i może doprowadzić do przepełnienia stosu (na którym oprócz tego muszą zmieścić się zmienne lokalne i kilka innych rzeczy). Podczas przekazywania przez nazwę na stosie umieszczany jest zawsze czterobajtowej długości wskaźnik do parametru (niezależnie od faktycznej wielkości tego ostatniego), co oczywiście trwa znacznie krócej. Aby zabezpieczyć taki parametr przed niepożądaną modyfikacją, możemy zastosować w miejsce słowa **`var`** słowo kluczowe **`const`**, które również powoduje przekazanie wskaźnika do parametru, blokując jednocześnie możliwość modyfikowania tego ostatniego. Aby się o tym przekonać, spróbuj zmierzyć czas wykonania następującego programu:

```
program Parametry;

type
  TablInt = array [1..5000] of integer;

var
  i : integer;
  ti : TablInt;

procedure Pusta(t : TablInt);

var k : integer;

begin
  k := t[1]
end;

begin
  for i := 1 to 1000 do { to może potrwać! }
    Pusta(ti);
end.
```

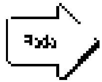
Jeśli już to zrobiłeś, zmień nagłówek procedury `Pusta` na

```
procedure Pusta(var t : TablInt)
```

Tym razem program powinien wykonać się błyskawicznie (przynajmniej w porównaniu do poprzedniej wersji). Dzieje się tak dlatego, że podczas każdego wywołania procedury na stosie umieszczany jest jedynie wskaźnik do tablicy `t` (w sumie tysiąc razy po 4 bajty), podczas gdy poprzednio na stos kopiowana była cała tablica (tysiąc razy 10 000 bajtów, czyli 2500 razy więcej). Dodatkową „atrakcją" związaną z przekazywaniem większych struktur przez wartość jest możliwość przepełnienia stosu (standardowo wielkość segmentu stosu ustawiana jest przez Turbo Pascala na 16 kB). Aby się o tym przekonać, zmień w pierwotnej wersji programu (tej bez słowa **`var`** w nagłówku procedury) rozmiar tablicy na 10000 liczb:

```
type
  TablInt = array [1..10000] of integer;
```

Próba wykonania programu skończy się tym razem komunikatem `Stack overflow error`: sumaryczny rozmiar tablicy wynosi 20000 bajtów (10000 dwubajtowych liczb całkowitych), czyli więcej niż można umieścić na stosie. Podobnie jak poprzednio, przekazanie parametru przez nazwę rozwiązuje problem.



Przekazując większe struktury danych jako parametry funkcji i procedur należy unikać przekazywania przez wartość i stosować w zamian przekazywanie przez nazwę. Aby zabezpieczyć się przed przypadkową modyfikacją wartości parametru, można zastosować słowo kluczowe **`const`** *w miejsce* **`var`**.

W ten sposób zakończyliśmy omawianie podstawowych zagadnień związanych z procedurami i funkcjami. W następnych rozdziałach zajmiemy się wyjaśnianiem dwóch nowinek, które pojawiły się w ostatnim programie, i które zapewne przyjąłeś na wiarę. Są to mianowicie tablice (a właściwie typy strukturalne) oraz definiowanie własnych typów, czyli słowo kluczowe **`type`**.

Zapamiętaj

- Wszelkie obiekty zadeklarowane wewnątrz funkcji (procedury) są tzw. obiektami lokalnymi. Obiekty lokalne są niewidoczne na zewnątrz funkcji.
- Obiekty zadeklarowane w programie głównym (na zewnątrz wszystkich procedur i funkcji) są obiektami globalnymi. Są one dostępne dla wszystkich elementów programu.
- Wszelkie obiekty przeznaczone wyłącznie do użytku wewnętrznego funkcji (procedury) powinny być deklarowane jako lokalne.
- Zakres „widzialności" obiektu zwany jest jego zasięgiem.
- Zadeklarowanie w funkcji (procedurze) obiektu lokalnego o takiej samej nazwie jak obiekt z bloku nadrzędnego powoduje przysłonięcie tego ostatniego.

- Modyfikowanie obiektów nielokalnych (czyli przekazywanie informacji do/z funkcji lub procedury drogą inną niż przez parametry lub nazwę funkcji) nosi nazwę efektów ubocznych.
- Efektów ubocznych należy unikać, a jeśli się ich używa — czynić to z rozwagą.
- Pascal udostępnia dwie metody przekazywania parametrów: przez wartość i przez nazwę.
- Przekazywanie przez wartość działa jednokierunkowo, tj. przekazuje informację od wywołującego do wywoływanej funkcji lub procedury.
- Przekazywanie przez nazwę umożliwia zwrócenie zmodyfikowanej wartości parametru do wywołującego.
- Przekazując do funkcji lub procedur większe struktury danych należy stosować przekazywanie przez nazwę. Skraca to czas wykonania programu i umożliwia uniknięcie przepełnienia stosu.

Programowanie na poważnie

W kolejnych kilku rozdziałach zajmiemy się realizacją prostego acz pouczającego zadania: masz oto napisać program wspomagający obsługę małej biblioteki. Oprócz wyjaśnienia zasygnalizowanego uprzednio tematu przechowywania większych ilości danych (i kilku innych rzeczy), spróbujemy na tym przykładzie przedstawić metodykę projektowania programu realizującego pewne rzeczywiste i praktyczne zadanie.

Sformułowanie naszego zadania brzmi „Napisz program wspomagający obsługę biblioteki szkolnej” i jest oczywiście zbyt ogólnikowe, by dało się skutecznie przetłumaczyć na program. Aby je uściślić, musimy sprecyzować wymagania użytkownika (czyli osoby obsługującej bibliotekę) oraz określić ograniczenia programu, czyli rzeczy, których z definicji nie będzie on w stanie zrobić i nad którymi nie będziemy się w ogóle zastanawiać.

Aby ustalić listę wymagań (będącą podstawą do zaprojektowania poszczególnych funkcji programu), musimy skonsultować się z jego przyszłym użytkownikiem (co w znacznej mierze sprowadza się do tłumaczenia, dlaczego tego czy tamtego się nie da zrobić). Załóżmy, że w wyniku pertraktacji z bibliotekarką ustaliliśmy, że nasz komputerowy katalog powinien umożliwiać:

- wprowadzenie z klawiatury i wyświetlenie na ekranie informacji o autorze i tytule książki;
- wyświetlenie listy książek uporządkowanych alfabetycznie według nazwiska autora lub tytułu;
- wyszukanie książki według pierwszych liter tytułu;
- sprawdzenie, czy książka jest na półce, czy też została wypożyczona (jeśli tak — to komu);
- wyszukanie książki najczęściej i najrzadziej wypożyczanej.

Dużo tego, ale jakoś sobie poradzimy. Mając do dyspozycji listę życzeń klienta możemy przystąpić do projektowania programu, którego zasadniczymi elementem będzie ustalenie sposobu przechowywania informacji (czyli struktur danych) oraz sposobu jej przetwarzania (czyli algorytmów). Ponieważ dane reprezentują pewne rzeczywiste obiekty (książki), musimy projektować nasz program pod kątem właściwości tych obiektów. Najpierw zatem utworzymy reprezentację danych, a następnie metody, za pomocą których będziemy je przetwarzać. Taka właśnie kolejność postępowania stosowana jest podczas projektowania większości programów.

Chociaż biblioteka zawiera wiele książek, wszystkie one mają pewien zestaw wspólnych cech. Kolejne uściślenie będzie polegało na opisaniu pojedynczej książki (rzecz jasna, należy pominąć informacje mające małe znaczenie dla użytkownika, jak np. liczba stron). Po przyjrzeniu się liście życzeń można stwierdzić, że w skład opisu pojedynczej książki powinny wchodzić następujące informacje:

- tytuł (ciąg znaków);
- nazwisko autora (ciąg znaków);
- nazwisko osoby, która książkę wypożyczyła (ciąg znaków);
- licznik wypożyczeń (liczba całkowita).

I tu pierwszy problem: jak reprezentuje się w programie ciąg znaków? W Turbo Pascalu służy do tego celu specjalny *typ łańcuchowy*, z angielska **`string`**. Zmienna typu łańcuchowego może przechowywać ciąg o długości do 255 znaków i deklarowana jest np. tak:

```
tytul : string;
```

Aby nie naruszać ciągłości tego rozdziału, dokładniejszy opis właściwości i możliwości typu łańcuchowego przeniesiemy nieco dalej (zobacz rozdział *Łańcuchy* na stronie 110), na razie zaś wrócimy do biblioteki. Skoro mamy już do dyspozycji właściwe typy, możemy zadeklarować w programie zmienne opisujące książkę:

```
var
  tytul, autor, wypozyczajacy : string;
  l_wypozyczen : integer;
```

Niestety, biblioteka zawiera znacznie więcej niż jedną książkę. Rzecz jasna, deklarowanie dla każdej pozycji oddzielnych zmiennych typu `Tytul1`, `Tytul2`... jest całkowicie bez sensu, wymagałoby bowiem utworzenia ogromnej liczby zmiennych (pomijając fakt, że nie zawsze wiadomo, z iloma obiektami będziemy mieli do czynienia). Problem ten został jednak już dawno rozwiązany: aby móc obsłużyć w programie dużą liczbę jednakowych (pod względem struktury) obiektów, musisz wykorzystać *tablice*.

Tak oto przechodzimy do bardzo ważnego zagadnienia, jakim są typy strukturalne.

Typy strukturalne, czyli jak przechować więcej danych

Z punktu widzenia użytkownika tablica jest po prostu ciągiem jednolitych „szufladek” przechowujących (to ważne) dane tego samego typu. Każda szufladka (którą w naszym przypadku można z grubsza utożsamiać z kartą katalogową) jest identyfikowana numerem — tzw. *indeksem* — który musi być typu porządkowego (trudno bowiem wyobrazić sobie element tablicy o numerze półtora). Tablice deklarowane są za pomocą słowa kluczowego **array** (tablica), zaś składnia deklaracji jest następująca:

```
nazwa_tablicy : array[indeks_dolny..indeks_górny] of typ;
```

W nawiasach kwadratowych (czy nie kojarzą Ci się z tablicą?) musisz umieścić dwie stałe określające najmniejszy i największy dopuszczalny indeks tablicy. Pamiętaj, że wartości użyte w deklaracji muszą być znane w momencie kompilacji programu (najczęściej podawane są jawnie, ewentualnie w postaci prostych wyrażeń dopuszczalnych w definicjach stałych). Pascal wymaga określenia rozmiaru tablicy w chwili kompilacji i nie dopuszcza użycia tzw. tablic dynamicznych (o rozmiarze określanym podczas wykonania) znanych z Basica. Oba indeksy mogą mieć dowolne wartości, ale muszą być tego samego typu porządkowego, a górny indeks nie może być mniejszy od dolnego. Zwykle tablice indeksuje się od 1, jak np. tablicę autorów książek:

```
var
  Autorzy : array[1..1000] of string[30];
```

ale całkowicie legalne są również deklaracje:

```
var
  Sinusy : array[0..359] of real;
  Odchylka : array[-10..10] of real;
```

Nie dość tego, tablice wcale nie muszą być indeksowane liczbami. Jeśli np. chcesz zliczyć częstość występowania w tekście poszczególnych znaków alfabetu, możesz użyć efektownej konstrukcji

```
var
    Liczniki : array['a'..'z'] of word;
```

Każdy z elementów takiej tablicy jest licznikiem o pojemności około 65 tysięcy „impulsów", adresowanym za pomocą odpowiedniego znaku alfabetu.

Aby odwołać się do elementu tablicy o danym numerze, musisz użyć składni

```
nazwa_tablicy[indeks]
```

gdzie `indeks` jest numerem żądanego elementu (stałą lub zmienną typu porządkowego o wartości zawartej pomiędzy dolnym a górnym zadeklarowanym indeksem). Aby zatem wyświetlić trzecią z kolei pozycję tablicy autorów, wystarczy użyć instrukcji

```
writeln(Autorzy[3]);
```

zaś instrukcja

```
Autorzy[5] := 'Ernest Hemingway';
```

ustali nazwisko autora piątej książki w katalogu. W przypadku naszej tablicy liczników znaków odwołania będą wyglądały następująco:

```
Inc(Liczniki[znak]);   { zwiększy licznik dla znaku o 1 }
writeln(Liczniki['a']; { wyświetli wartość}
                       { licznika znaków 'a' }
```

Próba użycia indeksu leżącego poza zadeklarowanym zakresem kończy się na ogół błędem wykonania, o ile opcja kontroli zakresów (***Options-Compiler-Range checking***) nie została wyłączona (**uwaga**: domyślnie opcja ta jest wyłączona!). W tym ostatnim przypadku odczyt z tablicy zwróci bezsensowną wartość, zaś zapis może mieć skutki opłakane (z zawieszeniem komputera włącznie), gdyż wartość zostanie wstawiona w miejsce przeznaczone na coś zupełnie innego.

Dwa główne obszary zastosowania tablic to bazy danych (czyli programy służące do przechowywania i zarządzania informacją opisującą rzeczywiste obiekty, np. książki w bibliotece, towary w hurtowni itp.) oraz oprogramowanie inżynierskie i naukowe, wykorzystujące tzw. wektory i macierze, czyli liniowe i prostokątne tablice liczb. Jak zapisać wektor (czyli liniową lub jednowymiarową tablicę liczb), już wiesz. Przykładowemu wektorowi złożonemu z dziesięciu liczb rzeczywistych

$$\left[x_1 \quad x_2 \quad \ldots \quad x_{10}\right]$$

będzie odpowiadała tablica

```
Wektor : array[1..10] of real
```

zaś dla prostokątnej tablicy (macierzy) o wymiarach *m* na *n*:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3n} \\ \dots & & \dots & & \dots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}$$

użyjemy deklaracji

```
Macierz : array[1..M, 1..N] of real
```

Jak widać, tworzenie tablic odpowiadających dwu- i więcejwymiarowym strukturom danych nie stanowi w Pascalu trudnego zadania. Ogólna składnia deklaracji tablicy wielowymiarowej wygląda tak:

```
nazwa_tablicy : array[zakres_1, zakres_2, ... zakres_n] of typ
```

gdzie `zakres_k` opisuje zakres indeksów dla *k*-tego wymiaru i ma formę `początek..koniec`. Dla tablic dwuwymiarowych pierwszy indeks oznacza numer wiersza, zaś drugi — numer kolumny (w pamięci komputera dane przechowywane są wierszami ułożonymi jeden za drugim). Chociaż Pascal nie narzuca ograniczenia na liczbę wymiarów tablicy (jedynymi ograniczeniami są łączny rozmiar elementów, nie mogący przekraczać 65520 bajtów, oraz zakres indeksów, nie mogący wykraczać poza zakres liczb całkowitych), tablic więcej niż dwuwymiarowych używa się bardzo rzadko. Tablice dwuwymiarowe stosowane są przede wszystkim w programach obliczeniowych (realizujących obliczenia tzw. rachunku macierzowego) i nie będziemy się tu nimi szczegółowo zajmować. Aby jednak nie poprzestawać na samym opisie, zademonstrujmy krótki przykład: oto program pozwalający na wprowadzenie z klawiatury wartości elementów macierzy o wymiarach 3 na 3 i wyświetlający największy wprowadzony element oraz sumę wszystkich elementów leżących na przekątnej i powyżej niej.

```
program Macierz3x3;
{ program demonstruje elementarne obliczenia macierzowe }
var
  t : array[1..3, 1..3] of real;  { macierz 3x3 }
  max, sum : real;  { tymczasowe maksimum i suma elementów }
  i, j : integer;   { liczniki wierszy i kolumn }

begin
{ wczytanie zawartości macierzy }
  for i := 1 to 3 do
    for j := 1 to 3 do
      begin
        write('Podaj element macierzy x[',i,',',j,']: ');
        readln(t[i,j]);
      end;
{ wyszukanie maksymalnego elementu }
  max := -1e12;   { pamiętaj o inicjalizacji maksimum }
  for i := 1 to 3 do
    for j := 1 to 3 do
      if t[i,j] > max then max := t[i,j];
{ obliczenie sumy elementów na i nad przekątną }
  sum := 0.0; { pamiętaj o wyzerowaniu sumy przed obliczaniem }
  for i := 1 to 3 do
```

```
      for j := i to 3 do
        sum := sum+t[i,j];
    writeln('Najwiekszy element: ', max:8:3);
    writeln('Suma elementow na i nad przekatna: ', sum:8:3);
    readln;
  end.
```

Jak nietrudno się domyślić, obsługa tablic (nie tylko dwuwymiarowych) wymaga intensywnego użycia pętli **for** (na ogół „przechodzimy” kolejno po elementach wiersza lub kolumny, znając numer pierwszego i ostatniego elementu). Powyższy program demonstruje dwie typowe operacje na tablicach:

- wyszukanie maksymalnego (minimalnego) elementu, polegające na sprawdzaniu, czy kolejny element nie jest większy (mniejszy) od aktualnej wartości maksimum (minimum), a jeśli tak — przyjęcie jego wartości jako nowego maksimum (minimum);
- obliczenie wartości sumy elementów na i ponad przekątną (w zbliżony sposób oblicza się sumę elementów pod przekątną, sumę wszystkich elementów dodatnich itp.).

Zauważ, że w trakcie obliczeń należy przeszukiwać zarówno wiersze, jak i kolumny, co realizowane jest za pomocą dwóch zagnieżdżonych (umieszczonych jedna w drugiej) pętli **for**. Zagnieżdżając pętle musisz pamiętać, by używały one **różnych liczników**, w przeciwnym przypadku bowiem wewnętrzna pętla będzie w niezamierzony sposób modyfikowała licznik pętli zewnętrznej (błąd ten jest tyleż pospolity, co nieprzyjemny). Zauważ ponadto, że wewnętrzna pętla rozpoczyna się nie od jedynki, lecz od bieżącego indeksu pętli zewnętrznej, dzięki czemu od razu omijamy elementy leżące pod przekątną (mające numer wiersza większy od numeru kolumny).

Mając do dyspozycji tablice możemy już zaprojektować struktury danych opisujące zawartość katalogu bibliotecznego. Dla uproszczenia załóżmy, że wszystkie dane będziemy przechowywali w pamięci komputera. Ponieważ ma ona ograniczoną pojemność, warto od razu przyjąć pewne założenia co do maksymalnej pojemności katalogu i poszczególnych fragmentów opisu książki. Dość oczywistym podejściem będzie ograniczenie długości pól przeznaczonych na tytuł i nazwisko autora oraz osoby wypożyczającej. Niech opis pojedynczej książki wygląda tak:

- tytuł: `string[30]`;
- nazwisko autora: `string[25]`;
- nazwisko wypożyczającego: `string[25]`;
- licznik wypożyczeń: `word`.

Wszystkie pola łącznie zajmują 85 bajtów, tak więc zakładając, że mamy do dyspozycji 62 kB (coś trzeba jeszcze zostawić na pomocnicze zmienne) jesteśmy w stanie opisać około 750 pozycji. *Skąd jednak wzięły się te 62 kilobajty, przecież w komputerze pamięci jest znacznie więcej* — mógłbyś zapytać. To prawda, jednak aby się do niej

dostać, potrzebne są nieco bardziej wymyślne rozwiązania, które zostaną omówione w rozdziale pt. *Więcej pamięci!* (strona 114).

Wyglądałoby na to, że można zakasać rękawy i wziąć się do roboty, czyli zadeklarować następujące tablice:

```
var
    Tytul : array[1..750] of  string[30];
    Autor : array[1..750] of string[25];
    Wypozyczajacy : array[1..750] of string[25];
    Licznik : array[1..750] of word;
```

Zauważ jednak, że opis pojedynczej książki rozproszony jest w czterech różnych tablicach. Co prawda poszczególne elementy opisu są „zjednoczone” wspólnym numerem (indeksem), jednak z punktu widzenia logiki rozwiązanie takie jest niespójne i mało eleganckie (chociaż da się wykorzystać). Rozproszenie danych opisujących pojedynczy obiekt w fizycznie odrębnych strukturach danych ma też tę wadę, że łatwiej jest przeoczyć jakiś fragment informacji lub pomylić numerację. Jak temu zaradzić? Przyjrzyjmy się naszej karcie katalogowej:

Tytuł	Emigranci
Autor	Mrożek Sławomir
Wypożyczający	Nowak Krzysztof
Licznik	78

Wszystkie dane opisujące fizyczny obiekt są tu połączone w jedną logicznie spójną całość, stanowiąc znacznie bardziej przekonujący i sensowny opis, aniżeli poprzednio (wyobraź sobie, jak ucieszyłby się bibliotekarz, gdyby kazano mu posługiwać się odrębnymi katalogami zawierającymi tytuły, nazwiska itd.). Pascalowym odpowiednikiem takiego opisu jest *rekord*.

O ile tablice pozwalały na pomieszczenie w jednej strukturze wielu danych tego samego typu, rekordy umożliwiają zamknięcie w całość danych różnych typów, nie pozwalając z kolei na indeksowany do nich dostęp. Składnia deklaracji zmiennej typu rekordowego jest następująca:

```
nazwa : record
  pole-1 : typ-1;
  pole-2 : typ-2;
  ...
  pole-n : typ-n;
end;
```

gdzie `pola` są po prostu nazwami elementów (pól) rekordu zawierających poszczególne dane, zaś `typy` określają ich typ. Przykładowy rekord odpowiadający naszej karcie katalogowej należałoby zadeklarować tak:

```
var
  Ksiazka : record
```

```
    Tytul : string[30];
    Autor : string[25];
    Wypozyczajacy : string[25];
    Licznik : word
  end;
```

Rzecz jasna, powyższa deklaracja nie rozwiązuje naszego problemu, pozwala bowiem na opisanie tylko jednej książki. Dlaczego jednak nie zorganizować rekordów w tablice? Oto przykład:

```
var
  Katalog : array[1..750] of record
    Tytul : string[30];
    Autor : string[25];
    Wypozyczajacy : string[25];
    Licznik : word
  end;
```

Przedstawiona powyżej struktura danych jest *tablicą rekordów* i stanowi (nie uwzględniając kilku drobnych poprawek) ostateczną formę, jaką będziemy się posługiwać w naszym programie. Podobnie, jak szuflada katalogu zawiera wiele jednakowych kart opisujących pojedyncze książki, tak i nasza tablica zawiera wiele jednakowych rekordów stanowiących elektroniczną formę takiego opisu. Tablice rekordów stanowią jedną z częściej wykorzystywanych struktur danych, umożliwiających efektywne tworzenie baz danych i przechowywanie informacji opisującej wiele jednakowych obiektów.

Jak zapamiętywać i odczytywać informację z rekordu? Zauważ, że każdy element opisu identyfikowany jest dwiema nazwami: nazwą samego rekordu (`Katalog[..]`) oraz odpowiedniego pola (np. `Tytul`). Aby odwołać się do konkretnego pola rekordu, musisz podać obydwie nazwy rozdzielając je kropką:

```
nazwa-rekordu.nazwa-pola
```

Chcąc więc wstawić do rekordu numer 45 nazwisko autora książki, wykorzystamy instrukcję

```
Katalog[45].Autor := 'Mrozek Slawomir';
```

zaś instrukcja

```
writeln(Katalog[29].Wypozyczajacy);
```

poinformuje nas, w czyim posiadaniu znajduje się obecnie książka opisana rekordem numer 29.

W przypadku wykonywania kilku operacji na polach tego samego rekordu używanie za każdym razem notacji „z kropką” jest nieco uciążliwe. Pascal umożliwia uproszczenie odwołań do pól rekordu przez wykorzystanie tzw. *instrukcji wiążącej* **with**:

```
with Katalog[120] do
  begin
    Tytul := 'Paragraf 22';
    Autor := 'Heller Joseph';
    Wypozyczajacy := 'Yossarian John';
    Licznik := 1254;
  end;
```

Instrukcja wiążąca wydatnie upraszcza zadania podobne do pokazanego wyżej, musisz jednak pamiętać o ujęciu wszystkich instrukcji operujących na polach rekordu w instrukcję złożoną **begin end** (w przeciwnym przypadku operacja będzie dotyczyła tylko pierwszego pola, a próba skompilowania kolejnych instrukcji najpewniej skończy się błędem). W zamian za to możesz wykorzystać instrukcję **with** do operowania na kilku rekordach naraz. Jeśli np. zadeklarowałeś następujące struktury:

```
DaneKlienta : array[1..1000] of record
  nazwisko : string[30]; { nazwisko klienta }
  SumaZakupow : real; { suma zakupów w danym roku }
  Saldo : real; { bieżące saldo klienta }
end;
Zakup : array[1..1000] of record
  IdKlienta : integer; { identyfikator klienta }
  IdTowaru : integer; { identyfikator towaru }
  Naleznosc : real; { należność za towar }
end;
```

to operacja

```
for i := 1 to LiczbaZakupow do
with DaneKlienta[50], Zakup[i] do
  if IdKlienta = 50 then
    begin
      SumaZakupow := SumaZakupow + Naleznosc;
      Saldo := Saldo – Naleznosc;
    end;
```

spowoduje przeszukanie tablicy zakupów i ustalenie sumy, jaką klient numer 50 zapłacił za wszystkie swoje zakupy oraz pomniejszenie jego salda o tę sumę. Zauważ, że opis klienta nie zawiera pola identyfikatora, który zastąpiony został indeksem w tablicy rekordów.

Aby skopiować wszystkie pola rekordu do innego rekordu tego samego typu, nie musisz w ogóle używać odwołań z kropką ani instrukcji **with**. Turbo Pascal umożliwia całościowe kopiowanie rekordów, np.

```
DaneKlienta[132] := DaneKlienta[1];
```

Niestety, operacji tego typu nie da się użyć do wprowadzenia zawartości rekordu z klawiatury ani wyprowadzenia jej na ekran. Operacje te należy wykonywać „pole po polu” (użyteczna jest tu instrukcja **with**), jak w poniższym przykładzie:

```
writeln('Pozycja w katalogu: ', i);
with Katalog[i] do
  begin
    writeln('Tytul: ', Tytul);  { wypisz zawartość }
                                { odpowiednich pól }
    writeln('Autor: ', Autor);
    writeln('Wypozyczajacy: ', Wypozyczajacy);
  end;
```

W analogiczny sposób można wprowadzać dane do rekordu.

W ten sposób przedstawiliśmy dwa najpopularniejsze typy strukturalne dostępne w Pascalu i zaprojektowaliśmy podstawową strukturę służącą do przechowywania informacji w naszym programie. W kolejnych rozdziałach zajmiemy się metodami przetwarzania informacji (czyli algorytmami i ich realizacją w postaci procedur i funkcji) oraz wynikającymi z tych zagadnień problemami.

Zapamiętaj

- Do przechowywania większych ilości danych służą typy strukturalne.
- Dwa najważniejsze typy strukturalne w Pascalu to tablice i rekordy.
- Tablice pozwalają na grupowanie danych tego samego typu. Zawarte w tablicy dane są dostępne poprzez indeksy.
- Tablice mogą być jedno- lub wielowymiarowe. Zakres indeksów tablicy jest dowolny, ale nie może przekraczać zakresu dozwolonego dla liczb całkowitych.
- Do obsługi tablic wykorzystuje się często pętle **for**.
- Rekordy grupują dane różnych typów. Zawarte w rekordzie dane są dostępne poprzez nazwy pól.

Struktura programu

Wiemy już, w jakiej formie będziemy przechowywać informację; kolejnym krokiem jest ustalenie sposobu jej przetwarzania odpowiedniego do używanych w programie struktur danych. Przede wszystkim musimy zatem ustalić, jakie operacje na danych program powinien wykonywać, a następnie zaprojektować odpowiadające im procedury lub funkcje (jako że programujemy porządnie, czyli strukturalnie — wykorzystując procedury i funkcje). Po skonsultowaniu się z listą życzeń (zobacz strona 75) stwierdzamy, że nasz program powinien umożliwiać:

- wprowadzenie zawartości wybranego rekordu z klawiatury;
- wyprowadzenie zawartości wybranego rekordu na ekran;
- zbadanie i zmianę zawartości pola `Wypozyczajacy` wybranego rekordu;
- wyszukanie rekordu zawierającego zadany ciąg znaków w polu `Tytul`;
- posortowanie tablicy rekordów według zadanego kryterium (tytuł lub nazwisko autora);
- wyszukanie rekordu o najmniejszej i największej zawartości pola `Licznik`.

Powyższa lista może zostać bezpośrednio przetłumaczona na zestaw podstawowych procedur tworzących program (rzecz jasna, trzeba będzie jeszcze dorzucić kilka procedur pomocniczych, no i oczywiście część operacyjną). Z formalnego punktu widzenia wymagany jest jeszcze schemat blokowy lub przynajmniej słowny zapis algorytmu, aby jednak uniknąć przerostu formy nad treścią, ograniczymy się do ogólnego schematu działania programu:

1. *wyświetl menu dostępnych operacji*
2. *w zależności od wybranej opcji:*

   wprowadź nowy rekord (dodanie książki do katalogu), lub

   zmień zawartość pola `Wypozyczajacy` *(wypożyczenie książki), lub*

   wyszukaj rekord zawierający dany tytuł (wyszukiwanie książki), lub

wyświetl wszystkie rekordy uporządkowane według tytułów lub nazwisk autorów, lub

zakończ pracę programu

3. *wróć do punktu (1)*

Jak widać, część operacyjna sprowadza się w zasadzie do wywoływania odpowiednich procedur stosownie do wybranej przez użytkownika funkcji (doskonałe miejsce dla instrukcji `case`). Ponieważ przedstawianie schematów poszczególnych procedur zajęłoby zbyt dużo miejsca, poprzestaniemy na zaprezentowaniu ich zapisu w Pascalu. Metoda, którą wykorzystamy do tworzenia naszego programu, jest typowym przykładem projektowania wstępującego: najpierw utworzymy zestaw odpowiednich procedur (realizujących poszczególne zadania podrzędne), a następnie powiążemy je w całość, czyli działający program.

Zanim jednak do tego dojdzie, zastanówmy się przez chwilę nad przekazywaniem danych do i z odpowiednich procedur i funkcji. W myśl dobrej praktyki nie będziemy do tego celu wykorzystywać efektów ubocznych, lecz stosownie zaprojektowane listy parametrów. Rozważmy jako przykład procedurę wprowadzania zawartości rekordu z klawiatury. Powinna ona przyjmować jako argument „pusty" rekord i zwracać tenże rekord wypełniony wpisanymi przez użytkownika danymi. Musimy więc wykorzystać mechanizm przekazywania parametru przez nazwę. Ale jak zadeklarować sam parametr? Nagłówek w postaci:

```
procedure Wprowadz(var r : record
  Tytul : string[30];
  Autor : string[25];
  Wypozyczajacy : string[25];
  Licznik : word);
```

wygląda co najmniej podejrzanie. Istotnie: próba skompilowania podobnej konstrukcji skończy się wyświetleniem komunikatu `Type identifier expected` (spodziewany identyfikator typu).

Rozwiązaniem tego problemu zajmiemy się w następnym rozdziale. Znajdzie się w nim również kilka słów o tak zwanych stałych symbolicznych.

Typy i stałe

Zajmiemy się obecnie dwoma spokrewnionymi ze sobą zagadnieniami: definiowaniem własnych typów oraz stałymi symbolicznymi. Główny obszar zastosowań stałych i typów związany jest z wykorzystaniem typów strukturalnych (dlatego mówimy o nich właśnie teraz); wykorzystanie definicji typów pozwala na ominięcie pewnych ograniczeń w przekazywaniu parametrów (zobacz koniec poprzedniego rozdziału). Poza tym używanie właściwie dobranych definicji typów i stałych polepsza czytelność i estetykę programu oraz zmniejsza ryzyko błędów.

Pokazany na końcu poprzedniego rozdziału przykład ilustruje niezręczną próbę przekazania do procedury parametru typu strukturalnego. Niestety, nasz zapis stanowi próbę przemycenia opisu (definicji) typu w miejscu do tego nie przeznaczonym: na liście parametrów formalnych procedury lub funkcji mogą znajdować się wyłącznie identyfikatory parametrów i odpowiadające im *identyfikatory* (nie definicje!) typów. Dopóki posługiwaliśmy się parametrami typu prostego, problem nie istniał, gdyż na liście parametrów umieszczałeś po prostu nazwę odpowiedniego typu. W przypadku typów strukturalnych sprawa się komplikuje. Nie dość, że na liście parametrów nie wolno umieścić definicji typu (jak więc zdefiniować parametr strukturalny?), to jeszcze identycznie wyglądające deklaracje tablic

```
var
  t1 : array[1..100] of real;
  t2 : array[1..100] of real;
```

wcale nie tworzą obiektów identycznego typu — każda z nich interpretowana jest jako definicja nowego typu, co utrudnia niektóre operacje (nie można np. dokonać przypisania całej tablicy `t2` do `t1`). Na dobitkę okazuje się, że zgodność typów wymagana jest również w przypadku przekazywania parametrów (co było zresztą łatwe do przewidzenia).

Wyjściem z tego impasu jest zdefiniowanie *identyfikatora typu*, czyli nazwy będącej synonimem danego typu strukturalnego. Identyfikator taki wykorzystamy następnie do deklarowania zmiennych i parametrów dokładnie tak samo, jak czynilibyśmy to z identyfikatorem typu prostego. Aby zdefiniować identyfikator typu, musisz użyć słowa kluczowego **type** (typ) w następującej konstrukcji:

```
type
  nazwa-typu = opis-typu
```

(zwróć uwagę, że w definicji typu wykorzystuje się znak równości, a nie dwukropek). `Nazwa-typu` określa definiowany identyfikator (i podlega wszystkim regułom dotyczącym identyfikatorów), zaś `opis-typu` jest takim samym opisem, jaki stosowany jest w deklaracjach zmiennych. Różnica polega tylko na tym, że w przypadku deklaracji zmiennej kompilator tworzy odpowiednią strukturę danych i rezerwuje dla niej miejsce w pamięci, natomiast zdefiniowanie typu powoduje jedynie utworzenie szablonu (nie jest tworzony żaden fizyczny obiekt), który będzie wykorzystywany w deklaracjach (dopiero wtedy kompilator będzie rezerwował miejsce na dane).

Przykładowa definicja typu rekordowego opisującego książkę (i deklaracja odpowiedniej tablicy rekordów) będzie więc miała postać

```
type
  Ksiazka = record { tak samo, jak w deklaracji zmiennej }
    Tytul : string[30];
    Autor : string[25];
    Wypozyczajacy : string[25];
    Licznik : word);
  end;

var
  Katalog : array[1..750] of Ksiazka;
```

Już na pierwszy rzut oka wygląda to ładniej (`Katalog` zawiera `Ksiazki`, a nie jakieś rekordy, których przeznaczenia należy się dopiero domyślać), no a poza tym możliwe jest wreszcie zadeklarowanie odpowiednich parametrów:

```
procedure Wprowadz(var r : Ksiazka);
```

Definiowanie nowych identyfikatorów typów ma zastosowanie głównie w programach posługujących się strukturalnymi reprezentacjami danych, chociaż możliwe jest również definiowanie nowych typów prostych (a nawet przedefiniowywanie typów standardowych). Oto przykład:

```
type
  float = extended; { real jeśli nie masz koprocesora }
```

Powyższa definicja tworzy nowy typ zmiennoprzecinkowy `float`, tożsamy z typem `extended` (dającym największą możliwą precyzję i zakres wartości). Jeśli program przeznaczony jest dla komputera pozbawionego koprocesora (i nie ma emulować tego ostatniego), wystarczy w miejsce typu `extended` wpisać `real`. Deklarując wszystkie liczby rzeczywiste w programie jako `float` uzyskasz w ten sposób możliwość łatwej adaptacji programu do warunków sprzętowych.

Zauważ, że w deklaracji zmiennej `Katalog` wymiar tablicy określony jest jawnie liczbą 750 (wynika ona z podzielenia wielkości dostępnej pamięci przez wielkość rekordu). Jest dość prawdopodobne, że do wartości tej będziesz się w programie odwoływał jeszcze kilkakrotnie (choćby podczas przeszukiwania katalogu lub usuwania jego elementów). Wyobraź sobie teraz, że postanowiłeś dodać do rekordu krótki opis książki w postaci łańcucha 20 znaków. Wpłynie to rzecz jasna na wielkość rekordu, a zatem i na dopuszczalną wielkość tablicy, która zmaleje teraz do około 600 pozycji. Po

dokonaniu zmiany będziesz musiał wyszukać w programie wszystkie miejsca, w których występowała liczba 750 i zastąpić ją liczbą 600 — jeśli o którymś zapomnisz, konsekwencje mogą być niezbyt przyjemne (do zawieszenia komputera włącznie). Drugi problem związany z użyciem liczby 750 (lub 600) pojawia się w kilka miesięcy po napisaniu programu i sprowadza się do faktu, że zwykle trudno sobie przypomnieć, co to za 750 i dlaczego akurat tyle.

Eleganckim sposobem na poradzenie sobie z opisanym problemem są *stałe symboliczne*. Podobnie jak definicja typu pozwala na utworzenie identyfikatora będącego synonimem definicji typu, tak definicja stałej symbolicznej umożliwia utworzenie identyfikatora będącego synonimem określonej wartości stałej (niekoniecznie liczbowej). Do definiowania stałych używane jest słowo kluczowe **const** (ang. ***const**ant* — stała), zaś składnia definicji wygląda następująco:

```
const
  nawa-stałej = wartość
```

Każde wystąpienie `nazwy-stałej` w programie zostanie podczas kompilacji zastąpione `wartością`. Podobnie jak w przypadku typów, zdefiniowanie stałej nie powoduje zarezerwowania miejsca w pamięci; w związku z tym musisz pamiętać, że stała symboliczna nie jest *l*-wartością i nie może się w programie znaleźć po lewej stronie operatora przypisania ani w miejscu parametru przekazywanego przez nazwę (pomijając oczywistą herezję, jaką jest próba zmiany wartości stałej).

Definicja stałej określającej maksymalną pojemność tablicy (i stosownie zmodyfikowana deklaracja tablicy rekordów) będzie więc wyglądała następująco:

```
const
  POJEMNOSC_KATALOGU = 750;
var
  Katalog : array[1..POJEMNOSC_KATALOGU] of Ksiazka;
```

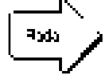
Zauważ, że identyfikator stałej zapisany jest dużymi literami (jest to istotne tylko dla programisty — Turbo Pascal nie odróżnia dużych liter od małych). Konwencja ta — pozwalająca łatwo odróżnić *STAŁE* od *zmiennych*, jest dość powszechnie stosowana przez programistów i warto ją sobie przyswoić.

Jeśli nie chce Ci się przeliczać wartości naszej stałej po każdej zmianie struktury rekordu (czego z drugiej strony nie powinieneś robić zbyt często, bo dowodzi to złego zaprojektowania programu), możesz użyć następującej definicji:

```
const
  MAX_PAMIEC = 63000; { maksymalna wielkosc katalogu }
                      { w bajtach }
  POJEMNOSC_KATALOGU = MAX_PAMIEC div SizeOf(Ksiazka);
```

Jak nietrudno się domyślić, raz zdefiniowanej stałej można użyć w kolejnych definicjach. Co jednak znacznie ciekawsze, stała może zostać zdefiniowana za pomocą wyrażenia zawierającego inne stałe i niektóre operatory (wyrażenie musi dać się obliczyć w trakcie kompilacji, nie może więc zawierać identyfikatorów zmiennych i funkcji

bibliotecznych; dokładniejsze informacje na ten temat znajdziesz w systemie pomocy). W naszym przypadku wartość stałej zdefiniowana jest jako wynik (całkowity) podzielenia wielkości dostępnej pamięci (zdefiniowanej inną stałą) przez rozmiar rekordu opisującego książkę. W tym momencie możesz przestać przejmować się jakimikolwiek liczbami.



Stosowanie stałych symbolicznych i definicji typów stanowi jedną z podstaw dobrej praktyki programowania. Odpowiednio użyte, mechanizmy te znacznie poprawiają czytelność programu i zmniejszają jego podatność na błędy w trakcie pisania i poprawiania.

Co prawda przed chwilą powiedziano, że stała zdefiniowana słowem kluczowym **const** nie jest *l*-wartością, okazuje się jednak, że słowa tego można użyć również do definiowania... zmiennych. Chodzi tu o tzw. *zmienne z wartością początkową* (*zmienne predefiniowane*, ang. *typed constant*). Zmienna taka jest normalną zmienną — tyle, że uzyskuje wartość już w momencie deklaracji (definicji). Składnia definicji zmiennej z wartością początkową jest następująca:

```
const
  nazwa-zmiennej : typ = wartość-początkowa
```

Zmienne predefiniowane pozwalają na łatwe inicjalizowanie danych w programie. Oto przykłady:

```
const
  Liczba_ksiazek : word = 0; { na początku }
                             { katalog jest pusty }
  Liczniki : array[1..4] of word = (0, 0, 0, 0);
                             { inicjalizacja tablicy }
```

Wiadomości przedstawione w tym rozdziale powinny pozwolić nam na napisanie podstawowej wersji naszego programu. Tym jednak zajmiemy się już w następnym rozdziale.

Zapamiętaj

- Deklarowanie parametrów typu strukturalnego możliwe jest wyłącznie przy użyciu identyfikatorów typów.
- Do zdefiniowania identyfikatora typu używane jest słowo kluczowe **type**.
- Definiowanie stałych symbolicznych, czyli identyfikatorów przechowujących wartości stałe umożliwia słowo kluczowe **const**.
- Słowo **const** pozwala również na deklarowanie zmiennych połączone z nadawaniem im wartości początkowych.
- Używanie definicji typów i stałych symbolicznych poprawia czytelność programu, jego uniwersalność i odporność na błędy.

Do dzieła!

Po tym nieco przydługim teoretyzowaniu większość Czytelników zapewne świerzbią już ręce, żeby wziąć się do programowania. W niniejszym rozdziale spróbujemy przedstawić podstawowe elementy składowe naszego programu i powiązać je w działającą całość. Zaczniemy od rzeczy podstawowych, czyli zdefiniowania struktur danych, na których program będzie operował.

```
program Biblioteka;
{ Prosty program do obsługi biblioteki }

uses Crt; { moduł obsługujący klawiaturę i monitor }

type
  string30 = string[30]; { potrzebne do przekazywania }
                         { parametrów }
  string25 = string[25]; { typu łańcuchowego do procedur }
  Ksiazka = record       { rekord opisujący książkę }
    Tytul : string30;
    Autor : string25;
    Wypozyczajacy : string25;
    Licznik : word;
  end;

const
  MAX_PAMIEC = 63000; { maksymalna wielkość katalogu }
                      { w bajtach }
  POJEMNOSC = MAX_PAMIEC div SizeOf(Ksiazka);

var
  Katalog : array[1..Pojemnosc] of Ksiazka; { właściwy }
                                            { katalog }
  LbPoz : integer; { liczba pozycji w katalogu }
```

Podstawowym warunkiem użyteczności katalogu jest możliwość wprowadzania do niego informacji (z klawiatury) i wyprowadzania jej (na ekran/drukarkę). Ponieważ podstawową jednostką organizacyjną katalogu jest rekord, powinniśmy zdefiniować procedury pozwalające na wprowadzenie i wyprowadzenie jego zawartości. W myśl tego, co powiedzieliśmy poprzednio, procedury te powinny operować na parametrze odpowiedniego typu rekordowego:

```
procedure WprowadzDane(var r : Ksiazka);
{ wprowadza pojedynczy rekord z klawiatury }

begin
  with r do  { wprowadź poszczególne pola rekordu }
    begin
      write('Tytul: '); readln(Tytul);
      write('Autor: '); readln(Autor);
      write('Wypozyczajacy: '); readln(Wypozyczajacy);
    end;
end;

procedure WypiszDane(r : Ksiazka);
{ wyprowadza zawartość rekordu na ekran }

begin
  with r do  { wyprowadź poszczególne pola rekordu }
    begin
      write('Tytul: '); writeln(Tytul);
      write('Autor: '); writeln(Autor);
      if Wypozyczajacy = '' then { nikt nie wypożyczył }
        writeln('Ksiazka znajduje sie na polce.')
      else { pole zawiera nazwisko wypożyczającego }
        write('Wypozyczajacy: '); writeln(Wypozyczajacy);
    end;
  writeln;
end;
```

Mimochodem przyjęliśmy konwencję, w myśl której książce nie wypożyczonej odpowiada puste pole `Wypozyczajacy` (co jest dość naturalne). Powyższe definicje wykorzystamy do utworzenia kilku dodatkowych procedur pozwalających na wprowadzenie opisu nowej książki do katalogu oraz wyprowadzenie całej zawartości katalogu na ekran (co w przypadku większej liczby książek jest raczej mało sensowne, chyba że zamiast ekranu użyje się drukarki). Przy okazji warto też zdefiniować procedurę pozwalającą na usunięcie książki z katalogu, co najprościej zrobić przez przepisanie ostatniego elementu katalogu w miejsce elementu usuwanego.

```
procedure DodajKsiazke;
{ dodaje nową książkę do katalogu }

begin
  Inc(LbPoz);  { dodajemy nową książkę }
  writeln('Nowa pozycja w katalogu: ', LbPoz);
  WprowadzDane(Katalog[LbPoz]);  { wprowadź dane }
                                 { nowej książki }
    Katalog[LbPoz].Licznik := 0; { książka nie była }
                                 { wypożyczana }
end;
```

```
procedure UsunKsiazke(Numer : integer);
{ usuwa książkę z katalogu }

begin
  Katalog[Numer] := Katalog[LbPoz]; { zamazuje daną pozycję }
  Dec(LbPoz);  { usunięto książkę }
end;

procedure WypiszKatalog;
{ wypisuje całą zawartość katalogu na ekran }

var
  i : integer;

begin
  for i := 1 to LbPoz do
    begin
      writeln('Pozycja katalogu nr ', i,':');
      WypiszDane(Katalog[i]);
    end;
end;
```

Porządny program powinien również zapewniać możliwość zmiany opisu książki; konstrukcję odpowiedniej procedury pozostawiam Ci jako ćwiczenie (zobacz też poniżej).

Kolejną sprawą jest obsługa wypożyczenia i zwrotu książki. W tym celu należy po pierwsze odszukać wybraną książkę (wedle wcześniejszych założeń — według tytułu), a następnie, w zależności od potrzeb, wprowadzić, sprawdzić lub usunąć nazwisko wypożyczającego. Jeśli chodzi o wyszukiwanie książki, to sensownym wydaje się zaprojektowanie funkcji, której argumentem będzie zadany fragment tytułu, a wartością zwracaną — numer książki w katalogu lub –1, jeśli książki nie znaleziono (numery książek są zawsze dodatnie, a więc liczba –1 nigdy nie będzie opisywała rzeczywiście istniejącej w katalogu książki). Ponieważ katalog może zawierać kilka książek o tym samym tytule, musimy zaprojektować funkcję wyszukującą tak, by umożliwiała znalezienie wszystkich „podejrzanych". W tym celu wyposażymy ją w możliwość szukania od zadanej pozycji:

```
function Szukaj(Tekst : string; Pozycja : integer) : integer;
{ wyszukuje książkę o tytule zawierającym Tekst }
{ szuka od zadanej pozycji katalogu }

var
  i : integer;

begin
  i := Pozycja;         { szukaj od zadanej pozycji }
  { sprawdź, czy tytuł zawiera tekst }
  { i czy przeszukano cały katalog }
```

```
while (Pos(Tekst, Katalog[i].Tytul) = 0) and (i <= LbPoz) do
    Inc(i);
  if i <= LbPoz then      { znaleziono odpowiednią }
                          { pozycję katalogu }
    Szukaj := i
  else                    { nie znaleziono pozycji w katalogu }
    Szukaj := -1;
end;
```

Zauważ, że istota przeszukującej tablicę pętli `while` skupia się w warunku, zaś jej treścią jest jedynie zwiększenie indeksu tablicy. Do sprawdzenia, czy tytuł zawiera dany ciąg znaków użyliśmy funkcji `Pos`, zwracającej pozycję, na której ciąg został znaleziony w zadanym łańcuchu (lub zero, jeśli go nie znaleziono). Przeszukiwanie kończy się w momencie, gdy indeks tablicy zrówna się z numerem ostatniej książki w katalogu.

Wyszukanie wszystkich książek o zadanym tytule (i wyświetlenie informacji o nich) jest już proste:

```
procedure SprawdzKsiazki(Tytul : string);
{ wyprowadza dane książek o zadanym tytule }

var
  i : integer;
  Tytul : string;

begin
  write('Podaj tytul szukanej ksiazki: ');
  readln(Tytul);
  i := 1;
  repeat
    i := Szukaj(Tytul, i); { znajdź kolejne }
                           { wystąpienie tytułu }
    if i <> -1 then        { znaleziono książkę }
      begin
        writeln('Pozycja katalogu nr ', i,':');
        WypiszDane(Katalog[i]);
        Inc(i);            { przejdź do następnej pozycji }
      end;
  until i = -1             { nie znaleziono więcej książek }
end;
```

Powyższa procedura cyklicznie wywołuje funkcję `Szukaj`, nakazując jej za każdym razem rozpoczęcie przeszukiwania od ostatnio znalezionej książki (dlaczego?) i wypisując dane znalezionych pozycji za pomocą procedury `WypiszDane`. W tym momencie łatwo już ustalić, czy książka została wypożyczona, a jeśli tak — to komu (w ramach ćwiczeń sugeruję Ci napisanie podobnych funkcji, umożliwiających ustalenie, które książki zostały wypożyczone osobie o podanym nazwisku).

Właściwą obsługą wypożyczenia lub zwrotu zajmuje się przedstawiona poniżej procedura `Wypozycz`. Zaznaczenie książki jako wypożyczonej odbywa się przez wpi-

sanie nazwiska wypożyczającego do odpowiedniego pola rekordu i zwiększenie licznika wypożyczeń. W chwili zwrotu pole `Wypozyczajacy` jest zerowane.

```
procedure Wypozycz(var r : Ksiazka);
{ obsługuje wypożyczenie/zwrot książki }

begin
  writeln;
  writeln('Biezace dane ksiazki:');
  WypiszDane(r); { wypisz informacje o książce dla kontroli }
  writeln;
  writeln('Wprowadz nazwisko wypozyczajacego.');
  writeln('W przypadku zwrotu ksiazki nacisnij Enter.');
  with r do { wprowadź nazwisko lub wyczyść pole rekordu }
    begin
      readln(Wypozyczajacy);
      if Wypozyczajacy <> '' then
        Inc(LbPoz); { przy wypożyczeniu }
                    { zwiększ licznik wypożyczeń }
    end;
end;
```

Powyższą procedurę można wywołać np. podczas sprawdzania danych książek, czyli z procedury `SprawdzKsiazki`. Wymaga to niewielkiej modyfikacji tej ostatniej:

```
{ ... }
  writeln('Pozycja katalogu nr ', i,':');
  WypiszDane(Katalog[i]);
  writeln('Obsluga wypozyczen: nacisnij "W"');
  writeln('Nastepna pozycja: nacisnij dowolny klawisz.');
  if UpCase(ReadKey) = 'W' then
     Wypozycz(Katalog[i]);
  Inc(i); { przejdz do nastepnej pozycji }
{ ... }
```

Jeśli podczas przeglądania wybranych pozycji bibliotekarz zechce wypożyczyć książkę lub przyjąć jej zwrot, powinien nacisnąć klawisz W, co wywoła procedurę `Wypozycz`. Obsługą naciśnięcia klawisza zajmuje się „importowana” z modułu `Crt` (zobacz str. 121) funkcja `ReadKey`, zaś funkcja `UpCase` uniezależnia program od wielkości liter (w lub W).

Statystykę wypożyczeń obsługuje dość obszerna, chociaż bardzo prosta procedura `Statystyka`. Oblicza ona najmniejszą, największą i średnią wartość licznika wypożyczeń dla całego katalogu. Wykorzystana w niej metoda wyszukiwania najmniejszej i największej wartości w tablicy jest bardzo typowa i szeroko stosowana, a polega na badaniu, czy kolejny element tablicy nie jest mniejszy (większy) od aktualnego minimum (maksimum); jeśli tak, jest on przyjmowany jako nowe minimum (maksimum).

```
procedure Statystyka;

var
  i : integer;  { licznik pętli }
  min, max, sum : longint;  { minimum, maksimum i suma }
  poz_min, poz_max : integer; { pozycje minimum i maksimum }

begin
  min := 1000000; { inicjalizuj minimum, maksimum i sumę }
  max := 0;
  sum := 0;
  for i := 1 to LbPoz do
    with Katalog[i] do
      begin
        if Licznik > max then { znaleziono nowe maksimum }
          begin
            max := Licznik; { zapamiętaj nowe maksimum }
            poz_max := i;    { i numer odpowiedniej pozycji }
          end;
        if Licznik < min then { to samo dla minimum }
          begin
            min := Licznik;
            poz_min := i;
          end;
        sum := sum + Licznik; { zwiększ licznik wypozyczeń }
      end;
  writeln('Statystyka wypozyczen:'); { wypisz wartości }
                                     { statystyk }
  writeln('Srednia wypozyczen na jedna ksiazke: ',
    sum/Licznik:6:0);
  writeln('Ksiazka najczesciej wypozyczana (', max,
    ' wypozyczen):');
  WypiszDane(Katalog[poz_max]);
  writeln('Ksiazka najrzadziej wypozyczana (', min,
    ' wypozyczen):');
  WypiszDane(Katalog[poz_min]);
end;
```

Pozostała nam do rozwiązania sprawa alfabetycznego uporządkowania zawartości katalogu. Zadanie to, będące jednym z klasycznych zadań programowania, nosi nazwę *sortowania*. Liczba znanych obecnie algorytmów sortowania jest dość spora; my zajmiemy się jednym z najprostszych — tak zwanym *sortowaniem przez proste wybieranie* (ang. *selection sort*), które, chociaż niezbyt efektywne, jest łatwe do zrozumienia i interpretacji.

Załóżmy, że musimy uporządkować (od wartości najmniejszej do największej) tablicę zawierającą pięć liczb całkowitych. W pierwszym kroku wyszukujemy najmniejszą liczbę w tablicy i zamieniamy ją miejscami z liczbą znajdującą się na pierwszej pozycji. W tym momencie najmniejsza wartość w tablicy jest już zlokalizowana i ustawiona na właściwym miejscu, „odcinamy” więc pierwszą pozycję tablicy i powtarzamy wyszukiwanie minimum dla pozostałych pozycji (od 2 do 5). Po znalezieniu minimum zamieniamy je miejscami z liczbą na pozycji drugiej, czego efektem jest ustalenie

dwóch pierwszych pozycji uporządkowanej tabeli. Po „odcięciu" pierwszych dwóch pozycji (już uporządkowanych) powtarzamy sortowanie dla pozycji 3 do 5 i tak dalej, aż do chwili, kiedy pozostanie nam do przeszukania ostatnia pozycja tablicy (oczywiście zawierająca największą liczbę, „zepchniętą" na sam koniec). A oto przykład:

Przed sortowaniem (krok 0)	*W trakcie sortowania*			
	krok 1	*krok 2*	*krok 3*	*krok 4*
4	1 (4)	1	1	1
1	**4 (1)**	2 (4)	2	2
3	**3**	**3**	3 (3)	3
5	**5**	**5**	**5**	4 (5)
2	**2**	**4 (2)**	**4**	**5 (4)**

Kolorem czarnym zaznaczono zakres przeszukiwania tablicy (jak widać, w kolejnych krokach zmniejsza się on o 1), zaś w nawiasach umieszczono wartości, które znajdowały się na danej pozycji przed zamianą, tj. 4 (1) oznacza, że na danej pozycji znajdowała się liczba 1, zamieniona następnie z liczbą 4.

Jak widać, nasz algorytm jest całkiem prosty, choć okupione jest to efektywnością (bardziej ambitnym Czytelnikom polecam przestudiowanie innych algorytmów sortowania, z których najczęściej stosowanym i chyba najefektywniejszym jest tzw. *szybkie sortowanie* (ang. *quicksort*), zilustrowane m.in. dostarczanym wraz z Turbo Pascalem programem QSORT.PAS). Niestety, sprawę komplikuje nieco wymaganie sortowania według tytułów lub nazwisk autorów. Jak już się zapewne zorientowałeś, wykorzystywane w algorytmie wyszukiwanie najmniejszej wartości polega na porównywaniu zawartości odpowiedniej komórki tablicy z bieżącą wartością minimum. Aby posortować katalog według tytułów, należy porównywać pola `Tytul` odpowiednich rekordów; sortowanie według nazwisk autorów wymaga porównywania pól `Autor`. Czyżby więc należało napisać dwie oddzielne procedury sortujące? Na szczęście nie, chociaż bardzo wielu początkujących programistów radzi sobie właśnie w ten sposób (czego absolutnie nie należy polecać!). Zauważ, że w obu przypadkach mechanizm sortowania jest identyczny, zaś różnica sprowadza się wyłącznie do porównywania różnych pól rekordów. Wystarczy zatem napisać odpowiednie funkcje porównujące dwa rekordy według pól `Autor` i `Tytul`, a następnie wywoływać je w zależności od aktualnego kryterium sortowania. Zwyczajowo funkcje takie zwracają wartość –1, 0 lub 1 odpowiednio do tego, czy pierwszy z porównywanych obiektów jest mniejszy, równy lub większy od drugiego. W naszym przypadku — ponieważ sortujemy obiekty rosnąco — wystarczy informacja, czy pierwszy rekord jest mniejszy od drugiego, czy też nie. Oto przykład funkcji porównującej rekordy według zawartości pola `Tytul` (drugą funkcję napisz sam):

```
function PorownajTytul(r1, r2 : Ksiazka) : integer;
{ porównuje rekordy według pola Tytul }

begin
  if r1.Tytul < r2.Tytul then PorownajTytul := -1
  else PorownajTytul := 1;
```

```
end;
```

Sama procedura sortująca powinna wykonywać algorytm opisany powyżej, wywołując odpowiednią funkcję porównującą w zależności od zadanego przez użytkownika kryterium. Na obecnym etapie znajomości Pascala jesteśmy w stanie zrealizować takie wywołanie w oparciu o instrukcję warunkową, jednak „prawdziwe" programy wykorzystują nieco inne rozwiązanie, oparte o tak zwane *parametry proceduralne*. Omawianie tego zagadnienia wykracza poza ramy niniejszej książki, jednak w skrócie metoda ta polega na przekazaniu procedurze sortującej wskaźnika opisującego położenie funkcji porównującej (jej adresu). Podstawiając pod parametr adresy różnych funkcji jesteśmy w stanie elegancko zmieniać kryteria sortowania.

A oto treść procedury:

```
procedure Sortuj(Kryterium : integer);
{ sortuje zawartość katalogu według zadanego kryterium }

var
  i, j : integer;
  CzyWiekszy : integer;
  Pomoc : Ksiazka;

begin
  for i := 1 to LbPoz-1 do { wykonaj LbPoz-1 przebiegów }
    for j := i to LbPoz do
      begin
        if Kryterium = 1 then  { porównanie wg tytułu/autora }
          CzyWiekszy := PorownajTytul(Katalog[i], Katalog[j])
        else
          CzyWiekszy := PorownajAutor(Katalog[i], Katalog[j]);
        if CzyWiekszy = 1 then { trzeba zamienić elementy }
          begin
            Pomoc := Katalog[i]; { zachowaj pierwszy rekord }
            Katalog[i] := Katalog[j]; { wstaw drugi }
                                      { do pierwszego }
            Katalog[j] := Pomoc;      { wstaw zachowany }
                                      { do drugiego }
          end;
      end;
end;
```

Jak widać, procedura wykonuje cykliczne porównania i ewentualne zamiany w dwóch zagnieżdżonych pętlach, z których zewnętrzna wyznacza kolejne kroki algorytmu, zaś wewnętrzna obsługuje przeglądanie pozostałych jeszcze do uporządkowania rekordów. Sama zamiana elementów wykorzystuje typowy schemat „podpierający się" pomocniczym rekordem służącym do tymczasowego przechowania danych. Zauważ wreszcie, że aby posortować rekordy malejąco, wystarczy zmodyfikować warunek

```
if CzyWiekszy = 1
```

zastępując wartość 1 wartością –1.

Jak nietrudno się domyślić, operacja sortowania powinna poprzedzać wyprowadzenie zawartości katalogu na ekran lub drukarkę. Procedura kojarząca obydwie operacje może wyglądać następująco:

procedure WypiszKatalog;

```
var
  Kryterium : integer;

begin
  write('Sortowanie wedlug tytulow (1)',
        ' lub nazwisk autorow (2)');
  readln(Kryterium);
  Sortuj(Kryterium-1); { parametr powinien być 0 lub 1 }
  Wypisz; { wypisz posortowany katalog }
  readln; { i pozwól go przeczytać }
end;
```

W ten sposób zakończyliśmy budowę zasadniczych klocków, z których składać się ma nasz program. Pozostaje jeszcze napisanie ładnego menu:

```
function Menu : integer;

var
  c : char;

begin
  ClrScr; { wyczyść ekran - wymaga włączenia modułu Crt }
  writeln('Program biblioteczny KATALOG, wersja 1.0, 1996');
  writeln;
  writeln('Wybierz funkcje:');
  writeln('1: Wyszukiwanie pozycji wg tytulu',
          ' i obsluga wypozyczen');
  writeln('2: Wyswietlenie wszystkich pozycji');
  writeln('3: Dodanie ksiazki do katalogu');
  writeln('5: Statystyka wypozyczen');
  writeln('0: Koniec pracy');
  writeln;
  writeln;
  repeat         { czytaj klawiaturę, }
    c := ReadKey { również wymaga włączenia modułu Crt }
  until c in ['0'..'5']; { ignoruj pozostałe znaki }
  Menu := ord(c)-ord('0');
end;
```

Sama część operacyjna programu jest również bardzo typowa. Wykorzystuje ona pętlę `repeat`, cyklicznie wyświetlającą menu dostępnych operacji. Kod zwrócony przez funkcję `Menu` jest wykorzystywany do wywołania odpowiedniej operacji w instrukcji `case`. Zwrócenie przez funkcję `Menu` wartości 0 powoduje zakończenie działania pętli, a tym samym całego programu.

```
begin { część operacyjna }
  repeat
```

```
      Wybor := Menu; { wybierz opcję z menu }
      case Wybor of
        1 : SprawdzKsiazki;
        2 : begin
              WypiszKatalog;
              readln;
            end;
        3 : DodajKsiazke;
        5 : Statystyka;
      end;
    until Wybor = 0;        { opcja 0 = koniec pracy }
  end. { program Biblioteka }
```

Jak zdążyłeś zapewne zauważyć, program nie umożliwia usunięcia książki z katalogu. Przeznaczona jest do tego brakująca czwarta pozycja menu, której zaprogramowanie proponuję Ci w ramach ćwiczeń.

Po poskładaniu opisanych w tym rozdziale fragmentów w jedną całość (najlepiej w kolejności, w jakiej były cytowane) uzyskujesz gotowy program. Niestety, pełnię szczęścia zakłócają dwa mankamenty, z których co najmniej jeden jest bardzo poważny. Okazuje się mianowicie, że nasz katalog istnieje dopóty, dopóki użytkownik nie wybierze opcji „Koniec pracy" — kolejne uruchomienie programu wiąże się z koniecznością wpisania wszystkich książek od nowa (co jest niedopuszczalne!!). No i sama pojemność katalogu też mogłaby być większa...

Rozwiązaniem naszych problemów zajmiemy się już za chwilę, w dwóch kolejnych rozdziałach traktujących o plikach oraz zmiennych dynamicznych.

Pliki, czyli jak uchronić dane przed zgubą

Pamięć operacyjna, wykorzystywana do przechowywania danych przetwarzanych przez program, ma dwie zasadnicze wady: ograniczoną pojemność i ulotność (to poetyckie słowo oznacza po prostu, że zawartość pamięci jest tracona w chwili wyłączenia zasilania). Komputer umożliwiający przechowywanie danych wyłącznie podczas włączenia do sieci byłby raczej mało użyteczny, dlatego też projektanci sprzętu opracowali szereg urządzeń — tzw. *pamięci masowych* — pozwalających na trwałe przechowywanie danych. Rolę pamięci masowej w komputerze osobistym pełnią dyskietki oraz dyski twarde (oba rozwiązania wykorzystują identyczną metodę zapisu i różnią się rozwiązaniami technologicznymi). Z logicznego punktu widzenia, dane zapisywane na dyskach organizowane są w *pliki*, a te z kolei przechowywane są w *katalogach*. Całością zarządza system operacyjny, z którego usług korzystają programy użytkowe.

Rzecz jasna, również i Turbo Pascal dysponuje możliwością korzystania z plików. Z trzech dostępnych w Pascalu rodzajów plików — elementowych (jednorodnych), tekstowych i amorficznych — omówimy dwa pierwsze, mające największe zastosowanie w praktyce.

Na początek nieco teorii. Sam plik jest pewną strukturą danych zapisaną na dysku i identyfikowaną za pomocą nazwy (ściślej — ścieżki dostępu). Dane przechowywane w pliku mogą mieć reprezentację binarną (taką samą, jak w pamięci komputera) lub tekstową (taką, jaka używana jest do wprowadzania informacji z klawiatury i wyprowadzania jej na ekran monitora lub drukarkę). Reprezentacjom tym odpowiadają w Pascalu pliki elementowe oraz tekstowe.

- Pliki elementowe przechowują dane w postaci binarnej, zaś pliki tekstowe — w postaci wierszy tekstu zakończonych znakami końca wiersza. Zawartość plików elementowych jest na ogół nieczytelna dla użytkownika, natomiast treść pliku tekstowego daje się łatwo odczytać i zinterpretować. Z drugiej strony, binarna reprezentacja danych jest bardziej zwarta i oszczędna.

- Wszystkie dane przechowywane w plikach elementowych muszą być tego samego typu (prostego lub strukturalnego). Pliki tekstowe, wykorzystujące znakowe (sformatowane) reprezentacje danych, mogą być użyte do przechowywania mieszanych typów danych (np. tekstów i liczb), gdyż wszelka informacja przechowywana jest w nich w postaci tekstowej. Pliki tekstowe umożliwiają również formatowanie zapisu i korzystanie z procedur `readln` i `writeln`, które są niedostępne dla plików elementowych.
- Pliki elementowe umożliwiają tzw. dostęp swobodny — w dowolnym momencie można odwołać się do dowolnego elementu pliku. Pliki tekstowe są plikami o dostępie sekwencyjnym, co oznacza, że aby dostać się do wybranego elementu pliku, należy przeczytać wszystkie elementy znajdujące się przed nim.

Odwołania do pliku (zapisy, odczyty i inne operacje) realizowane są przez wywołanie odpowiednich funkcji systemu operacyjnego, który z kolei posługuje się liczbowymi identyfikatorami plików (korzystanie z nazw byłoby niewygodne). Również program pascalowy nie odwołuje się do plików „bezpośrednio”, lecz poprzez tak zwane *zmienne plikowe*, czyli złożone struktury danych reprezentujące fizyczne pliki zapisane na dysku. Ogólny schemat operacji plikowej w Pascalu obejmuje cztery etapy:

- skojarzenie zmiennej plikowej z odpowiednim plikiem (znajdującym się na dysku lub nowo tworzonym);
- otwarcie pliku, przygotowujące go do zapisywania lub odczytywania informacji;
- jedna lub więcej operacji zapisu lub odczytu danych;
- zamknięcie pliku i przerwanie skojarzenia pomiędzy zmienną plikową i plikiem.

Samą zmienną plikową deklaruje się w sposób następujący:

```
nazwa : file of typ { dla pliku elementowego }
nazwa : text { dla pliku tekstowego }
```

gdzie `typ` określa typ elementu składowego pliku i może być dowolnym identyfikatorem typu prostego lub strukturalnego (z wyjątkiem typu plikowego i obiektowego). Zauważ, że pojedynczym elementem pliku elementowego jest nie bajt, lecz właśnie obiekt zadanego typu (co jest dość logiczne). Dlatego też do pliku rekordów (drugi przykład poniżej) możesz wpisywać wyłącznie całe rekordy (zapisywanie lub odczytywanie pojedynczych pól jest niewykonalne), a jego długość będzie zawsze równa wielokrotności rozmiaru pojedynczego rekordu. Podobnie deklaracja pliku, którego elementami składowymi są tablice liczb całkowitych, zmusi Cię do zapisywania i odczytywania całych tablic, gdyż odwołanie się do pojedynczego elementu tablicy będzie niemożliwe. Ten sam plik można oczywiście otworzyć jako plik liczb całkowitych, co pozwoli nam na odczytywanie pojedynczych wartości. W przypadku, gdy plik przechowuje jednorodne dane, deklaracja zmiennej plikowej wykorzystuje na ogół elementy składowe odpowiedniego typu prostego. Dla baz danych, złożonych z rekordów (przechowujących dane różnych typów), jedynym wyjściem jest deklaracja pliku rekordów.

Oto kilka przykładowych deklaracji:

```
var
  Probki : file of real; { plik liczb rzeczywistych }
  KatalogNaDysku : file of Ksiazka; { plik rekordów }
  KatalogTekstowy : text; { plik tekstowy }
```

Dla potrzeb naszego programu bibliotecznego możemy wykorzystać drugi lub trzeci z powyższych przykładów. Najpopularniejszym rozwiązaniem dla baz danych (a nasz program jest właśnie prostą bazą danych) są — jak już powiedziano — pliki rekordów, umożliwiające swobodny dostęp do danych i lepsze zagospodarowanie dysku. Jeśli jednak zależy Ci na czytelności pliku z danymi, możesz wykorzystać reprezentację w postaci pliku tekstowego.

Po zadeklarowaniu odpowiedniej zmiennej plikowej można przystąpić do właściwych operacji związanych z zapisywaniem i odczytywaniem danych. Przede wszystkim musimy skojarzyć zmienną plikową z fizycznym plikiem znajdującym się na dysku. Służy do tego procedura `assign`:

```
assign(zmienna-plikowa, nazwa-pliku)
```

`Nazwa-pliku` określa tu plik, do którego chcemy się odwoływać (łącznie z ewentualną ścieżką dostępu, czyli nazwą dysku, katalogu i ewentualnych podkatalogów zawierających plik). Po wykonaniu procedury `assign` wszelkie odwołania do zmiennej plikowej będą dotyczyły skojarzonego z nią pliku (o nazwie którego możemy zapomnieć). Jest to dość istotne, gdyż jednym z błędów często popełnianych przez początkujących programistów jest próba odwoływania się do pliku przez podanie jego nazwy, co jest rzecz jasna nielegalne.

Przykładowe skojarzenie zmiennej plikowej z plikiem może mieć postać

```
assign(KatalogNaDysku, 'c:\biblio\dane\katalog.dat')
```

lub (lepiej)

```
assign(KatalogNaDysku, NazwaPliku)
```

W drugim przypadku nazwa pliku przekazywana jest jako zmienna, co umożliwia np. wprowadzenie jej z zewnątrz („zaszycie” nazwy wewnątrz programu zmniejsza jego uniwersalność).

Następnym krokiem jest otwarcie pliku, czyli przygotowanie go do odczytu lub zapisu. Konieczność otwarcia (i późniejszego zamknięcia) pliku wynika z metod obsługi plików przyjętych w systemie operacyjnym, którego funkcje są zresztą w tym celu wykorzystywane. Wymiana informacji pomiędzy plikiem a programem możliwa jest dopiero po otwarciu tego ostatniego.

Dwiema podstawowymi procedurami używanymi w Pascalu do otwierania plików są `reset` i `rewrite`:

```
reset(zmienna-plikowa)
rewrite(zmienna-plikowa)
```

Procedura `reset` umożliwia otwarcie już istniejącego pliku, ustawiając tzw. *wskaźnik plikowy* na jego początku. W przypadku, gdy otwierany plik nie istnieje, wywołanie procedury reset kończy się błędem wykonania. Z kolei `rewrite` umożliwia otwarcie pliku niezależnie od tego, czy istniał on poprzednio: jeśli nie — tworzy ona nowy plik o danej nazwie, zaś jeśli tak — zeruje długość istniejącego pliku i ustawia wskaźnik plikowy na jego początku (czego efektem jest **utracenie wszystkich danych zawartych w pliku**). Warto pamiętać, że w przypadku plików tekstowych procedura `reset` otwiera plik wyłącznie do odczytu, zaś `rewrite` — wyłącznie do zapisu (nie ma zatem możliwości mieszania odczytów i zapisów w jednym cyklu otwarcia). Zasada ta nie obowiązuje dla plików elementowych, które można odczytywać i zapisywać bez ograniczeń niezależnie od tego, czy zostały otwarte za pomocą procedury `reset`, czy `rewrite` (w tym ostatnim przypadku trzeba najpierw zapisać do pliku jakieś dane).

Sam wskaźnik plikowy jest po prostu kolejnym numerem elementu (nie bajtu!) w pliku, przy czym numeracja rozpoczyna się od zera. Każda operacja odczytu lub zapisu powoduje przesunięcie wskaźnika o wartość równą liczbie odczytanych lub zapisanych elementów, przy czym dla plików o dostępie swobodnym (elementowych) możliwe jest również jego dowolne przestawianie (nawet poza koniec pliku, choć ma to mały sens i najczęściej powoduje błędy wykonania).

Trzecią procedurą otwierającą, dostępną wyłącznie dla plików tekstowych, jest `Append`. Procedura ta otwiera plik do dopisywania, tj. otwiera go do zapisu nie niszcząc poprzedniej zawartości i ustawia wskaźnik plikowy na jego końcu. Umożliwia to dodawanie danych do plików tekstowych, które — jako pliki o dostępie sekwencyjnym — nie umożliwiają programowego przestawiania wskaźnika plikowego.

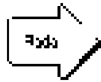
Do wymiany danych pomiędzy programem a plikiem służą znane nam już procedury `read` (odczyt) i `write` (zapis). Ponieważ w „standardowej" wersji obsługują one ekran monitora i klawiaturę, niezbędne jest podanie dodatkowego argumentu określającego plik, z/do którego informacja ma być odczytana lub zapisana. Argumentem tym jest właśnie nazwa odpowiedniej zmiennej plikowej:

```
read(zmienna-plikowa, lista-elementów)
write(zmienna-plikowa, lista-elementów)
```

Powyższe operacje odnoszą się zarówno do plików elementowych, jak i tekstowych. Dla tych ostatnich możliwe jest ponadto użycie procedur `readln` i `writeln`, odczytujących lub zapisujących dane wraz ze znakami końca wiersza. Ponieważ pliki elementowe przechowują wyłącznie dane określonego typu i nie mogą zawierać znaków końca wiersza, użycie procedur `readln` i `writeln` jest w ich przypadku nielegalne. Drugą istotną różnicą jest zawartość `listy-elementów`. W przypadku plików tekstowych lista ta może zawierać dowolne zmienne, stałe i wyrażenia (gdyż wszystkie zapisywane są w pliku w postaci tekstu), natomiast dla plików elementowych jej składnikami mogą

być wyłącznie zmienne odpowiedniego typu. Dzieje się tak dlatego, że plik elementowy może zawierać wyłącznie dane jednego typu, zaś poszczególne elementy listy przekazywane są przez nazwę.

Po wykonaniu żądanych operacji zapisu i odczytu danych plik należy *zamknąć*. Ta bardzo ważna operacja jest ignorowana przez wielu programistów, co w efekcie prowadzi do przykrych niespodzianek w postaci zgubionych danych. U podłoża całego problemu leży tak zwane *buforowanie operacji dyskowych*, czyli technika polegająca na odczytywaniu i zapisywaniu danych nie pojedynczo, lecz całymi paczkami, za pośrednictwem specjalnego obszaru pamięci — tzw. *bufora dyskowego*. Wykorzystanie bufora pozwala na zredukowanie liczby fizycznych odczytów i zapisów na dysku, a przez to zmniejszenie jego mechanicznego obciążenia i poprawę wydajności operacji dyskowych. Ponieważ jednak podczas zapisu zawartość bufora wysyłana jest na dysk dopiero po jego zapełnieniu (lub w chwili zamknięcia pliku), przerwanie wykonywania programu może spowodować utratę danych. Również poprawne zakończenie programu powoduje co prawda automatyczne zamknięcie otwartych plików, nie opróżnia jednak buforów przechowujących nie zapisane jeszcze dane, co może spowodować ich utratę. W przypadku, gdy plik wykorzystywany jest wyłącznie do odczytu danych, niezamknięcie nie powoduje utraty informacji, co nie znaczy, że można je sobie odpuścić, bowiem lenistwo takie zwykle mści się w najmniej stosownych okolicznościach.



Pamiętaj: zamknięcie pliku jest praktycznie ***jedynym sposobem*** *na bezpieczne zapisanie w nim wszystkich danych.*

Na szczęście zamknięcie pliku jest bardzo proste. Realizuje je procedura `close`:

```
close(zmienna-plikowa)
```

Jej wywołanie ma tę samą formę dla wszystkich rodzajów plików.

Tyle teorii. Aby wprowadzić ją w życie, spróbujmy napisać zestaw procedur pozwalających na zapisanie naszego katalogu w pliku dyskowym i odczytanie go z pliku. Zgodnie z tym, co powiedzieliśmy wcześniej, do przechowywania zawartości katalogu wykorzystamy plik elementowy typu **`file of`** `record`.

```
procedure ZapiszNaDysku(NazwaPliku : string);

var
  f : file of Ksiazka;
  i : integer;

begin
  assign(f, NazwaPliku); { skojarz plik ze zmienną plikową }
  rewrite(f);            { otwórz (utwórz) plik }
  for i := 1 to LbPoz do { zapisz kolejne rekordy }
    write(f, Katalog[i]);
  close(f);              { zamknij plik }
end;
```

Powyższa procedura ilustruje typową metodę zapisywania w pliku zawartości bazy danych. Po skojarzeniu pliku z odpowiednią zmienną i otwarciu go zapisujemy kolejne rekordy znajdujące się w tablicy (zwróć uwagę, że rekordy zapisywane są w całości, a nie polami). Po zapisaniu właściwej liczby rekordów zamykamy plik... i to wszystko. Musisz jeszcze zdawać sobie sprawę, że każde otwarcie będzie powodowało utratę poprzedniej zawartości pliku (`rewrite`!), ale ponieważ przed chwilą niejawnie założyliśmy, że każdorazowo zapisujemy cały katalog, jest to do przyjęcia.

Odczytanie zawartości katalogu z pliku przebiega według nieco innego schematu. Ponieważ nie wiemy, ile rekordów mamy właściwie odczytać, nie możemy zastosować pętli **`for`**. Najpopularniejszym rozwiązaniem jest czytanie kolejnych rekordów do momentu napotkania końca pliku, co wykrywane jest przez procedurę `eof` (ang. ***end-of-file*** — koniec pliku). Jak nietrudno się domyślić, tym razem zastosujemy pętlę **`while`** (lub **`repeat`**):

```
procedure OdczytajZDysku(NazwaPliku : string);

var
  f : file of Ksiazka;
  i : integer;

begin
  assign(f, NazwaPliku); { skojarz plik ze zmienną plikową }
  reset(f);              { otwórz plik (musi istnieć!) }
  i := 0;                { wyzeruj licznik rekordów }
  while not eof(f) do    { czytaj aż do końca pliku }
    begin
      Inc(i);            { kolejny rekord }
      read(f, Katalog[i]);
    end;
  LbPoz := i;            { wczytano tyle rekordów }
     close(f);
end;
```

Pozostałe operacje wykonywane w procedurze są praktycznie takie same, jedynie do otwarcia pliku wykorzystujemy tym razem procedurę `reset`. Zauważ, że konstrukcja procedur `ZapiszNaDysku` i `OdczytajZDysku` przewiduje przekazanie nazwy pliku jako parametru, co z kolei pozwala na ich użycie bez konieczności każdorazowego komunikowania się z użytkownikiem (nazwę można „zaszyć" w programie jako stałą). Jednym z możliwych (i często stosowanych) rozwiązań kwestii zapamiętywania danych jest każdorazowe zapisywanie całej bazy w chwili zakończenia programu i odczytywanie jej zaraz po uruchomieniu. W tym celu wystarczy na początku części operacyjnej (przed wywołaniem menu) wstawić instrukcję

```
OdczytajZDysku(PLIK_KATALOGU);
```

zaś przed samym końcem programu dopisać

```
ZapiszNaDysku(PLIK_KATALOGU);
```

przy czym stałą `PLIK_KATALOGU` należy wcześniej zdefiniować jako np.

```
PLIK_KATALOGU = 'Katalog.dat'
```

Metoda ta pozwala na każdorazowe zapamiętywanie treści katalogu po zakończeniu programu i jej odtwarzanie na początku kolejnej sesji bez konieczności podawania nazwy pliku (uwaga: przed pierwszym uruchomieniem programu musisz utworzyć pusty plik o nazwie KATALOG.DAT, w przeciwnym przypadku próba otwarcia nie powiedzie się). Innym sposobem jest uzupełnienie menu o polecenia zapisu i odczytu danych, co jednak wiąże się z koniecznością wywoływania odpowiednich poleceń. Jeśli wreszcie chcesz dać użytkownikowi możliwość zmiany nazwy pliku z danymi, musisz uzupełnić program o odpowiednie instrukcje wczytujące ją z klawiatury.

Do przechowywania danych można również wykorzystać plik tekstowy, jednak operacje odczytu i zapisu będą nieco bardziej skomplikowane. Wymiana danych z plikiem tekstowym odbywa się tak samo, jak z monitorem i klawiaturą, a więc poszczególne pola rekordów należy zapisywać i odczytywać indywidualnie. W zamian za to uzyskujemy możliwość łatwego obejrzenia treści pliku (gdyż zawiera on wyłącznie tekst), a także skierowania (lub odczytania) danych do (z) jednego ze standardowych urządzeń wyjścia (wejścia) obsługiwanych przez system operacyjny (np. drukarki). Oto przykład procedury zapisującej katalog do pliku tekstowego (operację odczytu możesz zrealizować w ramach ćwiczeń):

```
procedure ZapiszNaDysku(NazwaPliku : string);

var
  f : text; { tym razem plik tekstowy }
  i : integer;

begin
  assign(f, NazwaPliku); { skojarz plik ze zmienną plikową }
  rewrite(f);            { utwórz plik }
  for i := 1 to LbPoz do { zapisz kolejne rekordy }
    with Katalog[i] do  { wyprowadź poszczególne pola rekordu }
      begin
        writeln(f, 'Pozycja katalogu: ', i);
        writeln(f, 'Tytul: ', Tytul);
        writeln(f, 'Autor: ', Autor);
        if Wypozyczajacy = '' then { nikt nie wypożyczył }
          writeln(f, 'Ksiazka znajduje sie na polce.')
        else { pole zawiera nazwisko wypożyczającego }
          writeln(f, 'Wypozyczajacy: ', Wypozyczajacy);
        writeln(f);
      end;
  close(f); { zamknij plik }
end;
```

Sposób zapisania treści rekordu do pliku przypomina procedurę `WypiszDane` (zobacz poprzedni rozdział), z tym, że każda instrukcja `write(ln)` uzupełniona jest o specyfikację zmiennej plikowej, np.

```
writeln(f, 'Autor: ', Autor);
```

Odczytywanie danych z plików tekstowych nastręcza nieco więcej kłopotów, zwłaszcza gdy plik zawiera informacje mieszanych typów (tekst i liczby). Ceną za „luźny" format zapisu jest najczęściej konieczność tworzenia skomplikowanych procedur realizujących konwersje typów i sprawdzanie poprawności danych.

Na szczęście pliki tekstowe posiadają również zalety (jedną z nich jest właśnie czytelny format). Ponieważ standardowe urządzenia wyjściowe są w systemie DOS obsługiwane tak samo, jak pliki tekstowe, aby wyprowadzić treść katalogu na ekran (tzw. konsolę operatorską), wystarczy następujące wywołanie:

```
ZapiszNaDysku('con');
```

zaś aby wydrukować katalog na drukarce, musisz użyć wywołania

```
ZapiszNaDysku('prn');
```

gdzie `con` i `prn` są nazwami systemowych urządzeń wyjścia (konsoli i drukarki).

Na koniec wrócimy na chwilę do plików elementowych, by powiedzieć kilka słów o metodach realizowania swobodnego dostępu do danych. Bieżący element pliku (tj. element, którego dotyczyła będzie kolejna instrukcja `read` lub `write`) wskazywany jest przez wspomniany już wskaźnik plikowy. Wywołanie

```
Seek(zmienna-plikowa, numer-elementu)
```

powoduje ustawienie wskaźnika plikowego tak, by kolejna operacja odczytu lub zapisu rozpoczęła się od elementu danego `numerem-elementu` (elementy numerowane są od zera). Dodatkowymi funkcjami wykorzystywanymi podczas manipulowania wskaźnikiem plikowym są `FilePos`, zwracająca jego bieżącą wartość, oraz `FileSize`, zwracająca rozmiar pliku. Tak więc — zakładając, że `f` jest plikiem typu **`file of`** `integer` — wywołanie

```
Seek(f, FileSize(f));
```

ustawi wskaźnik plikowy za ostatnim elementem pliku (czyli przygotuje plik do dopisywania), zaś konstrukcja

```
for i := 1 to 10 do
  begin
    Seek(f, random(FileSize(f));
    read(f, i);
    writeln(i);
  end;
```

odczyta z pliku dziesięć losowo wybranych liczb (funkcja `random(k)` zwraca liczbę pseudoprzypadkową z zakresu od zera do `k`) i wyświetli je na ekranie.

Swobodny dostęp do danych, chociaż bardzo przydatny, nie powinien być nadużywany. Wszelkie operacje na plikach są znacznie bardziej czasochłonne, niż operacje na danych umieszczonych w pamięci, zaś operacje w trybie sekwencyjnym są znacznie szybsze, niż w trybie dostępu swobodnego. Niezależnie od tych uwag musisz pamiętać, że wszelkie manipulacje na zawartości plików wiążą się z ryzykiem utraty danych w przypadku awarii systemu (np. wyłączenia zasilania). Dlatego też pliki należy zamykać bezpośrednio po wykonaniu niezbędnych czynności.

Zapamiętaj

- Do trwałego przechowywania informacji w systemie komputerowym służą pliki.
- Turbo Pascal umożliwia obsługę plików elementowych, tekstowych oraz amorficznych.
- Pliki elementowe składają się z elementów tego samego typu (prostego lub strukturalnego). Liczba elementów jest zawsze całkowita.
- Pliki tekstowe zawierają dane reprezentowane w postaci wierszy tekstu. Można ich używać do przechowywania danych różnych typów.
- Pliki elementowe umożliwiają dostęp swobodny, natomiast pliki tekstowe pozwalają wyłącznie na dostęp sekwencyjny.
- Plik reprezentowany jest w programie przez zmienną plikową, którą należy skojarzyć z fizycznym plikiem za pomocą procedury `assign`.
- Operacje na zawartości pliku muszą być poprzedzone jego otwarciem (`reset` lub `rewrite`) i **zakończone zamknięciem** (`close`).
- Swobodny dostęp do elementów pliku umożliwia procedura `Seek`, ustawiająca wskaźnik plikowy na zadanej pozycji.
- Mechanizmy obsługi plików tekstowych umożliwiają przesyłanie danych nie tylko na dysk, lecz również z i do standardowych urządzeń wejścia-wyjścia systemu operacyjnego.

Łańcuchy

Dane tekstowe mają — obok grafiki — największy udział w objętości informacji przetwarzanej i przechowywanej we współczesnych systemach komputerowych. Z tego też względu każdy szanujący się język wysokiego poziomu jest wyposażony w mechanizmy pozwalające na reprezentowanie, przechowywanie i przetwarzanie tekstów. W Turbo Pascalu służy do tego typ łańcuchowy (**string**), którym zajmiemy się w tym rozdziale. Dowolny tekst (łańcuch, ang. *string*) przechowywany jest w programie w postaci ciągu znaków, który może być interpretowany jako specyficzna tablica

```
array[0..255] of char
```

Przykładowa deklaracja zmiennej łańcuchowej ma postać

```
var
  Napis : string;
```

Stałe łańcuchowe zapisuje się natomiast w postaci ciągów znaków ujętych w apostrofy (podobnie jak stałe znakowe):

```
const
  STALY_NAPIS = 'Turbo Pascal';
```

Zerowy element łańcucha przechowuje jego aktualną długość (tzw. *długość dynamiczną*); będąc typu znakowego może on przyjmować wartości od 0 do 255. Stąd właśnie wynika ograniczenie długości łańcucha do 255 znaków, co zresztą w większości przypadków wystarcza aż nadto. Aby „skrócić" łańcuch (dana typu **string** zajmuje zawsze 256 bajtów, niezależnie od rzeczywistej długości tekstu), można wykorzystać deklarację

```
nazwa-zmiennej : string[długość]
```

Możliwość ta jest szczególnie cenna, jeśli w programie wykorzystujesz np. tablicę łańcuchów: deklarując element składowy tablicy jako `string[20]` oszczędzasz 235 bajtów, co przy stu elementach daje zysk ponad 20 kB. Warto zauważyć, że próba zapisania do „skróconego" łańcucha tekstu dłuższego niż pozwala deklaracja nie spowoduje błędu, a jedynie obcięcie nadmiarowych znaków.

Operacje na łańcuchach w zasadzie nie różnią się zapisem od operacji na zmiennych typu prostego i nie wymagają stosowania żadnych specjalnych sztuczek. Do wprowadzania, wyprowadzania i przypisywania łańcuchów wykorzystuje się — podobnie jak dla zmiennych typów prostych — procedury `read(ln)`, `write(ln)` oraz operator przypisania. Również porównanie dwóch łańcuchów zapisywane jest identycznie, przy czym „wewnętrznie" odbywa się ono przez porównanie kodów odpowiadających sobie znaków. Tak więc:

`'C' < 'Pascal'`	(kod ASCII znaku 'C' jest mniejszy od kodu 'P')
`'c' > 'Pascal'`	(kod ASCII znaku 'c' jest większy od kodu 'P')
`'C' > ''`	(dowolny łańcuch jest większy od łańcucha pustego)

Z rzeczy prostych pozostało jeszcze dodawanie łańcuchów, polegające na ich zwykłym „sklejaniu" (niestety, łańcuchów nie da się odejmować, mnożyć ani dzielić). Jeżeli zmienna `lancuch1` zawiera tekst `'Turbo'`, zaś `lancuch2` — tekst `'Pascal'`, to wynikiem sklejenia obu zmiennych:

```
wynik := lancuch1 + lancuch2;
```

będzie oczywiście tekst `'TurboPascal'`.

Również odwołania do poszczególnych znaków łańcucha realizuje się w sposób elementarny. Ponieważ może on być traktowany jako tablica znaków, instrukcja

```
s[5] := 'x'
```

wstawi znak `x` na piątą pozycję w łańcuchu `s`.

Bardziej wymyślne operacje na łańcuchach wymagają użycia specjalnie do tego celu przeznaczonych funkcji, z których najważniejsze opisano poniżej:

`Length(s)`	— zwraca bieżącą długość łańcucha `s`;
`Concat(s1, s2)`	— skleja łańcuchy `s1` i `s2` (podobnie, jak operator `+`)
`Copy(s, m, n)`	— zwraca podłańcuch o długości `m` znaków wycięty z łańcucha `s` poczynając od pozycji `n`;
`Pos(ch, s)`	— zwraca numer pozycji, na której w łańcuchu `s` znajduje się znak `ch`;
`Delete(s, m, n)`	— usuwa `n` znaków z łańcucha `s` poczynając od pozycji `m`.

Jak powiedziano wyżej, aktualną długość łańcucha można odczytać funkcją `Length` (lub przez bezpośrednie odwołanie do zerowej komórki łańcucha). Aby zmienić długość dynamiczną łańcucha, musisz użyć konstrukcji

```
s[0] := chr[n]
```

gdzie `n` jest żądaną długością (ponieważ łańcuch składa się ze znaków, musimy przekształcić liczbę `n` na odpowiadający jej znak funkcją `chr`). Ponieważ operacja ta czasem przynosi niezbyt pożądane efekty, lepiej jej unikać.

Przytoczony poniżej program `Lancuchy` demonstruje niektóre możliwości obróbki łańcuchów i w zasadzie nie wymaga dodatkowego komentarza. Poza wywołaniami opisanych wyżej procedur znalazła się w nim również funkcja `UpCase`, przekształcająca małą literę alfabetu na dużą. Operuje ona co prawda na typie znakowym, jednak typowe jej zastosowanie sprowadza się do konwersji całych łańcuchów, jak pokazano niżej.

```
program Lancuchy;
{ Demonstracja operacji na łańcuchach }

const
  TP = 'Turbo Pascal to bomba';

var
  s1, s2 : string;
  i : integer;

begin
  s1 := TP; { przypisanie }
  s2 := ''; { j.w., łańcuch pusty }

  for i := 1 to Length(s1) do
    s2 := s2 + ' '; { dodawanie łańcuchów/znaków }
  writeln(s1);

  i := Pos('a', s1); { wyszukanie znaku }
  s2[i] := '^'; { wstawienie znaku }
  writeln(s2); { i co z tego wynikło? }

  Delete(s1, 1, 6); { usunięcie części łańcucha }
  writeln(s1);

  for i := Length(s1) downto 1 do { wypisanie łańcucha }
                                  { od tyłu }
    write(s1[i]);
  writeln;

  for i := 1 to Length(s1) do { zamiana na duże znaki }
    write(UpCase(s1[i]));
  writeln;

  s1 := Copy(TP, 1, 13); { wycięcie podłańcucha }
  for i := Length(s1) downto 1 do
    begin
      writeln(s1);
      Dec(s1[0]); { skracanie łańcucha }
    end;
end.
```

Na zakończenie tego rozdziału wspomnimy o alternatywnej metodzie reprezentowania danych tekstowych — tak zwanych *łańcuchach zakończonych zerem* (ang. *null-terminated string*), zwanych też ASCIIZ. Łańcuch ASCIIZ (podobnie jak **`string`**) jest zwykłą tablicą znaków, jednak nie posiada pola przechowującego długość; w zamian za

to jego koniec sygnalizowany jest znakiem o kodzie 0 (nie mylić ze znakiem „0”). Efektywna pojemność łańcucha ASCIIZ ograniczona jest wielkością dostępnej pamięci (w praktyce do 64 kB) a więc jest znacznie większa, niż dla typu **`string`**.

Łańcuchy ASCIIZ (dostępne począwszy od wersji 7.0 Turbo Pascala) deklarowane są jako zwykłe tablice znaków indeksowane od zera, np.:

```
var
  BardzoDlugiLancuch : array[0..10000] of char;
```

Elementarna obsługa łańcuchów ASCIIZ (wczytywanie, wyprowadzanie, przypisywanie) realizowana jest tak samo, jak dla zwykłych łańcuchów, pod warunkiem włączenia tzw. rozszerzonej składni dyrektywą kompilatora `{$X+}` (***Options-Compiler-Extended Syntax***). Bardziej złożone operacje na łańcuchach ASCIIZ (kopiowanie, porównywanie, przeszukiwanie, konwersja do typu **`string`** i vice versa) realizowane są przez procedury zawarte w module bibliotecznym `Strings` (o modułach wkrótce) i nie będą tu omawiane. Warto wreszcie wspomnieć o typie wskaźnikowym `PChar`, umożliwiającym manipulowanie na dynamicznie tworzonych i usuwanych łańcuchach ASCIIZ.

Ponieważ w większości przypadków typ **`string`** znakomicie spełnia swoje zadanie, poprzestaniemy na powyższych wzmiankach, odsyłając zainteresowanych Czytelników do systemu pomocy i literatury [2, 3]. W następnym rozdziale zajmiemy się problemem braku pamięci, czyli wspomnianym przed chwilą dynamicznym tworzeniem i usuwaniem zmiennych.

Zapamiętaj

- Do przechowywania danych tekstowych (napisów) służy w Turbo Pascalu typ **`string`**.
- Łańcuch typu **`string`** jest specyficzną tablicą o pojemności do 255 znaków, przy czym bieżąca długość łańcucha przechowywana jest w zerowej komórce tablicy.
- Zakres podstawowych operacji na łańcuchach obejmuje wprowadzanie, wyprowadzanie, przypisywanie i porównywanie (realizowane za pomocą „standardowych” procedur i operatorów).
- Bardziej zaawansowane operacje na łańcuchach to kopiowanie, przeszukiwanie, wycinanie i skracanie. Operacje te realizowane są za pomocą specjalnych funkcji i procedur.
- Turbo Pascal 7.0 umożliwia również korzystanie z łańcuchów zakończonych zerem (ASCIIZ).

Więcej pamięci!

Jak być może jeszcze pamiętasz, przystępując do pisania naszego programu bibliotecznego określiliśmy maksymalną liczbę pozycji (rekordów) na około 750. Ograniczenie to wynikało z wielkości dostępnej pamięci, którą z kolei oszacowaliśmy na z grubsza 64 kB. Dlaczego jednak tylko tyle? W chwili obecnej trudno jest znaleźć komputer PC posiadający mniej niż 1 MB pamięci, zaś standardem jest 8 lub 16 MB. Co prawda system operacyjny DOS, pod nadzorem którego pracują programy pascalowe, umożliwia dostęp tylko do 640 kB, ale i tak program

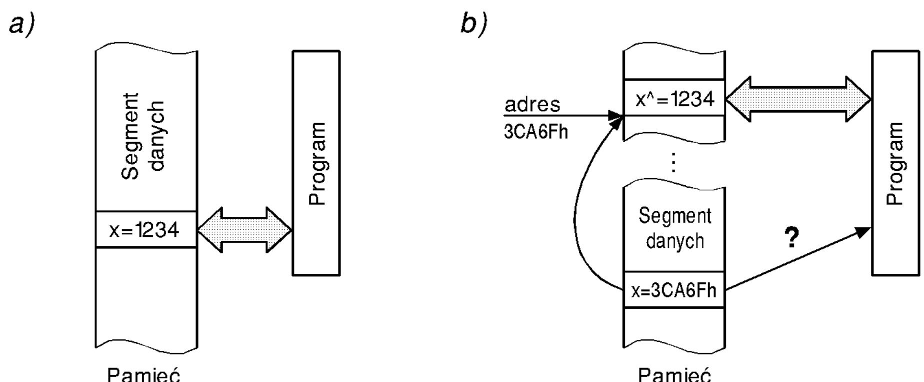
```
program IlePamieci;

begin
  writeln('Masz w tej chwili ', MemAvail, ' bajtow wolnej
pamieci.');
end.
```

pokaże zapewne liczbę większą od 64 kB. Jak zatem dobrać się do pozostałej pamięci?

Zaczniemy od małej dawki teorii. Wszystkie zmienne, którymi posługiwałeś się do tej pory, należały do klasy zmiennych globalnych (jeśli zdefiniowałeś je na początku programu) lub lokalnych (jeżeli zostały zdefiniowane w procedurach). Zmienne globalne umieszczane są w tzw. *segmencie danych*, którego wielkość wynosi 64 kB i jest stała; zmienne lokalne przechowywane są w *segmencie stosu*, którego wielkość można zmieniać w granicach od 1 do 64 kB. Zmienne globalne są *statyczne*, tj. istnieją przez cały czas wykonywania programu, natomiast zmienne lokalne „żyją” tylko tak długo, jak długo wykonują się posiadające je procedury lub funkcje. Koniec końców, na zmienne możemy przeznaczyć co najwyżej 128 kilobajtów, z czego 64 kB dostępne jest „warunkowo”. Skoro jednak program `IlePamieci` pokazał (mam nadzieję...), że pamięci masz nieco więcej, musimy znaleźć sposób na jej wykorzystanie.

Sposobem tym są tak zwane *zmienne wskazywane*. W odróżnieniu od „zwykłych” zmiennych, zmienne wskazywane mogą być umieszczane w dowolnym miejscu pamięci, z czym wiąże się nieco inny sposób odwoływania do nich, wykorzystujący tzw. *wskaźniki*.

Wskaźnik do zmiennej jest po prostu jej adresem, czyli liczbą opisującą jej położenie w pamięci. Różnicę pomiędzy zmienną statyczną a wskazywaną ilustruje poniższy rysunek:



Rysunek 12. *Sposób obsługi zmiennych statycznych (a) i wskazywanych (b)*

Jak widać, dostęp do zmiennej wskazywanej odbywa się dwuetapowo, za pośrednictwem wskaźnika: program odczytuje wartość tego ostatniego (czyli adres zmiennej) i na jej podstawie odwołuje się do właściwej zmiennej, położonej w pamięci w miejscu opisanym adresem (w naszym przypadku 3CA6F szesnastkowo). Metoda ta powoduje nieznaczne wydłużenie czasu dostępu do zmiennej, ale pozwala za to umieścić ją praktycznie w dowolnym miejscu pamięci (dokładnie, zmienne wskazywane umieszczane są na tzw. *stercie* (ang. *heap*) — wydzielonym obszarze pamięci zarządzanym przez specjalne procedury).

Deklaracja wskaźnika do zmiennej określonego typu jest bardzo podobna do „zwykłej" deklaracji zmiennej i różni się od niej jedynie symbolem wskaźnika (^):

```
zmienna : ^typ
```

przy czym `typ` musi być identyfikatorem typu prostego lub zdefiniowanego wcześniej w sekcji **`type`**. Odwołanie do samego wskaźnika wygląda dokładnie tak samo, jak dla każdej innej zmiennej, natomiast odwołanie do wskazywanej przez niego zmiennej zawiera dodatkowo znaczek ^, tym razem położony za nazwą wskaźnika:

```
zmienna^
```

A oto kilka przykładów:

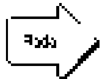
```
type
  TabReal = array[1..200] of real; { tablica 100 liczb real }
  PTabReal = ^TabReal; { wskaźnik do tablicy }
  string100 = string[100]; { skrócony łańcuch }
var
  TabTabReal : array[1..200] of PTabReal; { tablica wskaźników }
  Pstring100 : ^string100; { wskaźnik do łańcucha }
  pi : ^integer; { wskaźnik do liczby integer }
```

```
{ ... }

TabTabReal[10]^[30] := 12.345;
Pstring100^ := 'Lancuch dynamiczny';
writeln(PString100^);
```

Na szczególną uwagę zasługuje tu zmienna `TabTabReal`, będąca tablicą wskaźników do tablic liczb rzeczywistych. Struktura tego typu pozwala na efektywne przechowanie dwuwymiarowej tablicy składającej się z 200 wierszy (wskazywanych wskaźnikami) zawierających po 200 liczb każdy, czyli w sumie 40 tysięcy liczb o objętości około 240 kB! Umieszczoną nieco niżej instrukcję `TabTabReal[10]^[30] := 12.345` można wytłumaczyć jako odwołanie do trzydziestej komórki w dziesiątym wierszu, czyli tablicy wskazywanej przez dziesiąty wskaźnik w tablicy `TabTabReal`. Jak nietrudno się już domyślić, głównym zastosowaniem wskaźników jest obsługa dużych struktur danych (np. tablic rekordów, czyli katalogów bibliotecznych...). Z tego też względu mały sens ma deklaracja zmiennej `pi`, gdyż liczba typu `integer` zajmuje dwa bajty, zaś wskaźnik — cztery.

Warto tu jeszcze wspomnieć, że oprócz wskaźników do obiektów konkretnego typu Turbo Pascal udostępnia również wskaźniki amorficzne, tj. wskazujące na obszar pamięci i nie związane z żadnym konkretnym typem. Wskaźnik taki deklarowany jest słowem `pointer` (*wskaźnik*) i jest zgodny pod względem przypisania z innymi typami wskaźnikowymi. Typ `pointer` i związane z nim operacje wykorzystywane są przez bardziej zaawansowanych programistów do bezpośredniego, „niskopoziomowego" manipulowania zawartością pamięci.

Drugą bardzo ważną i pożyteczną cechą zmiennych wskazywanych jest możliwość ich tworzenia i niszczenia w zależności od potrzeb. Wymaga to co prawda użycia specjalnych procedur, pozwala jednak na znacznie bardziej efektywną gospodarkę pamięcią. W odróżnieniu od statycznych zmiennych globalnych, istniejących przez cały czas wykonywania programu, zmienne wskazywane należą do klasy zmiennych *dynamicznych* (czyli istnieją dokładnie wtedy, gdy życzy sobie tego programista). Sam proces utworzenia zmiennej dynamicznej polega na zarezerwowaniu odpowiedniego obszaru pamięci i zapamiętaniu adresu tego obszaru we wskaźniku wskazującym na zmienną. Z kolei usunięcie zmiennej powoduje „zwolnienie rezerwacji" (zawartości zmiennej oraz wskaźnika nie są fizycznie niszczone, ale lepiej się już do nich nie odwoływać). Ceną za możliwość swobodnego przydzielania i zwalniania pamięci jest konieczność bezwzględnego inicjalizowania wskaźników, które przed utworzeniem wskazywanej zmiennej mają wartość nieokreśloną (zwykle zero, co odpowiada wskaźnikowi **`nil`**, nie wskazującemu na żaden obiekt). Próba odczytu zmiennej wskazywanej przez taki wskaźnik dostarczy „tylko" bezsensownego wyniku, natomiast próba zapisu (w przypadkowe miejsce pamięci!!) może skończyć się zawieszeniem komputera lub zupełnie nieprzewidzianym jego zachowaniem.



Pamiętaj o inicjalizacji wskaźników!

Turbo Pascal oferuje kilka metod tworzenia i usuwania zmiennych dynamicznych, z których najpopularniejszą realizuje para procedur `new` i `dispose`:

```
new(wskaźnik-do-zmiennej)
dispose(wskaźnik-do-zmiennej)
```

Procedura `new` wykonuje czynności związane z utworzeniem zmiennej wskazywanej, natomiast `dispose` — operacje związane z jej usunięciem. Drugą parę zarządzającą dynamicznym przydziałem pamięci tworzą procedury `GetMem` i `FreeMem`:

```
GetMem(wskaźnik, rozmiar-bloku)
FreeMem(wskaźnik, rozmiar-bloku)
```

W odróżnieniu od pary `new-dispose`, procedury te wykorzystują wskaźniki amorficzne (typu `pointer`) i służą do bardziej „wewnętrznego" manipulowania pamięcią, tj. przydzielania i zwalniania bloków bajtów (a nie zmiennych wskazywanych jakiegoś konkretnego typu). Wielkość przydzielanego lub zwalnianego bloku (w bajtach) określa parametr `rozmiar-bloku`. Korzystając z obu grup procedur musisz pamiętać, że pamięć przydzielona przez `GetMem` nie może być zwolniona procedurą `dispose` i odwrotnie.

Ostatnią, chyba najrzadziej stosowaną parę tworzą procedury `mark` i `release`:

```
mark(wskaźnik)
release(wskaźnik)
```

Wykonanie procedury `mark` nie powoduje przydzielenia pamięci ani utworzenia zmiennej, a jedynie zapamiętanie bieżącej „wysokości" sterty w zmiennej `wskaźnik`. Zwolnienia całego obszaru sterty leżącego powyżej `wskaźnika` dokonuje się za pomocą procedury `release`. Obydwie procedury stosowane są — podobnie jak `GetMem` i `FreeMem` — głównie w programowaniu niskiego poziomu, do „masowego" zwalniania pamięci przydzielonej na stercie.

Korzystając ze zmiennych dynamicznych musisz pamiętać, że sterta nie jest automatycznie porządkowana, toteż kolejne operacje przydzielenia i zwolnienia bloków pamięci (dowolną metodą) mogą doprowadzić do tzw. *fragmentacji*, czyli rozbicia wolnego jeszcze obszaru pamięci na mniejsze, rozłączne bloki. Ponieważ rozmiar tworzonej zmiennej dynamicznej nie może być większy od rozmiaru największego wolnego bloku pamięci, może się okazać, że próba utworzenia zmiennej skończy się niepowodzeniem, mimo iż wielkość dostępnej pamięci będzie wystarczająca. Dlatego też właściwą miarą możliwości utworzenia większej struktury danych na stercie jest nie funkcja `MemAvail` (zwracająca sumaryczny rozmiar wolnej pamięci), lecz `MaxAvail` (zwracająca rozmiar największego wolnego bloku).

Pora na przykłady. Na początek przedstawimy „zbiorczą" demonstrację możliwości obsługi zmiennych dynamicznych.

```
program ZmienneDynamiczne;

type
```

```
  TabReal = array[1..5000] of real; { tablica liczb }
                                    { rzeczywistych }
  PString = ^string; { wskaźnik do łańcucha }

var
  s : PString; { zmienna typu wskaźnik do łańcucha }
  TabTabReal : array[1..100] of ^TabReal; { tablica }
                                          { wskaźników }
  Sterta : pointer; { wskaźnik wysokości sterty }
  i : integer; { pomocniczy licznik }

procedure IlePamieci;

begin
  writeln('Wolne: ', MemAvail, ' max. blok: ', MaxAvail,
          ' bajtow.');
end;

begin
  writeln(s^); { zmienna nie utworzona }
  new(s);      { więc ją tworzymy }
  writeln(s^); { utworzona, lecz nie zainicjalizowana }
  s^ := 'No wreszcie!'; { inicjalizujemy }
  writeln(s^); { teraz jest OK }
  dispose(s);  { usuwamy }
  writeln(s^); { zmienna nie została całkowicie zniszczona! }
  mark(Sterta); { zaznaczamy 'poziom' sterty }
  i := 1;      { tworzymy tablicę tablic dynamicznych }
  while MemAvail > SizeOf(TabReal) do { tyle wierszy )
                                      { ile się da }
    begin
      IlePamieci; { ile mamy pamięci? }
      new(TabTabReal[i]); { tworzymy nowy wiersz }
      Inc(i); { zwiększamy indeks wiersza }
    end;
  dispose(TabTabReal[3]); { usuwamy jeden wiersz tablicy }
  IlePamieci;
  release(Sterta); { zwalniamy hurtem całą pamięć }
  IlePamieci;
end.
```

Pierwsza część programu demonstruje etapy tworzenia, wykorzystania i usunięcia zmiennej wskazywanej (w naszym przypadku łańcucha) za pomocą procedur `new` i `dispose`. Zauważ, że utworzenie zmiennej wskazywanej nie jest równoznaczne z jej inicjalizacją, a po wykonaniu procedury `dispose` treść łańcucha nie jest niszczona, chociaż może być niekompletna.

Druga część tworzy typową strukturę wielkiej tablicy, przydzielając pamięć dla poszczególnych wierszy, dopóki to jest możliwe. Zauważ, że po usunięciu trzeciego wiersza tablicy na ogół okazuje się, że rozmiar największego dostępnego bloku jest mniejszy od całkowitego rozmiaru wolnego obszaru sterty, co uniemożliwia tworzenie

większych struktur dynamicznych. Wreszcie instrukcja `release` zwalnia całą stertę „wzwyż” począwszy od miejsca zarejestrowanego w zmiennej `Sterta`.

Na zakończenie tego rozdziału spróbujemy wykorzystać mechanizmy zmiennych dynamicznych do lepszego zagospodarowania pamięci w programie obsługi biblioteki. Podane niżej informacje będą miały z konieczności charakter wskazówek, powinny jednak umożliwić Ci skuteczne wprowadzenie zmian do programu. Przede wszystkim należy zadeklarować odpowiednie struktury danych:

```
type
{ ... }
PKsiazka = ^Ksiazka;

var
  Katalog : array[1..2000] of PKsiazka;
```

i usunąć zbędne już definicje stałych określających ograniczenia pamięciowe. Liczba 2000 w deklaracji tablicy `Katalog` została wybrana arbitralnie (2000 pozycji to około 170 kB). Kolejnym krokiem jest zmodyfikowanie wszystkich procedur korzystających z parametrów typu `Ksiazka` przez zmianę typu parametru na `PKsiazka`, np.:

```
procedure WprowadzDane(var r : PKsiazka);
```

co umożliwi im operowanie na zmiennych wskazywanych parametrem. Oczywiście w treści tak zmodyfikowanej procedury należy wszystkie odwołania do „zwykłego” parametru zastąpić odwołaniami wykorzystującymi wskaźnik, np.:

```
with r^ do {... }
```

Modyfikacja procedury sortującej jest zbędna, a nawet niepożądana. W nowym programie zamiana rekordów przyjmie formę zamiany wskaźników, co oczywiście będzie szybsze i bardziej efektywne niż kopiowanie całych rekordów.

Wszelkie wywołania procedur pozostaną niezmienione, chociaż faktycznie zamiast rekordu zawsze przekazywany będzie wskaźnik do rekordu. Natomiast wszystkie bezpośrednie odwołania do pól rekordów zapisanych w tablicy `Katalog` należy zastąpić odwołaniami wykorzystującymi wskaźniki:

```
Katalog[LbPoz]^.Licznik := 0;
```

Nie wolno również zapomnieć o uzupełnieniu procedur tworzących i usuwających rekordy o wywołania procedur `new` i `dispose`, których brak spowoduje najpewniej błyskawiczne zawieszenie komputera:

```
procedure DodajKsiazke;
{ ... }
  writeln('Nowa pozycja w katalogu: ', Licznik);
  new(Katalog[Licznik]); { utwórz nowy rekord w katalogu }
{ ... }
```

```
procedure UsunKsiazke(Numer : integer);
begin
  dispose(Katalog[Numer]);
{ ... }
```

Podobnie musisz postąpić z procedurą `OdczytajZDysku` (w procedurze zapisu usuwanie rekordów nie jest konieczne, ale przydzielona pamięć powinna zostać zwolniona przed zakończeniem działania programu).

Powyższe wskazówki nie obejmują operacji pomocniczych, jak np. sprawdzanie możliwości utworzenia kolejnego rekordu. Powinny one jednak wystarczyć Ci do skutecznego zmodyfikowania programu.

Zapamiętaj

- Do przechowywania większych ilości danych możesz w Pascalu wykorzystać zmienne wskazywane (dynamiczne).
- Zmienne wskazywane są umieszczane na tzw. stercie (teoretycznie w dowolnym miejscu pamięci). Mogą one być tworzone i niszczone dynamicznie, w zależności od potrzeb.
- Zmienna wskazywana lokalizowana jest za pomocą wskaźnika, który zawiera jej adres (miejsce w pamięci). Wskaźniki mogą wskazywać na zmienne konkretnego typu, mogą też być wskaźnikami amorficznymi (`pointer`).
- Przed wykorzystaniem zmiennej dynamicznej należy ją utworzyć (procedurą *new*), a po wykorzystaniu — usunąć (procedurą `dispose`).
- Do przydzielania i zwalniania bloków pamięci na stercie służą również procedury `GetMem`, `FreeMem`, `mark` i `release`.

Pożyteczne drobiazgi, czyli moduły biblioteczne

Ile razy zastanawiałeś się, jak wyczyścić ekran, zmienić kolor tekstu czy wprowadzić znak z klawiatury bez naciskania klawisza ENTER? Wykonanie tych czynności nie wymaga zachodu: wystarczy wywołać odpowiednią procedurę biblioteczną. Jak sama nazwa wskazuje, procedura taka nie jest na stałe „wbudowana” w język programowania, lecz pobierana z oddzielnego pliku (biblioteki) i dołączana do programu w trakcie tzw. konsolidacji (następującej po kompilacji).

Pascalowe pliki biblioteczne noszą nazwę modułów (ang. *unit*). Wersja 7.0 oferuje programiście dziesięć modułów standardowych:

`System` — wszystkie procedury standardowe;

`Crt` — obsługa monitora, klawiatury i głośnika;

`Dos` — obsługa funkcji systemowych (wywołania funkcji systemu MS-DOS);

`Graph` — obsługa grafiki;

`Strings` — obsługa łańcuchów ASCIIZ;

`WinDos` — odpowiednik modułu DOS wykorzystujący łańcuchy ASCIIZ;

`Printer` — obsługa drukarki;

`Overlay` — obsługa tzw. nakładek;

`Turbo3` — moduł „uzgadniający” z Turbo Pascalem 3.0;

`Graph3` — obsługa tzw. grafiki żółwia (używanej w Turbo Pascalu 3.0).

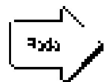
Oddzielna, obszerna grupa modułów tworzy bibliotekę Turbo Vision, którą nie będziemy się tutaj zajmować.

Jak nietrudno zauważyć, każdy moduł „specjalizuje się” w określonych operacjach (wyjątkiem jest moduł `System`, zawierający wszystkie procedury i funkcje standardowe). Najczęściej używane moduły (`System`, `Crt`, `Dos`, `Printer` i `Overlay`) umieszczono w specjalnym pliku TURBO.TPL, ładowanym do pamięci wraz z IDE Turbo Pascala, dzięki czemu ich zawartość dostępna jest natychmiast. Kod pozostałych modułów zapisany jest w odpowiednich plikach z rozszerzeniem .TPU. Niezależnie od tego, czy kod modułu znajduje się na dysku, czy jest ładowany do pamięci, aby wykorzystać dowolny jego element, za nagłówkiem programu musisz umieścić deklarację

```
uses nazwa-modułu;
```

Jeśli wykorzystujesz kilka różnych modułów, musisz rozdzielić ich nazwy przecinkami.

Chociaż moduły traktowane są najczęściej jako biblioteki procedur, zawierają one również dane w postaci stałych (np. definiujących kolory, jak `Red=2`), zmiennych (np. ustalających sposób otwierania plików — `FileMode`, czy też obsługi monitora — `DirectVideo`) i typów (np. `DateTime` — typ rekordowy przeznaczony do przechowywania czasu i daty, `PointType` — typ opisujący punkt na płaszczyźnie). Zarówno procedury, jak i dane możesz wykorzystywać bez ograniczeń, a także przysłaniać własnymi definicjami. Pamiętaj jednak, że ceną za dostęp do zawartych w module obiektów jest powiększenie objętości gotowego programu, wynikające z dołączenia „importowanego” kodu i danych. Jeśli zależy Ci na minimalizacji wielkości kodu wynikowego, zastanów się, czy koniecznie musisz używać np. modułu `Crt`. Generalnie jednak należy zalecić używanie modułów bibliotecznych, gdyż dają one programiście możliwość skorzystania ze sprawdzonych rozwiązań wielu problemów, zwalniając od konieczności wymyślania ich na własną rękę.



Stosuj moduły biblioteczne.

Spośród wymienionych wyżej modułów najczęściej stosowane są: `Crt`, umożliwiający efektywną (i efektowną) obsługę komunikacji z użytkownikiem, `Graph`, realizujący operacje w trybie graficznym, oraz `Dos`, umożliwiający wykonywanie funkcji systemowych. Ponieważ omawianie treści poszczególnych modułów jest kwestią drugorzędną (nasza książka ma przede wszystkim uczyć programowania, a nie służyć jako leksykon języka), a także ze względu na szczupłość miejsca, ograniczymy się do omówienia kilku najpopularniejszych elementów modułów `Crt` i `Dos` wraz z prostymi przykładami. Z tych samych przyczyn zrezygnujemy z omawiania bardzo obszernej zawartości modułu `Graph`, odsyłając zainteresowanych Czytelników do literatury [2].

Zaczniemy od modułu `Crt`, wspomagającego operacje wejścia i wyjścia z użyciem konsoli. Z procedur „ulepszających” obsługę ekranu i klawiatury chyba najczęściej wykorzystywana jest bezparametrowa procedura `ClrScr`, powodująca wyczyszczenie ekranu. Przykład jej użycia znajdziesz w funkcji `Menu` naszego programu bibliotecz-

nego. Jeśli dodatkowo przy okazji czyszczenia ekranu chciałbyć zmienić kolory, możesz wykorzystać procedury `TextColor` i `TextBackground`:

TextColor(kolor-tekstu)

```
TextBackground(kolor-tła)
```

zmieniające odpowiednio kolor znaków oraz tła. Oczywiście zamiast liczbowych wartości kolorów (kto by je pamiętał...) najlepiej stosować stałe symboliczne (np. `Blue=1` dla koloru niebieskiego). Efekt działania obu procedur widoczny jest dopiero po wykonaniu kolejnych operacji wyjścia (lub wyczyszczenia ekranu), zaś bieżące wartości kolorów przechowywane są w zmiennej `TextAttr`.

Procedura `GotoXY` służy do wyprowadzania tekstu na zadanych współrzędnych (ekran tekstowy liczy sobie 25 wierszy po 80 znaków; wiersze i znaki numerowane są od 1). Przykładowe wywołanie

```
for i := 1 to 24 do
  begin
    GotoXY(i,i);
    write('*');
  end;
```

wyświetli na ekranie ukośną linię złożoną z gwiazdek. Zbliżone zastosowanie ma procedura `Window`:

```
Window(x1, y1, x2, y2)
```

umożliwiająca wyznaczenie na ekranie okienka ograniczającego obszar wypisywania tekstu do prostokąta o współrzędnych `x1`, `y1` (lewy górny róg) i `x2`, `y2` (prawy dolny róg). Działanie procedury `Window` najlepiej ilustruje poniższy przykład:

```
Window(10, 10, 20, 20); { okienko 10x10 }
write('W tym okienku wypisuje sie bardzo dlugi tekst');
```

Obydwie procedury umożliwiają skuteczne tworzenie ładnych menu czy „formularzy" niewielkim nakładem kosztów, chociaż często okazuje się, że lepiej w tym celu wykorzystać np. bibliotekę Turbo Vision.

Aby wprowadzić z klawiatury pojedynczy znak bez naciskania klawisza ENTER warto wykorzystać funkcję `ReadKey`. Zwraca ona kod ASCII znaku odpowiadającego klawiszowi albo — w przypadku naciśnięcia klawisza funkcyjnego lub specjalnego — tzw. rozszerzony kod klawisza poprzedzony znakiem o kodzie 0 (w tym przypadku konieczne są dwa wywołania). Funkcja `ReadKey` jest stosowana głównie do realizacji menu (*vide* program `Katalog`) i innych wymyślniejszych operacji wykorzystujących klawiaturę. Spokrewniona z nią funkcja `KeyPressed` służy do badania stanu bufora klawiatury: jeśli przed jej wywołaniem naciśnięto jakiś klawisz, zwróci ona wartość *true*. Warto pamiętać, że `KeyPressed` nie opróżnia bufora klawiatury, toteż kolejne wywołania będą zwracały *true* aż do chwili, kiedy znak zostanie odczytany np. wywołaniem funkcji `ReadKey`. Z tego też względu funkcji `KeyPressed` należy używać bardzo ostrożnie, zastępując ją w miarę możliwości wywołaniem `ReadKey`.

Realizację efektów dźwiękowych umożliwiają procedury Sound, NoSound i Delay. Pierwsze dwie odpowiednio włączają i wyłączają wewnętrzny głośniczek komputera, powodując generowanie dźwięku o częstotliwości określonej parametrem. Procedura Delay umożliwia ustalenie długości trwania dźwięku. Ponieważ głośnik komputera PC obsługiwany jest sprzętowo, musisz pamiętać o wyłączeniu dźwięku procedurą NoSound, w przeciwnym przypadku może on być generowany nawet po zakończeniu programu. Krótką demonstrację efektów dźwiękowych pokazano poniżej:

```
for i := 1 to 8 do
  begin
    Sound(1000); { dźwięk o częstotliwości 1 kHz }
    Delay(100); { opóźnienie 100 milisekund }
    NoSound; { wyłącz dźwięk }
    Delay(100); { jeszcze jedno opóźnienie }
  end;
```

Procedury i dane zawarte w module Dos wykorzystywane są znacznie rzadziej, głównie przez bardziej doświadczonych programistów. Pozwalają one na wykonywanie operacji na systemie plików, obsługę zegara systemowego oraz zarządzanie procesami (programami) i tzw. przerwaniami (co jest zabawą nie zawsze bezpieczną).

Elementarne informacje o systemie plików możesz uzyskać za pomocą funkcji (procedur) DiskSize, DiskFree, GetDir, FindFirst i FindNext. Pierwsze dwie funkcje zwracają całkowitą pojemność oraz ilość wolnego miejsca dla zadanego dysku (w bajtach), zaś procedura GetDir umożliwia określenie nazwy bieżącego katalogu:

```
GetDir(0, s);  { 0 oznacza dysk bieżący }
writeln('Biezacy katalog: ', s);  { s jest typu string }
writeln('Pojemnosc dysku: ', DiskSize(0), ' bajtow.');
writeln('Wolne miejsce: ', DiskFree(0), ' bajtow.');
```

Warto wiedzieć, że dyski w systemie DOS numerowane są od jedynki (która odpowiada dyskowi A:), zaś wartość 0 oznacza zawsze dysk bieżący.

Procedury FindFirst i FindNext wykorzystywane są na ogół do sprawdzania obecności pliku na dysku i odczytu listy plików zawartych w katalogu. Poniżej przedstawiono typową konstrukcję wyświetlającą listę plików zawartych w katalogu głównym dysku C:

```
FindFirst('c:\*.*', AnyFile, OpisPliku); { znajdź pierwszy }
                                         { plik }
while DosError = 0 do { szukaj aż do wyczerpania plików }
  begin
    FindNext(OpisPliku); { znajdź kolejny plik }
    writeln(OpisPliku.Name);
  end;
```

Procedura FindFirst znajduje pierwszy plik o nazwie zadanej pierwszym parametrem i tworzy tzw. rekord opisu pliku (zmienna OpisPliku typu SearchRec), wykorzystywany następnie przez FindNext do wyszukiwania kolejnych plików. Zmienna DosError przechowuje wynik ostatnio wykonanej funkcji systemu operacyjnego

(w naszym przypadku — funkcji wyszukania pliku), przy czym wartość 0 odpowiada wykonaniu bezbłędnemu, natomiast pozostałe wartości są kodami odpowiednich błędów zwracanymi przez DOS.

Na tym zakończymy nasze krótkie spotkanie z modułami bibliotecznymi. Z konieczności omówiliśmy w nim zagadnienia najbardziej podstawowe, pomijając szczegółowy opis zawartości modułów. Jeśli okaże się, że jest Ci potrzebna jakaś funkcja lub struktura danych, najlepiej zrobisz przeglądając odpowiednie tematy systemu pomocy, ewentualnie odwołując się do literatury.

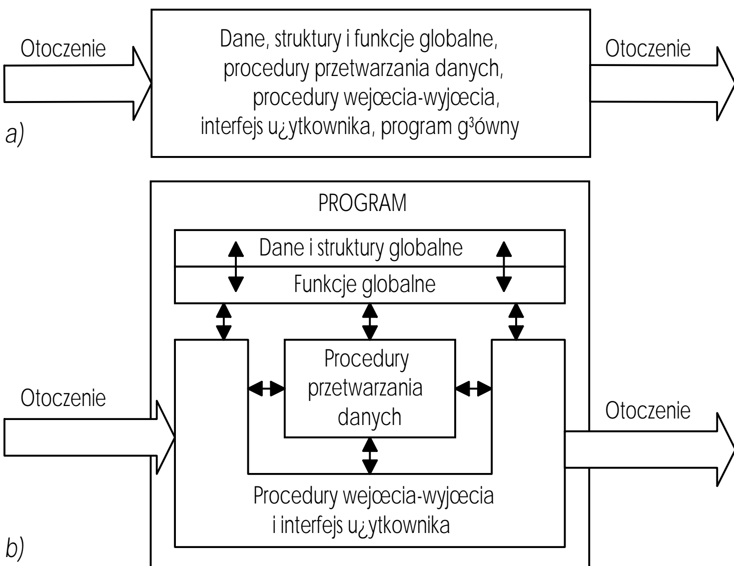
W kolejnym rozdziale pokażemy, jak tworzyć własne moduły i jakie z tego płyną korzyści.

Zapamiętaj

- Pascalowe moduły biblioteczne zawierają wiele użytecznych procedur i struktur danych.
- Aby odwołać się do zawartości modułu, musisz umieścić na początku programu deklarację **`uses`**.
- Wykorzystanie modułów powoduje powiększenie objętości skompilowanego programu.
- Najczęściej wykorzystywanymi modułami są `Crt` (obsługa monitora i klawiatury), `Graph` (obsługa grafiki) oraz `Dos` (obsługa wywołań funkcji systemowych).

Moduły własne

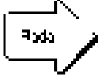
W miarę rozwoju Twoich umiejętności będziesz rozwiązywał coraz bardziej złożone problemy i pisał coraz bardziej złożone programy. Prędzej czy później staniesz też przed koniecznością rozbicia programu na kilka prostszych fragmentów, czyli podzielenia go na *moduły*. Modułowość, będąca jedną z podstawowych zasad dobrego programowania, jest rozszerzeniem podejścia proceduralnego. Struktura modularna zakłada wydzielenie funkcji zajmujących się daną dziedziną (np. operacjami wejścia-wyjścia) i zgrupowanie ich w oddzielnych plikach. Różnicę pomiędzy programem „liniowym" a programem modułowym realizującym to samo zadanie przedstawia poniższy rysunek.



Rysunek 13. *Struktura programu „liniowego" (a) i modułowego (b)*

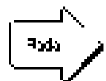
Co daje zastosowanie struktury modułowej? Jak nietrudno zauważyć, program „liniowy" grupuje wszystkie elementy w jednym pliku źródłowym. Przy większych programach liczba wierszy kodu źródłowego idzie w tysiące, co z kolei drastycznie wydłuża czas potrzebny na znalezienie żądanego fragmentu programu, pogarsza jego czytelność

i utrudnia interpretację i uruchamianie. Odpowiedni program o strukturze modułowej składa się z kilku krótszych (a więc łatwiejszych do czytania) plików zawierających funkcje pogrupowane tematycznie (co z kolei ułatwia wyszukiwanie odpowiednich fragmentów programu i jego uruchamianie).



Pierwszą korzyścią z zastosowania struktury modułowej jest poprawa czytelności kodu źródłowego oraz ułatwienie interpretacji i uruchamiania programu.

Nie dość tego, raz napisane i przetestowane funkcje i struktury danych mogą przydać Ci się w przyszłości. Oczywiście podczas pisania kolejnego programu możesz wrócić do odpowiedniego tekstu źródłowego i „potraktować go" edytorem, kopiując odpowiednie fragmenty, jednak znacznie lepiej jest zamknąć raz napisany kod w module. Dlaczego? Po pierwsze, unikasz w ten sposób „przekopywania się" przez stare programy, wyszukiwania odpowiednich fragmentów i złożonych nieraz operacji w edytorze, zastępując to wszystko zwykłym dołączeniem modułu. Po drugie, programowanie z użyciem bibliotek przypomina układanie klocków (bez konieczności zastanawiania się, z czego zostały one zrobione i jak działają — fachowo nazywa się to *ukrywaniem szczegółów implementacyjnych*). Po trzecie, wykorzystanie wcześniej napisanych i przetestowanych procedur i funkcji ułatwia uruchamianie programu (zakładając, że klocki zostały zrobione solidnie). Podejście modułowe wymusza również na programiście poprawne projektowanie elementów programu, co bez wątpienia przyniesie profity w przyszłości. Nie bez znaczenia jest też kwestia czasowa (raz skompilowany moduł nie kompiluje się ponownie). Wykorzystanie modułów daje wreszcie programistom możliwość rozpowszechniania kodu w postaci skompilowanych bibliotek, bez konieczności ujawniania tekstów źródłowych.



Drugą korzyścią z zastosowania struktury modułowej jest możliwość łatwego wykorzystania wcześniej stworzonych i przetestowanych funkcji i struktur danych (ang. reusability) oraz ułatwienie ich dystrybucji.

Jak już powiedzieliśmy, konieczność programowania modułowego przyjdzie z czasem — programy pisane przez Ciebie obecnie są nieco za krótkie. Dlatego też na razie ograniczymy się do przedstawienia ogólnych reguł tworzenia modułów i ich wykorzystania. Za przykład posłuży nam krótki moduł `Test`:

```
unit Test;

interface { sekcja publiczna }

type
  float = real;

function Pow(x, y : float) : float;
{ podnosi liczbę x do potęgi y }

function IleWywolan : word;
{ zwraca liczbę wywołań funkcji Pow }
```

```
implementation { sekcja prywatna }

var
  Licznik : word; { licznik wywołań funkcji }

function Pow(x, y : float) : float;

begin
  Inc(Licznik);
  Pow := exp(y*ln(x));
end;

function IleWywolan : word;

begin
  IleWywolan := Licznik;
end;

begin { sekcja inicjalizacji }
  Licznik := 0;
end.
```

Moduł ten zawiera trzy zasadnicze sekcje: publiczną (**`interface`**), prywatną (**`implementation`**) oraz inicjalizacyjną.

Sekcja publiczna, otwierana słowem **`interface`**, definiuje obiekty, które będą „eksportowane” z modułu na zewnątrz. Zawiera on definicje stałych i typów oraz deklaracje zmiennych, a także tzw. *deklaracje zapowiadające* procedur i funkcji (czyli po prostu ich nagłówki). Sekcja publiczna stanowi jedyną bramę, przez którą program korzystający z modułu może dostać się do jego zawartości: wszelkie obiekty zdefiniowane wewnątrz modułu lecz nie zadeklarowane w sekcji publicznej, będą niewidoczne na zewnątrz. Warto również pamiętać, że komunikacja z modułem jest wyłącznie jednostronna, tj. obiekty zdefiniowane w module nie mogą odwoływać się do obiektów wykorzystującego go programu. Możliwe jest natomiast wykorzystanie w module innego modułu (w tym celu po nagłówku modułu należy umieścić odpowiednią deklarację **`uses`**).

Definicje obiektów zapowiedzianych w sekcji publicznej zawiera sekcja prywatna, otwierana słowem **`implementation`**. Jej zawartość jest niewidoczna na zewnątrz, co pozostaje w zgodzie z ideą ukrywania szczegółów implementacyjnych: obiekty prywatne wykorzystywane są najczęściej do wewnętrznych potrzeb modułu, tak więc nie ma sensu ujawnianie ich na zewnątrz. Możliwe jest co prawda odwoływanie się do funkcji i procedur zadeklarowanych w sekcji **`interface`**, ale „widoczność” sprowadza się tutaj wyłącznie do znajomości posłużenia się daną funkcją — wywołujący program musi znać jej nagłówek (określający sposób przekazania parametrów i odebrania wyniku), natomiast znajomość treści nie jest mu do niczego potrzebna.

Ostatnią sekcją modułu jest nieobowiązkowa sekcja inicjalizacji, rozpoczynająca się (podobnie, jak w zwykłym programie) słowem kluczowym **`begin`**. Instrukcje zawarte w sekcji inicjalizacji są wykonywane w chwili uruchomienia programu i zwykle wyko-

rzystywane są (jak sama nazwa wskazuje) do automatycznego ustalania jego stanu początkowego (np. zapamiętywania kolorów ekranu w celu ich późniejszego odtworzenia, inicjalizacji drukarki, automatycznego ładowania plików konfiguracyjnych).

Całość modułu rozpoczyna się nagłówkiem o postaci **unit** nazwa (ang. *unit* — moduł) i kończy słowem kluczowym **end** (z kropką). Tu uwaga: treść modułu (kod źródłowy i wynikowy) zapamiętywana jest w oddzielnych plikach, których nazwy muszą być takie same, jak nazwa użyta w nagłówku (w naszym przypadku tekst źródłowy modułu należy umieścić w pliku TEST.PAS, zaś kod wynikowy zostanie zapisany do pliku TEST.TPU). Wymienione wyżej sekcje powinny występować w takiej kolejności, w jakiej zostały opisane, chociaż każdą z nich można pominąć. Pospolitym zjawiskiem jest pominięcie części inicjalizacyjnej (w chwili uruchomienia programu moduł nie wykonuje żadnych operacji). Pominięcie części prywatnej spotykane jest głównie w modułach deklarujących wyłącznie stałe, zmienne i typy. Brak części publicznej automatycznie uniemożliwia komunikację z modułem, toteż sytuacja taka spotykana jest niezwykle rzadko.

Wróćmy do naszego przykładu. Moduł Test zawiera w części publicznej deklarację dwóch funkcji: Pow, podnoszącej liczbę rzeczywistą do potęgi rzeczywistej, oraz (znacznie mniej użytecznej) IleWywolan, zwracającej liczbę wywołań funkcji Pow. Sekcja publiczna zawiera również definicję prostego typu zmiennoprzecinkowego float.

Sekcja prywatna zawiera oczywiście definicje funkcji zapowiedzianych w sekcji publicznej, jak również deklarację prywatnej zmiennej Licznik. Zmienna ta przechowuje liczbę wywołań funkcji Pow. Ponieważ Licznik jest niedostępny z zewnątrz (spróbuj!), jego wartość zwracana jest przez funkcję IleWywolan. Warto tu wspomnieć, że wszelkie globalne zmienne prywatne modułu są statyczne, tj. przechowują swoją wartość przez cały czas wykonywania programu.

Część inicjalizacyjna zawiera jedną instrukcję, inicjalizującą wartość zmiennej Licznik na zero. O sensowności tego rozwiązania nie trzeba chyba nikogo przekonywać.

Do sprawdzenia działania naszego modułu wykorzystamy krótki program TestModulu. Program ten jest na tyle banalny, że nie wymaga komentarza.

```
program TestModulu;

uses
  Test;

var
  x : float; { wykorzystanie zdefiniowanego typu }

begin
  writeln('2 do potegi 8 = ', Pow(2,8):8:4);
  x := 1.5;
  writeln('Pi do potegi 1.5 = ', Pow(Pi, x):8:4);
  writeln('Funkcje Pow wywolano ', IleWywolan, ' razy.');
end.
```

Jak widać, wykorzystanie własnego modułu odbywa się dokładnie tak samo, jak standardowego modułu bibliotecznego. Jeśli przyjrzysz się uważnie raportowi z kompilacji zauważysz, że przed skompilowaniem samego programu Turbo Pascal wyświetli informację o kompilacji modułu. Po skompilowaniu programu na dysku powinien pojawić się plik o nazwie TEST.TPU, zawierający skompilowany kod wynikowy modułu. Odpowiednie fragmenty kodu są pobierane z pliku i dołączane do kodu wynikowego programu w procesie konsolidacji. Po utworzeniu pliku .TPU moduł nie będzie kompilowany ponownie, o ile nie wprowadzisz zmian do kodu źródłowego lub nie wydasz polecenia ***Build***, wymuszającego bezwarunkowe skompilowanie wszystkich elementów składowych programu.

Jak powiedzieliśmy na początku tego rozdziału, zagadnienie modułów jest raczej „przyszłościowe", toteż nie będziemy się nim szerzej zajmować. Od samej strony technicznej (której opis możesz znaleźć w literaturze) ważniejsza wydaje się idea programowania (i projektowania) modułowego, pozwalająca na szybkie i skuteczne tworzenie efektownych, efektywnych i bezbłędnych programów.

Zapamiętaj

- Zasada modularności jest jedną z podstawowych zasad dobrego programowania. Stosowana jest ona przede wszystkim podczas realizacji bardziej złożonych zadań.
- Realizację tej zasady umożliwiają w Turbo Pascalu moduły.
- Moduł zawiera definicje i deklaracje obiektów zawarte w trzech sekcjach. Sekcja publiczna (**`interface`**) określa, które obiekty będą widoczne na zewnątrz, sekcja prywatna (**`implementation`**) definiuje ich treść (a także definiuje obiekty prywatne modułu), natomiast sekcja inicjalizacyjna przeznaczona jest do automatycznego wykonywania zadanych czynności w chwili rozpoczęcia programu.
- Obiekty zdefiniowane w sekcji prywatnej są niewidoczne na zewnątrz modułu, o ile nie zostaną zadeklarowane w sekcji publicznej.
- Skompilowana treść modułów jest przechowywana w plikach o rozszerzeniu .TPU i dołączana do właściwego programu w procesie konsolidacji.
- Podział programu na moduły oraz grupowanie funkcji, procedur i struktur danych w biblioteki gotowych elementów umożliwia szybkie i skuteczne tworzenie efektywnych i bezbłędnych programów.

Jak uruchamiać oporne programy

Im bardziej złożony program, tym mniejsza szansa, że uda Ci się od razu napisać go bezbłędnie. Wyłapanie wszystkich błędów kompilacji to dopiero początek: zawsze może Ci się przytrafić dzielenie przez zero czy próba otwarcia nieistniejącego pliku. Błędy te, zwane błędami wykonania (ang. *runtime errors*), są znacznie mniej przyjemne i trudniejsze do usunięcia od błędów kompilacji. Ich lokalizacja wymaga na ogół użycia specjalnego narzędzia uruchomieniowego, zwanego z angielska *debuggerem* (czyli odpluskwiaczem). System uruchomieniowy Turbo Pascala będzie tematem ostatniego już rozdziału poświęconego temu kompilatorowi.

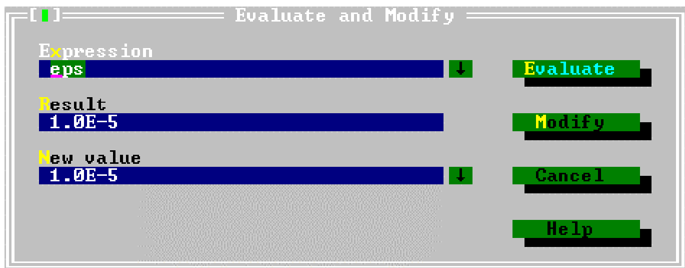
Polecenia systemu uruchomieniowego zgrupowane są w menu ***Run*** oraz ***Debug***, przy czym większości z nich odpowiadają klawisze skrótu. Do eksperymentów z debuggerem można wykorzystać praktycznie dowolny program, chociaż warto, by zawierał on definicje procedur i zmiennych lokalnych. Dla naszych potrzeb odpowiedni będzie przedstawiony na stronie 62 program `Bisekcja`.

Jak już wiesz, wystąpienie błędu wykonania powoduje przerwanie działania programu i wskazanie kursorem instrukcji, która ten błąd spowodowała. Niestety, na ogół informacja taka jest niewystarczająca, gdyż do skutecznej lokalizacji błędu wymagane jest prześledzenie zachowania programu na co najmniej kilka instrukcji przed jego wystąpieniem. Możliwość taką daje w nowoczesnych narzędziach uruchomieniowych tzw. *tryb krokowy*, pozwalający na wykonywanie programu instrukcja po instrukcji.

Turbo Pascal pozwala na krokowe wykonywanie programu na dwa sposoby, realizowane odpowiednio poleceniami ***Trace Into*** i ***Step Over*** z menu ***Run*** (lub odpowiadającymi im klawiszami F7 i F8). Różnica pomiędzy obydwoma trybami sprowadza się do innego sposobu traktowania procedur i funkcji: polecenie ***Trace Into*** pozwala na „wejście" do wnętrza procedury, zaś ***Step Over*** wykonuje ją jako jedną instrukcję. W obu przypadkach aktualnie wykonywana instrukcja zostaje wyróżniona w tekście programu kolorowym paskiem. Aby wypróbować działanie trybu krokowego, skompiluj program `Bisekcja` (F9) i wykonaj go w obu trybach. Zauważ też, że

wydanie polecenia ***Program Reset*** (CTRL-F2) pozwala na zrestartowanie programu, co może się przydać gdy się okaże, że dalsze jego wykonywanie nie ma sensu.

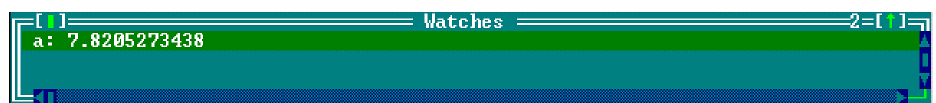
Powyższa wprawka nie przyniosła chyba żadnych niespodzianek i... niewiele informacji. Tryb krokowy, pozwalający na ustalenie drogi, jaką „przebywa” wykonanie programu, nie daje żadnych wskazówek na temat wartości przyjmowanych przez zmienne, które w większości przypadków są odpowiedzialne za sterowanie pracą programu, a więc i ewentualne kolizje. Na szczęście podejrzenie zawartości wybranej zmiennej jest bardzo łatwe: wystarczy do tego polecenie ***Evaluate/modify*** (CTRL-F4) z menu ***Debug***. Jego wydanie powoduje wyświetlenie okienka zawierającego informację o wartości zmiennej oraz pozwalającego na jej zmianę (tak!).



Rysunek 14. *Pole dialogowe* ***Evaluate and Modify***

W pole ***Expression*** wpisujemy wyrażenie (np. nazwę zmiennej, w naszym przypadku `eps`), którego wartość chcemy obejrzeć. Wartość wyświetlana jest w polu ***Result***, zaś pole ***New value*** umożliwia jej zmianę (z czym jednak należy nieco uważać).

Czasami okazuje się, że podejrzaną zmienną należy śledzić cały czas. W takiej sytuacji zamiast mało wygodnego podglądania poleceniem ***Evaluate*** lepiej jest użyć polecenia ***Watch*** (CTRL-F7). Po jego wydaniu (i wpisaniu nazwy odpowiedniej zmiennej lub wyrażenia w okienku ***Add Watch***) na dole ekranu pojawi się okienko ***Watches***, zawierające wartości śledzonych zmiennych (w naszym przypadku zmiennej `a`, czyli lewej granicy przedziału poszukiwań pierwiastka).



Rysunek 15. *Okienko* ***Watches***

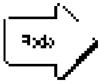
Ponieważ powrót do okienka edytora zwykle powoduje przysłonięcie okienka ***Watches***, warto uporządkować układ okienek na ekranie, np. przesuwając je za pomocą myszki lub wydając polecenie ***Tile*** z menu ***Window***.

Dysponując tymi wiadomościami możesz już wykorzystać tryb krokowy i polecenia podglądania zmiennych do prześledzenia zachowania zmiennej `c`, będącej bieżącą wartością pierwiastka.

W przypadku większych programów może się okazać, że dotarcie do fatalnej instrukcji z wykorzystaniem trybu krokowego jest zbyt czasochłonne. W takich sytuacjach z pomocą przychodzi polecenie ***Go to cursor*** (F4), powodujące wykonanie wszystkich instrukcji aż do miejsca wskazanego kursorem, a następnie przejście do pracy krokowej. Wykorzystując je możesz łatwo „wskoczyć" np. do wnętrza funkcji *f(x)* bez konieczności wykonywania instrukcji poprzedzających jej wywołanie.

Jeśli powyższe czynności masz wykonywać wielokrotnie, znacznie bardziej użyteczne od funkcji ***Go to cursor*** okazuje się polecenie ***Add breakpoint*** z menu ***Debug*** (CTRL-F8). Jego wydanie pozwala na ustawienie w miejscu wskazanym kursorem tzw. *punktu wstrzymania*, powodującego zatrzymanie programu po każdorazowym jego osiągnięciu. Nie dość tego, dodatkowe parametry punktu wstrzymania (***Condition*** i ***Pass count***) umożliwiają jego warunkowe wykonywanie lub zignorowanie określonej liczby przejść. Opcje te używane są rzadko i nie będziemy ich tu omawiać; w większości przypadków użycie punktów wstrzymania sprowadza się do ich ustawiania i usuwania za pomocą klawiszy CTRL-F8 (odpowiednia instrukcja zostanie wyróżniona w treści programu kolorowym paskiem). Po dojściu programu do punktu wstrzymania na ogół wystarczy sprawdzić zawartość podejrzanej zmiennej poleceniem ***Evaluate*** lub ***Add Watch***.

Skrótowe omówienie poleceń systemu uruchomieniowego kończy nasze wprowadzenie do programowania i Turbo Pascala. Dalsze wiadomości zdobędziesz korzystając z bardziej zaawansowanej literatury, a przede wszystkim na drodze praktycznej — pisząc programy. Pamiętaj:



Korzystanie z komputera nie zwalnia od myślenia.

Im większy nacisk położysz na właściwe zaprojektowanie rozwiązania, tym mniej czasu będziesz musiał poświęcić na jego zaprogramowanie i tym mniejsze szanse, że będziesz musiał odwoływać się do pomocy środków uruchomieniowych...

Powodzenia!

Literatura

1. A. Marciniak: *Turbo Pascal 7.0*, NAKOM, Poznań, 1995
2. Tomasz M. Sadowski: *Praktyczny kurs Turbo Pascala*, Helion, Gliwice 1991, 1993.
3. J. Zahorski: *Turbo Pascal 7.0*, Helion, Gliwice 1995.