Міністерство освіти та науки України

Технічний коледж
Тернопільського державного технічного університету

Тхір І.Л., Юзьків А.В.

Курс лекцій по Turbo Pascal 7.0

СМП "Астон"
Тернопіль 2001

Тхір І.Л., Юзьків А.В.
Курс лекцій по Turbo Pascal 7.0. -Тернопіль: Технічний коледж ТДТУ, 2001, -с.144.

"Курс лекцій по Turbo Pascal 7.0" призначений для користувачів ПК, які хочуть отримати початкові навички у програмуванні мовами високого рівня. Даний посібник побудовано на основі конспекту лекцій для студентів спеціальностей комп'ютерного напрямку Технічного коледжу.

Він містить велику кількість прикладів написання реальних програм, тому може застосовуватися як посібник для самостійного навчання. Використовуючи отримані базові знання ви надалі зможете вдосконалити свій рівень програміста в інших мовах програмування.

Розглянуто і схвалено на засіданні циклової комісії "Інформатики і ОТ".
Протокол №2 від 19.10.2000 року.

Рекомендовано до друку Педагогічною радою Технічного коледжу ТДТУ.
Протокол №5 від 02.11.2000 року.

Зміст

Вступ 5

Розділ 1. Короткий огляд інтерфейсу оболонки Turbo Pascal 7

Розділ 2. Алфавіт мови Turbo Pascal. Типи даних 10

Розділ 3. Структура Pascal-програми. Правила написання програм 12

Розділ 4. Прості типи даних у Turbo Pascal 16

4.1. Цілі числа 16
4.2. Дійсні числа 19
4.3. Символьний тип 22
4.4. Логічний тип 23
4.5. Інтервальний тип даних 24
4.6. Перераховний тип даних 25

Розділ 5. Оператори мови Turbo Pascal 26

5.1. Введення та виведення інформації у Pascal-програмах 26
5.2. Оператор присвоєння в Turbo Pascal 27
5.3. Оператор безумовного переходу 29
5.4. Оператор перевірки умови (розгалуження) 29
5.5. Оператор вибору 31
5.6. Оператори циклу 32
5.6.1. Цикл із наперед заданою кількістю повторень 33
5.6.2. Цикл із передумовою 34
5.6.3. Цикл із післяумовою 35

Розділ 6. Робота з масивами в Turbo Pascal 37

6.1. Поняття про масиви. Класифікація масивів 37
6.2. Операції над одномірними масивами (рядами) 37
6.3. Операції над двомірними масивами (матрицями) 40

Розділ 7. Використання підпрограм у Turbo Pascal 45

7.1. Класифікація підпрограм. Глобальні та локальні ідентифікатори
Параметри значення та параметри змінні 45
7.2. Підпрограми-функції. Приклади створення 46
7.3. Підпрограми-процедури. Приклади створення 48
7.4. Рекурсивні підпрограми 50

Розділ 8. Робота з множинами в Turbo Pascal 52

8.1. Поняття про множини. Операції над множинами 52
8.2. Приклади типових програм по обробці множин 53

Розділ 9. Стрічки в Turbo Pascal 56

9.1. Поняття про стрічки. Стандартні процедури та функції
для роботи із стрічками 56
9.2. Приклади типових програм по обробці стрічок 28

Розділ 10. Записи в Turbo Pascal **60**

Розділ 11. Тип даних, файли **62**

11.1. Класифікація файлових типів у Turbo Pascal 62

11.2. Файли вказаного типу. Стандартні процедури та функції для роботи з ними 64

11.3. Текстові файли. Стандартні процедури та функції для роботи з ними 70

11.4. Файли без типу. Стандартні процедури та функції для роботи з ними 73

Розділ 12. Вказівниковий тип даних. Робота з динамічними змінними **75**

Розділ 13. Тип даних - списки **82**

13.1. Лінійні списки 82

13.2. Стеки та черги 85

13.3. Нелінійні списки 87

Розділ 14. Модулі в Turbo Pascal **89**

14.1. Типи модулів. Створення модуля користувача 89

14.2. Модуль CRT 94

14.3. Модуль Graph 98

14.4. Модуль Strings 114

14.5. Модуль DOS 117

14.6. Модуль WinDOS 129

14.7. Модуль Overlay 132

14.8. Модуль Printer 138

Додаток. Повідомлення про помилки **139**

Рекомендована література **143**

Вступ

Як відомо, для того, щоб комп'ютер міг виконувати потрібні нам задачі, ним повинна керувати програма. Основна відмініть ЕОМ, в тому числі ПК, від інших пристроїв полягає в тому, що ним керує програма.

Комп'ютерна програма - це набір інструкцій, які вказують ЕОМ послідовність дій по вирішенню певної поставленої перед ним задачі.

Для того, щоб комп'ютер „зрозумів" вказівки програміста, він повинен „спілкуватись" із програмістом на спільній мові, яку називають **мовою програмування**.

На сьогоднішній день у світі існують сотні різних мов програмування, які в загальному можна поділити на мови програмування низького (**Asembler**) та високого рівня **(Pascal, Delphi, C** і т.д.). Крім цього мови програмування можна поділити на функціональні, структурні, логічні, процедурні, об'єктно-орієнтовані.

Мова Pascal є однією із класичних мов програмування. Найбільшої популярності вона досягла у 80-х - першій половині 90-х років. На її основі розроблено цілий ряд сучасних мов програмування, таких як Borland Delphi.

Мову програмування Pascal розроблено в 1968 році на кафедрі інформатики Стенфордського університету швейцарським ученим Ніклаусом Віртом. Ця мова програмування отримала назву на честь відомого французького математика й філософа Блеза Паскаля (1623-1662). Спочатку вона розглядалась як навчальна мова програмування і завдяки простоті використання застосовувалась у навчальних закладах як „перша" мова програмування для студентів. В 1983 році фірма Borland International перевела її на комерційну основу в якості мови програмування для персональних комп'ютерів. Переломним етапом для мови Pascal став 1985 рік, коли появилась її нова версія, яку назвали Turbo Pascal 3.0. Вона містила оболонку для зручного написання та редагування тексту програм та компілятор стандартного Pascal. З цього часу Pascal набув широкого використання в колах як досвідчених програмістів так і початківців.

В наступній версії Turbo Pascal 4.0 було усунуто багато обмежень компілятора, що часто піддавались критиці, та підвищено продуктивність системи. Найбільш важливим нововведенням стало застосування модульної концепції (UNIT-концепції), що була запозичена з мови програмування Modula-2. Це дозволило реалізувати в Turbo Pascal розробку великих програмних продуктів.

З появою версії 5.0 Turbo Pascal отримала ще більші можливості для використання професійними програмістами. Одним із важливих нововведень стало використання апаратних перекриттів або оверлеїв (**overlays**). Вони дозволили створювати потужні програмні продукти, які розраховані на використання при малих об'ємах пам'яті. Механізм оверлеїв полягає в поділі програми на частини, що почергово завантажуються в пам'ять (з дискети або жорсткого диска) по мірі необхідності. Крім цього в Turbo Pascal 5.0 розширено можливості відлагодження програм і забезпечено можливість підтримки розширеної пам'яті.

Ще одним важливим етапом для Turbo Pascal була поява версії 5.5. Саме в цій версії з'явилась концепція **об'єктно-орієнтованого програмування (ООП)**. Фактично мова

Pascal стала засновником цього напрямку розвитку мов програмування. ООП полягає у використанні спеціального типу даних - **об'єктів (Object)**, які спрощують процес програмування, роблячи тексти програм значно компактнішими. В результаті чого з'явився термін **Object Pascal**. Паралельно з цим, об'єктно-орієнтоване програмування розвивається і в іншій популярній мові програмування **C**, в результаті чого з'являється **C++**. На основі Object Pascal корпорація Borland розробила нову мову програмування Delphi, яка набула великої популярності в другій половині 90-х років.

Популярність цієї мови пояснюється ще й тим, що вона належить до мов **візуального програмування**. Мови візуального програмування значно полегшують процес програмування інтерфейсу створюваних програм. Адже зовнішній вигляд вікон та інших об'єктів програміст створює лише з допомогою миші та панелей об'єктів-заготовок. Серед поширених мов візуального програмування можна також відмітити Visual Basic - це мова, на якій створюють макроси для документів дуже популярного офісного пакета Microsoft Office.

В даному посібнику розглянуто основні можливості програмування на стандартній мові програмування Turbo Pascal версії 7.0. Цей посібник розрахований перш за все на початківців, які тільки знайомляться із світом програмування. Знання Pascal допоможуть їм набути базових навичок програмування та стануть основою для подальшого вивчення більш потужних мов, таких як Delphi.

Turbo Pascal 7.0, залишаючись сумісним із попередніми версіями Turbo Pascal та Turbo Pascal for Windows, надає нам ряд нових можливостей. Найважливішою з них є наявність значно швидшого компілятора програм, який став своєрідним світовим стандартом для компіляторів.

Компілятор - це спеціальна програма (програмний модуль), яка здійснює перетворення тексту програми, написаного на відповідній мові програмування, у програмний код, зрозумілий комп'ютеру. Сам процес перетворення тексту програми, у програмний код називається **компіляцією**.

Потрібно відмітити, що при компіляції відбувається автоматичний пошук компілятором помилок у тексті програми.

Розділ 1. Короткий огляд інтерфейсу оболонки Turbo Pascal

Для створення тексту програм у Borland Pascal служить програмна оболонка, яка включає в себе текстовий редактор для введення (редагування) тексту програми та компілятор для створення програмного коду. Пакет Borland Pascal включає в себе п'ять різних варіантів завантаження оболонки середовища Borland Pascal:

- **bp.exe** - інтегроване середовище розробки програм (IDE), що працює в захищеному режимі DOS і генерує прикладні програми DOS в реальному режимі DOS та захищеному режимі Windows;
- **turbo.exe** - інтегроване середовище, що працює в реальному режимі DOS і генерує тільки прикладні програми в реальному режимі;
- **bpw.exe** - інтегроване середовище, що працює під Windows і генерує прикладні програми DOS і Windows реального та захищеного режиму;
- **bpc.exe** - компілятор, що працює в режимі командного рядка в захищеному режимі DOS і генерує прикладні програми і Windows реального та захищеного режиму;
- **tpc.exe** - компілятор, що працює в режимі командного рядка в реальному режимі DOS і генерує тільки прикладні програми DOS реального режиму.

Після завантаження одного із трьох перших описаних вище файлів відкривається вікно оболонки, що містить робоче поле та головне меню. Крім цього у версії для Windows буде присутнє піктографічне меню. В будь-якому із розглянутих типів оболонки є можливості роботи як із клавіатурою, так і мишею.

Для відкриття файлу з текстом програми, або створення нової програми, потрібно натиснути клавішу **F3,** або в пункті меню **File** вибрати команду **Open** (**New** для створення файлу програми). Потрібно зауважити, що для активізації головного меню слід натиснути клавішу **F10,** або Alt і першу літеру назви пункту меню.

В будь-якому випадку відкривається вікно, де в полі **Name** слід ввести ім'я потрібного файлу, або вибрати його у списку **Files**. Якщо файл із таким іменем існував, то він відкриється в окремому вікні, а якщо ні, то з'явиться порожнє вікно, для створення нової програми. Потрібно зауважити, що розширення файлу Borland Pascal надає автоматично - PAS.

Середовище Borland Pascal підтримує багатовіконний інтерфейс, надаючи кожному відкритому для редагування файлу окреме вікно. Перехід між вікнами здійснюється клавішею **F6** (в пункті меню Window вибрати команду **Next**) для переходу до наступного вікна, та **Shift+F6** (у пункті меню Window вибрати команду **Previous**) - до попереднього вікна. Можна також вибрати команду **List** із цього ж пункту меню і в списку вікон вибрати потрібне. Крім цього кожне відкрите вікно має свій номер і для переходу в це вікно можна натиснути комбінацію клавіш **Alt**+цифру, що відповідає номеру вікна.

Для того, щоб закрити активне вікно з файлом, необхідно натиснути комбінацію клавіш **Alt+F3,** або в пункті **Window** вибрати команду **Close**. Можна також натиснути ліву клавішу миші на спеціальному мітчику в лівому верхньому куті вікна.

У випадку, коли необхідно розгорнути вікно на весь екран, потрібно натиснути клавішу **F5,** або в пункті **Window** вибрати команду **Zoom**. Таким самим чином вікно переводиться із повноекранного режиму у звичайний. Для того, щоб змінити розміри вікна, або його положення на екрані, потрібно натиснути комбінацію клавіш **Ctrl+F5** (в пункті **Window** вибрати команду **Move/Size**), а тоді клавішами стрілок перемістити вікно в певне положення на екрані (для зміни розмірів потрібно використовувати клавіші стрілок у комбінації з **Shift**). Після встановлення потрібних параметрів, слід натиснути клавішу **Enter**. Для зміни розмірів вікна мишею потрібно зафіксувати її курсор в правому нижньому куті вікна, натиснути ліву клавішу і перетягнути в потрібне місце. Щоб змінити положення вікна, потрібно перетягнути його за заголовок.

Для того, щоб впорядкувати розміщення відкритих вікон, їх можна розмістити каскадом (в пункті меню **Window** команда **Cascade**) або черепицею (у пункті меню **Window** команда **Tile**).

Після того, як у файлі програми зроблено зміни, їх необхідно записати на диск. Для цього потрібно натиснути клавішу **F2,** або в пункті меню File вибрати команду **Save**. Якщо до цього файл був без імені, то додатково з'явиться вікно, де слід указати каталог, куди його буде записано, та ім'я самого файлу.

Для виходу з оболонки Borland Pascal потрібно натиснути комбінацію клавіш Alt+X, або в пункті меню **File** вибрати команду **Exit**.

Під час редагування тексту програми досить часто виникає потреба в копіюванні та перенесенні окремих частин тексту. Для цього відповідний блок програми необхідно виділити. Щоб виділити частину тексту, спочатку потрібно встановити курсор на початок (кінець) цього блоку і натиснути комбінацію клавіш **Shift** та відповідні клавіші стрілок. Можна також натиснути ліву клавішу миші і, не відпускаючи її, перетягнути вздовж потрібного тексту.

Для того, щоб скопіювати виділений текст в буфер пам'яті, потрібно натиснути комбінацію клавіш **Ctrl+Insert** (у пункті меню **Edit** вибрати команду **Copy**). Щоб вирізати виділений блок в пам'ять, необхідно натиснути комбінацію клавіш **Shift+Delete** (у пункті меню **Edit** вибрати команду **Cut**). Після того, слід перемістити курсор у відповідне місце тексту програми і натиснути комбінацію клавіш **Shift+Insert** (у пункті меню **Edit** вибрати команду **Paste**). В результаті вміст буфера пам'яті буде поміщено в позицію курсору. Якщо користувачу потрібно переглянути вміст буфера пам'яті, то в пункті меню **Edit** слід вибрати команду **Show clipboard**.

Щоб знищити виділений блок, не поміщаючи його в буфер пам'яті, необхідно натиснути комбінацію клавіш **Ctrl+Delete** (у пункті меню **Edit** вибрати команду **Clear**). Якщо потрібно знищити лише один рядок, то можна встановити на нього курсор і натиснути комбінацію клавіш **Ctrl+Y**.

Для того, щоб зняти виділення блоку, можна скористатись комбінацією клавіш **Ctrl+B**, а тоді клавіша **H** (латинський алфавіт).

В Turbo Pascal розвинутий механізм контекстного меню (нагадаємо, викликається правою клавішею миші), яке дозволяє пришвидшити виклик команд оболонки. Крім миші це меню можна викликати комбінацією клавіш **Alt+F10**.

Якщо користувач випадково виконав непотрібну йому операцію редагування тексту, то її можна відмінити комбінацією клавіш **Alt+Backspace (**або в пункті меню **Edit** у контекстному меню вибрати команду **Undo**). Повторити відмінену дію можна командою **Redo** із цього ж пункту меню.

Після того, як текст програми буде введено, її потрібно відкомпілювати (див. вище), щоб створити програмний код. Для цього слід натиснути комбінацію клавіш **Alt+F9,** або в пункті меню **Compile** вибрати однойменну команду. При цьому програма може компілюватись або безпосередньо в оперативну пам'ять, або на диск (з утворенням виконуючого файлу з розширенням exe). Для цього, щоб вибрати напрямок компілювання, потрібно в пункті меню **Compile** вибрати команду **Destination ...** Причому, якщо в меню знаходиться команда **Destination Disk**, то компіляція відбувається в пам'ять і вибір цієї команди встановить компіляцію в exe-файл. А якщо в меню знаходиться команда **Destination Memory**, то компіляція відбувається на диск і вибір цієї команди встановить компіляцію в пам'ять.

Якщо дана програма включає в себе не тільки вихідний код в активному вікні, а складається з декількох файлів (наприклад, основний файл, один або більше модулів, зовнішні модулі на мові Asembler), то можна сформувати свою програму, прокомпілювавши всі ці файли разом. Для цього в пункті меню **Compile** потрібно вибрати **Make,** або натиснути клавішу F9.

Для завантаження відкомпільованої програми на виконання (з оболонки Borland Pascal) потрібно натиснути комбінацію клавіш **Ctrl+F9** або в пункті меню **Run** вибрати однойменну команду. При чому, якщо програма до цього часу не була відкомпільована, вона попередньо компілюється.

Інколи, при компіляції програми важко знайти помилку в тексті програми, тоді рекомендується виконувати програму покроково. Для цього можна скористатись клавішею **F7** (команда **Trace into** у пункті меню **Run**). В результаті програма буде виконуватись по одній команді при кожному натискуванні клавіші F7 (виборі команди). При цьому відповідний рядок програми (що виконується) буде виділятись іншим кольором. Подібною до попередньої команди є команда **Step over** (клавіша **F8**) із пункту меню Run, але вона виконує всі підпрограми як єдине ціле, не розбиваючи їх покомандно.

Розділ 2. Алфавіт мови Turbo Pascal. Типи даних.

Як описано вище, комп'ютерні програми створюються з допомогою спеціальних мов, які називаються мовами програмування. Мова програмування, які і будь-яка інша мова, містить свій алфавіт.

Алфавіт мови програмування - це набір констант, типів даних, змінних, стандартних процедур і функцій, операндів та операторів, з яких складається програма.

Розглянемо ці терміни більш детально.

Константи - це дані, значення яких відоме до завантаження програми і не змінюється в процесі її виконання.

В Pascal існують як стандартні константи, так і вказані користувачем. До стандартних констант належить, наприклад, число Pi=3,141592.... Константи, задані користувачем, потрібно вказувати в спеціальному розділі програми (див. наступний розділ посібника).

Змінні - це дані, значення яких вводиться та змінюється під час виконання програми. Усі змінні, що використовуються в Pascal-програмі, потрібно описувати в спеціальному розділі програми (див. наступний розділ посібника)..

Тип даних - це набір даних (констант, змінних, значень функцій і т.д.), які мають спільні характеристики (формат представлення в пам'яті ПК, множина допустимих значень, множина допустимих операцій, що можна використовувати для даного типу).

Типи даних в Pascal поділяються на прості та складні. До простих типів даних відносяться:

- цілі числа;
- дійсні числа;
- символьний тип;
- логічний тип;
- інтервальний тип;
- перераховний тип.

Складні типи даних - це типи, які складаються з елементів, що відносяться до простих типів. До складних типів даних відносяться:

- масиви;
- множини;
- стрічки;
- записи;
- файли;
- динамічні змінні;
- вказівки;
- лінійні списки (стеки, черги);
- нелінійні списки (двійкові дерева, несиметричні дерева, тексти, графи);
- процедурний тип;
- об'єкти.

Крім цього, типи даних у Turbo Pascal можна поділити на впорядковані та невпорядковані. **Впорядковані** - це типи, в яких дані розміщені в певному, наперед визначеному, порядку і кожен з елементів характеризується своїм порядковим номером. **Невпорядковані** - це типи, в яких дані не мають своїх порядкових номерів.

Процедури та **функції** - це підпрограми, що використовуються в середині програми (більш детально будуть описані далі). До алфавіту мови програмування відносять стандартні процедури та функції, тобто такі, що сприймаються мовою програмування без їх додаткового опису.

Операнди - це спеціальні символи або послідовності символів, які виконують над даними певні операції (математичні, логічні і т.д.). Прикладом операндів можуть бути операнди математичних операцій: „+“ - додавання даних), „-“ - віднімання даних, „*“ - множення даних, „/“ - ділення даних, „=“ - рівність даних і т.д.

Оператори - це деякі неподільні елементи програми, що дозволяють виконувати певні алгоритмічні дії у програмі, тобто виконувати в програмі певні команди. Фактично, оператор - це окрема команда в алгоритмі програми, тобто окремий крок виконання програми.

В Turbo Pascal оператори поділяються на прості та структурні. До простих операторів відносяться:

- оператор присвоєння (**:=**);
- оператор безумовного переходу (**goto**);
- оператор звертання до процедури (функції).

Структурними операторами називаються такі, що складаються з інших операторів. До них відносяться:

- складний оператор - представляє собою набір операторів, що поміщені в операторні дужки (**begin - end**);
- умовний оператор (**if**);
- оператор вибору (**case**);
- оператори циклу (**repeat**, **while**, **for**);
- оператор приєднання (**with**).

Усі вищеописані елементи алфавіту мови програмування складаються з окремих символів. В якості символів, що складають елементи алфавіту, в Turbo Pascal можна використовувати більшість символів, що входять в стандартну ASCII-таблицю. Заборонено використовувати символи розширеної ASCII-таблиці, тобто символи з кодами від 128 до 255, а також символи: (&), (!), (%), (~), (“). Ці символи можна використовувати лише в якості коментарів та в текстових стрічках, які беруться в одинарні лапки (’).

Розділ 3. Структура Pascal-програми. Правила написання програм

Програма, що написана на мові Turbo Pascal створюється у відповідності з правилами, що представляють собою дещо розширені й спрощені правила синтаксису стандартного Pascal. Але ці спрощені правила (тобто порядок розміщення в тексті програми різних блоків) повинні строго зберігатись при написанні програми.

Будь-яку програму в Turbo Pascal можна умовно розбити на три основних частини:

- розділ описів та узгоджень;
- розділ текстів, процедур та функцій;
- розділ основного блоку програми.

Потрібно відмітити, що присутність першого та третього розділів є обов'язковим у програмі, тоді як другий (розділ текстів, процедур та функцій) з'являється в програмі по мірі необхідності.

Кожен з вищеописаних розділів поділяється ще на певні підрозділи, деякі з яких є обов'язковими, а деякі вказуються по мірі необхідності. Нижче описано найбільш повну структуру Pascal-програми із вказанням усіх можливих підрозділів. Потрібно відмітити, що підрозділи представлені в квадратних дужках, є необов'язковими і вказуються лише в потрібних випадках (самі квадратні дужки в тексті програми не вказуються). Крім цього необхідно зауважити, що інформація, представлена в програмі у фігурних дужках, є коментарем і при виконані програми ігнорується (крім випадку, коли за відкритою фігурною дужкою стоїть знак „$").

{розділ описів та узгоджень}

```
[program   ім'я програми;          ]
[ {$ ....} ]
[uses      модуль1, модуль2 ...;   ]
[label     мітка1, мітка2 ...;     ]
[const
          ім'я = значення;
                  ,,,              ]
[type
          тип = опис типу;
                  ,,,              ]
var
          змінна1, змінна2, ... : тип;
                  ,,,
```

{розділ текстів процедур та функцій}

```
[procedure    ім'я процедури(список параметрів);
          { тіло процедури }
[              ,,,                    ]]
```

```
[function      ім'я функції(список параметрів):тип результату;
          {   тіло функції    }
[              ,,,                              ]]
```

{розділ основного блоку програми}

```
begin
          {текст програми}
end.
```

В першому розділі програми програміст повідомляє компілятору, якими ідентифікаторами він позначає дані (константи, змінні), а також встановлює власні типи даних, які надалі він сподівається використовувати в програмі. При цьому необхідно слідкувати, щоб імена змінних, констант, назви типів не повторювались для різних даних.

Розглянемо більш детально всі підрозділи, що можуть використовуватись в програмі.

Program - це заголовок програми, що вказує її ім'я. Для Turbo Pascal 6.0 і більш новіших версій цей підрозділ вказувати необов'язково, хоча рекомендовано вказувати заголовок програми, щоб уже при першому знайомстві з її текстом можна було отримати інформацію про її призначення. Потрібно відмітити, що не слід у заголовку програми намагатись вказати всю відому інформацію про програму, адже для цієї мети використовуються коментарі (нагадаю, що коментар вказується у фігурних дужках). Найчастіше в заголовку програми вказують назву програми та її версію.

Потрібно відмітити, що заголовок програми, що слідує, за словом PROGRAM є ідентифікатором і володіє всіма його властивостями. Наприклад, всередині тіла програми не можуть бути оголошені елементи (змінні, константи, і т.д.), що співпадають із заголовком програми. Крім цього, заголовок програми обов'язково повинен починатись з латинської літери, а далі можуть знаходитись символи, допустимі для алфавіту Turbo Pascal (див. розділ 2 даного посібника).

**{$...}** - це підрозділ опису глобальних директив компілятора (нагадаю, що цей підрозділ відрізняється від коментарів тим, що відразу за відкритою фігурною дужкою слідує знак „$"). В цьому розділі програми можна вказати для компілятора певний режим роботи при трансляції самої програми. Такі вказівки можуть містити „замовлення" на включення в текст програми фрагментів інших програм, інформацію відлагоджувача або відомості про необхідність використання арифметичного співпроцесора. Якщо спеціальних директив для компілятора не потрібно використовувати, то цей підрозділ у програмі не вказується.

Uses - цей підрозділ вказує назви модулів та бібліотек, що потрібно підключити до програми. Поняття „модуль", „бібліотека", „блок" складають основу термінології програмування на Pascal. Модуль представляє собою замкнутий блок, що має своє ім'я, компілюється окремо і підключається до вашої програми, як ніби „чорна скринька" із набором певних (описаних у ньому) процедур, функцій, типів даних, констант і т.д. Бібліотека представляє собою набір таких модулів. Якщо модулі в програмі не потрібно використовувати (не використовується процедури, функції і т.д., що описані в модулі), то цей підрозділ не вказуємо.

Потрібно відмітити, що оператор USES може використовуватись в програмі лише один раз, при цьому в нього є чітко визначене місце (він знаходиться поперед усіх операторів та підрозділів (крім заголовка програми та директив компілятора)).

Label - підрозділ, в якому вказується список усіх міток, що використовуються в програмі. Якщо міток у програмі немає, то підрозділ LABEL не вказуємо.

Вважається неофіційним правилом: не використовувати в Pascal-програмі міток, оскільки це суперечить принципам програмування в Pascal. Якщо в програмі використано мітки, то така програма вважається написаною дуже безграмотно.

Const - підрозділ опису констант. Якщо в програмі будуть застосовуватись константи, то їх імена та значення (після знаку рівності) вводять в розділі **Const.** В інших випадках цей підрозділ не вказується. Слід відмітити, що стандартні константи, прийняті в Pascal, не потрібно описувати в даному розділі.

Type - підрозділ опису користувацьких типів даних. В цьому підрозділі переважно вказують складні та нестандартні типи даних. Якщо такі типи в програмі не використовують, то даний підрозділ не вказується.

Var - підрозділ опису всіх змінних, що використовуються в програмі. Потрібно відмітити, що всі змінні, що використовуються в програмі, обов'язково повинні бути описані в підрозділі **var**. Це єдиний підрозділ розділу описів та узгоджень, який обов'язково повинен бути представлений у кожній Pascal-програмі.

Розділ текстів процедур та функцій вказується лише у випадку, якщо в програмі використовуються нестандартні процедури та функції, що створені самим користувачем (якщо вони не описані в одному з модулів вказаних у розділі USES). Потрібно відмітити, що процедури та функції - це спеціальним чином оформлені послідовності команд (у вигляді підпрограми). Доступ до цієї підпрограми може здійснюватись з будь-якого місця основної програми, а також з будь-якої процедури та функції, що описані нижче по тексту програми. Більш детально про використання процедур та функцій буде описано далі.

Третій розділ програми - це розділ основного блоку програми. В цьому розділі знаходиться основний текст програми. Починається цей розділ словом **begin** і закінчується словом **end**, після якого слідує крапка.

Для того, щоб ви навчились правильно створювати програми на Pascal, слід засвоїти декілька важливих правил написання програм.

1. Основний текст будь-якої програми починається службовим словом **begin** і закінчується словом **end**, після чого слідує крапка. Без крапки програма вважається не закінченою. І навпаки, якщо в програмі знаходиться крапка, то всі команди, що слідують за нею ігноруються (оскільки програма завершена).
2. В кінці кожної команди ставиться крапка з комою („;“) - символ, що розділяє команди між собою.
3. Після команди, яка знаходиться перед **end**, крапку з комою („;“) бажано не встановлювати, оскільки буде вважатись, що перед end є ще один порожній оператор.
4. Команди в Pascal можна записувати в один рядок, хоча для полегшення читабельності програми бажано кожну команду вказувати з нового рядка.

5. Якщо користувач бажає помістити в текст програми коментарі, то їх необхідно вказувати у фігурних дужках (замість фігурних дужок можна вказувати альтернативний набір символів - „**(*** ,,, ***)**“).
6. При вказанні в програмі виразів, що містять будь-які дужки потрібно пам'ятати, що кількість закритих та відкритих дужок повинна бути однаковою.
7. Якщо потрібно використати декілька операторів у якості одного складеного оператора, то їх слід взяти в операторні дужки, що починаються словом **begin** і закінчуються **end**. При цьому кількість слів **begin** у програмі повинна співпадати з кількістю слів **end**.
8. Усі змінні, константи та типи даних, що використовуються в програмі, повинні бути описані в розділах **const**, **type** та **var**.

Необов'язково (або "Правила хорошого тону")

1. Після команди **begin** всі наступні команди, аж до відповідного йому **end**, бажано записувати з відступом (наприклад, на величину слова **begin**).
2. Назви всіх команд бажано записувати малими літерами, а змінні - великими.

Розділ 4. Прості типи даних у Turbo Pascal

4.1. Цілі числа

Як відомо з математики числа бувають дійсні та цілі, тому в Turbo Pascal числові типи даних мають такий ж поділ. Хоча в Pascal цілі числа бувають п'яти підтипів.

Такий поділ цілих чисел на типи пов'язаний з різним діапазоном допустимих значень, а отже, з різним розміром числа в пам'яті. Чим більше число, тим більше пам`яті потрібно для його запам'ятовування, а в Pascal для даних кожного однакового типу відводиться однаковий об'єм оперативної пам'яті. Тому, якщо програміст знає, що в програмі будуть використовуватись невеликі числа (наприклад, до 200), то йому не потрібно відводити для змінної стільки ж пам'яті, як для чисел кратних сотням тисяч.

В таблиці 4.1 наведено назви всіх типів цілих чисел із вказанням діапазону їх допустимих значень та розміру, що вони займають у пам'яті.

Таблиця 4.1.

Типи цілих чисел у Turbo Pascal

Тип	**Діапазон допустимих значень**	**Об'єм у пам'яті (байт)**
ShortInt	-128 ... 127	1
Byte	0 ... 255	1
Integer	-32768 ... 32767	2
Word	0 ... 65535	2
LongInt	-2147483648 ... 2147483647 (-2^{31} ... $2^{31}-1$)	4

В тексті програми цілі числа записуються в звичній для будь-якого користувача формі, наприклад, 31784 (допустимо +31784) або -12345.

Потрібно відмітити, що запис цілого числа в тексті програми ні в якому якому разі не повинен містити десяткової крапки, адже це число буде вже сприйматись як дійсне і в програмі буде виведено помилку несумісності типів.

Змінні цілого типу в Pascal-програмі обов'язково потрібно описати в розділі **var**. Для цього записуємо команду наступного формату:

var

змінна1,змінна2,... : цілий тип;

Наприклад:

```
var
      X,Y : integer;
```

Цілі числа належать до впорядкованих типів даних, тобто кожне число має свій порядковий номер, причому цей номер відповідає самому числу.

Над даними цілого типу в програмі можна виконувати операції, представлені в таблиці 4.2.

Потрібно відмітити, що при виконанні звичайної операції ділення отримуємо дійсний результат, тому для отримання цілого результату, над цілими числами можна виконувати дві операції цілочисельного ділення:

1. Ділення з визначенням цілої частини (div), наприклад, 7 div 2 = 3
2. Ділення з визначенням дробової частини (mod). При виконанні цієї операції

перше число націло ділиться на друге і визначається остача. Наприклад, 7 mod 2 = 1

Для даних цілого типу можна використовувати дві стандартних константи (нагадаємо, що їх не потрібно описувати в розділі констант):

- **MaxInt** = 32767 - найбільше ціле число типу Integer;
- **MaxLongInt** = 2 147 483 647 = 2^{31}-1 - найбільше ціле число типу LongInt.

Таблиця 4.2

Операції над цілими числами

Операція	**Позначення**	**Приклад**
Додавання	+	X + Y
Віднімання	-	X - Y
Множення	*	X * Y
Цілочисельне ділення з визначенням цілої частини	div	X div Y
Цілочисельне ділення з визначенням дробової частини	mod	X mod Y
Рівність	=	X=Y
Не рівність	<>	X<>Y
Менше (порівняння)	<	X<Y
Менше або рівне (порівняння)	<=	X<=Y
Більше (порівняння)	>	X>Y
Більше або рівне (порівняння)	>=	X>=Y

Процедури й функції, що використовуються для цілих чисел

abs (X) - функція, що знаходить модуль числа X;

sqr (X) - функція, що знаходить квадрат числа X;

inc (X,Y) - процедура, що збільшує значення змінної X на величину Y. Якщо Y не вказувати, то значення X збільшується на „1“ ;

dec (X,Y) - процедура, що зменшує значення змінної X на величину Y. Якщо Y не вказувати, то значення X зменшується на „1“

succ(X) - функція, що знаходить елемент з наступним порядковим номером. Оскільки, як повідомлялось вище, цілі числа мають порядкові номери такі ж як значення чисел, то фактично отримуємо збільшення числа на одиницю;

pred(X) - функція, що знаходить число з попереднім порядковим номером (фактично число на одиницю менше);

odd(X) - функція, що визначає непарність числа X. Дана функція дає логічний результат **true**, якщо число непарне і **false** - парне (більш детально про логічний тип даних буде описано нижче).

Логічні (булеві) операції над цілими числами

Над цілими числами можна виконувати не лише вищеописані математичні операції, але й логічні. При цьому числа переводяться комп'ютером у двійкову систему числення (приклад двійкового представлення десяткових чисел від 0 до 15 показано в таблиці 4.3)

і над двійковими цифрами виконуються логічні операції. Тому логічні операції над цілими числами ще називають двійковими або булевими операціями.

Таблиця 4.3.

Приклади двійкового представлення десяткових чисел

0 - 0000	4 - 0100	8 - 1000	12 - 1100
1 - 0001	5 - 0101	9 - 1001	13 - 1101
2 - 0010	6 - 0110	10 - 1010	14 - 1110
3 - 0011	7 - 0111	11 - 1011	15 - 1111

Потрібно відмітити, що над цілими числами можна виконати шість булевих операцій:

1. Операція логічного **or** („АБО"). При виконанні цієї операції попарно порівнюються двійкові цифри в числах, при чому при співпаданні двох „нулів" (0) отримується результат „нуль" (0). У всіх інших випадках отримується „одиниця" (1).
   Приклади:

   7 **or** 2=7
   ```
   0111
   or
   0010
   0111
   ```
   5 **or** 8=13
   ```
   0101
   or
   1000
   1101
   ```
2. Операція логічного **and** („І"). При виконанні цієї операції попарно порівнюються двійкові цифри в числах, при чому при співпаданні двох „одиниць" (1) отримується результат „один" (1). У всіх інших випадках отримується „нуль" (0).
   Приклади:

   7 **and** 2=2
   ```
   0111
   and
   0010
   0010
   ```
   5 **and** 8=0
   ```
   0101
   and
   1000
   0000
   ```
3. Операція логічного **xor** (заперечення „АБО"). При виконанні цієї операції попарно порівнюються двійкові цифри в числах, при чому при співпаданні двох „одиниць" (1) або двох „нулів" (0) отримується результат „нуль" (0). У випадку, якщо двійкові цифри не співпадають („один" xor „нуль", або „нуль" xor „один"), то отримується результат „один" (1).
   Приклади:

   7 **xor** 2=5
   ```
   0111
   xor
   0010
   0101
   ```
   5 **xor** 8=15
   ```
   0101
   xor
   1000
   1111
   ```
4. Операція логічного **not** („НІ"). При виконанні цієї операції всі двійкові цифри в числі міняються на протилежні, тобто „один" (1) на „нуль" (0), а „нуль" (0) на „один" (1).

Приклади:

not 2=13 **not** 8=7

not	not
0010	1000
1101	0111

5. Операція логічного **shl** (побітовий зсув вліво). При виконанні цієї операції число, що стоїть зліва від shl, переводиться у двійкову систему числення, а тоді з правого боку до нього дописуються „нулі“, кількість яких відповідає числу справа від shl.
   Приклади:
   7 **shl** 2 = 28 1 **shl** 3= 8

   0111 shl 2 = 011100 → 28 0001 shl 3 = 0001000 → 8
6. Операція логічного **shr** (побітовий зсув вправо). При виконанні цієї операції число, що стоїть зліва від shr переводиться у двійкову систему числення, а тоді з лівого боку до нього дописуються „нулі“, кількість яких відповідає числу справа від shr. При цьому з правого боку двійкового числа така ж кількість цифр побітового знищується.
   Приклади:
   7 **shr** 2 = 1 12 **shr** 1= 6

   0111 shr 2 = 0001 → 1 1100 shr 1 = 0110 → 6

4.2. Дійсні числа

Як вже повідомлялось вище, спроба ввести в вирази з цілим типом даних числові значення, що містять крапку, приводить до повідомлень про помилку, некоректної роботи програми або аварійного її завершення. Тому для таких виразів у Pascal використовується дійсний тип даних.

Дійсні числа характеризуються присутністю в їх записі цілої та дробової частини, які в Pascal розділяє символ „.“ (крапка). Дійсні числа, на відміну від цілих, належать до невпорядкованого типу даних. Ще одна важлива характеристика дійсних чисел полягає в тому, що вони ніколи не мають точного значення. Будь-яке дійсне число має приблизне значення, що задається з певною точністю обчислення.

Потрібно відмітити, що в Turbo Pascal дійсні числа можуть бути представлені у двох форматах:

1. Дійсні числа з фіксованою крапкою (наприклад, 234.5 або 0.00178)
2. Дійсні числа з плаваючою крапкою (наприклад, 2.345e2 або 1.78e-3)

В Borland Pascal є п’ять стандартних дійсних типів даних (таблиця 4.4), які відрізняються діапазоном допустимих значень та точністю обчислення. Ці параметри повністю залежать від об’єму оперативної пам’яті, що відводиться для зберігання кожного дійсного числа. При розробці програми програміст повинен оцінити допустимі значення,

що можуть отримати відповідні змінні дійсного типу та точність обчислення і вибрати такий тип, щоб він займав найменше пам'яті при забезпечені вищеописаних характеристик.

Таблиця 4.4

Дійсні типи даних у Pascal

Тип	Діапазон	розмір у байтах	кількість значущих цифр
Real	$\pm 2{,}9 \cdot 10^{-39} \ldots \pm 1{,}7 \cdot 10^{38}$	6	11-12
Single	$\pm 1{,}5 \cdot 10^{-45} \ldots \pm 3{,}4 \cdot 10^{38}$	4	7-8
Double	$\pm 5{,}0 \cdot 10^{-324} \ldots \pm 1{,}7 \cdot 10^{308}$	8	15-16
External	$\pm 3{,}4 \cdot 10^{-4932} \ldots \pm 1{,}1 \cdot 10^{4932}$	10	19-20
Comp	$9{,}2 \cdot 10^{-18} \ldots 9{,}2 \cdot 10^{18}$	8	19-20

Змінні дійсного типу в Pascal-програмі обов'язково потрібно описати в розділі **var**. Для цього записуємо команду наступного формату:

var

змінна1,змінна2,... : дійсний тип;

Наприклад:

```
var
      X,Y  : real;
```

Над даними цілого типу в програмі можна виконувати операції, представлені в табл.4.5.

Таблиця 4.5

Операції над дійсними числами

Операція	Позначення	Приклад
Додавання	+	X + Y
Віднімання	-	X - Y
Множення	*	X * Y
Ділення	/	X / Y
Рівність	=	X=Y
Не рівність	<>	X<>Y
Менше (порівняння)	<	X<Y
Менше або рівне (порівняння)	<=	X<=Y
Більше (порівняння)	>	X>Y
Більше або рівне (порівняння)	>=	X>=Y

Для даних дійсного типу можна використовувати стандартну константу (нагадаю, що стандартні константи не потрібно описувати в розділі констант):

Pi - 3,14159265358979323846264 3

Процедури та функції, що використовуються для роботи з дійсними числами

abs (X) - функція, що знаходить модуль числа;

sqr (X) - функція піднесення X до квадрату;

sqrt (X) - функція, що знаходить корінь квадратний від X;

ln (X) - функція, що знаходить логарифм натуральний від X;

exp (X) - функція, що знаходить експоненту від числа X;

sin (X) - функція, що знаходить синус тригонометричний від X;

cos (X) - функція, що знаходить косинус тригонометричний від X;

arctan (X) - функція, що знаходить арктангенс тригонометричний від X;

trunc (X) - функція, що знаходить цілу частину числа X, при цьому отримуємо результат цілого типу;

round(X) - функція, що знаходить дробову частину числа X, при цьому отримуємо результат цілого типу;

int(X) - функція, що знаходить цілу частину числа X, при цьому отримуємо результат дійсного типу. При цьому відкидається дробова частина числа і вибирається найближче менше, що відповідає цілому числу;

frac(X) - функція, що знаходить дробову частину числа X, при цьому отримуємо результат дійсного типу;

random(X) - функція, що вибирає випадкове число в діапазоні від 0 до X. Якщо дану функцію вказати без аргументу (просто **random**), то випадкове число вибирається з діапазону 0...1.

Потрібно зауважити, що тригонометричні функції, що використовуються в Pascal-програмах знаходять значення кутів, представлених у радіанах. Тому часто доводиться використовувати перетворення:

X=X*Pi/180 - перетворення градусів у радіани;

X=X*180/Pi - перетворення радіан у градуси.

Для того, щоб знайти значення інших математичних та тригонометричних функцій у Pascal-програмах, потрібно використовувати формули перетворення. Деякі з найчастіше використовуваних виразів представлені нижче в посібнику.

Для піднесення X у степінь Y потрібно встановити знак X і у випадку від'ємного значення X парність (непарність) Y:

X>0	$X^Y = \exp(Y*\ln(X))$
X=0	$X^Y = 1$
X<0, Y - парне	$X^Y = \exp(Y*\ln(\text{abs}(X)))$
X<0, Y - не парне	$X^Y = -\exp(Y*\ln(\text{abs}(X)))$

Аналогічно для знаходження кореня X степені Y:

X>0	$\sqrt[Y]{X} = \exp((1/Y)*\ln(X))$
X=0	$\sqrt[Y]{X} = 1$
X<0, Y - непарне	$\sqrt[Y]{X} = -\exp((1/Y)*\ln(\text{abs}(X)))$
X<0, Y - парне	Не існує

В таблиці 4.6 представлено інші тригонометричні та математичні функції, що обчислюються через додаткові вирази.

Таблиця 4.6.

Вирази для обчислення деяких математичних та тригонометричних функцій

Функція	**Вираз**
Логарифм звичайний	$Log_Y X$ = ln(X)/ln(Y)
Логарифм десятковий	LgX = ln(X)/ln(10)
Тангенс	TgX = sin(X)/cos(X)
Котангенс	CtgX = cos(X)/sin(X)
Арксинус	arcsinX = arctan(X/sqrt(1 - sqr(X)))
Арккосинус	arccosX = arctan((1 - sqr(X))/X)
Арккотангенс	ArcctgX = 1/arctan(X)

4.3. Символьний тип

Символьний тип призначений для зберігання лише одного символу, що належить стандартній ASCII-таблиці символів. У змінну цього типу може бути поміщений будь-який із 256 символів основної або розширеної ASCII-таблиці. Якщо в програмі використовуються не змінні, а самі символи як константи, то ці символи беруться в одинарні лапки.

Цей тип даних впорядкований, кожний елемент якого має порядковий номер, що відповідає порядковому номеру символу в **ASCII** таблиці (від 0 до 255). Нагадаємо, що коди символів ASCII можна поділити на декілька діапазонів:

- від 0 до 31 - керуючі символи (наприклад, 1 - переведення курсору в новий рядок, 7 - звуковий сигнал „beep", 8 - „Backspace", 13 - „Enter", 27 - „ESC" і т.д.);
- від 32 до 47 - спеціальні символи (32 - пропуск, 33 - !, 34 - „, 35 - #, 36 - $, 37 - %, 38 - &, 39 - ', 40 - (, 41 -), 42 - *, 43 - +, 44 - кома, 45 - -, 46 - крапка, 47 - /);
- від 48 до 57 - цифри 0 ... 9;
- від 58 до 64 - спеціальні символи (58 - :, 59 - ;, 60 - <, 61 - =, 62 - >, 63 - ?, 64 - @);
- від 65 до 90 - великі латинські літери (A, B... Z);
- від 91 до 96 - спеціальні символи (91 - [, 92 - \, 93 -], 94 - ^, 95 - _, 96 - `);
- від 97 до 122 - малі латинські літери (a, b ... z);
- від 123 до 127 - спеціальні символи (123 - {, 124 - |, 125 - }, 126 - ~, 127 - Ctrl+Backspace);
- від 128 до 255 - великі та малі грецькі літери та символи псевдографіки. Потрібно відмітити, що при підключенні драйверу кирилиці, деякі стандартні символи в цьому діапазоні таблиці заміняються на великі та малі літери кирилиці.

В Pascal символьний тип даних позначається **char** і описується в розділі var. Для цього необхідно вказати таку конструкцію:

var

змінна1,змінна2,... : char;

Наприклад:

```
var
      X,Y  : char;
```

Змінні символьного типу займають у пам'яті по одному байту (так само, як дані цілого типу byte).

До даних символьного типу можна використовувати такі стандартні функції:

chr(X) - вивести символ по його номеру в ASCII-таблиці (**X** - ціле число, що вказує номер символу). Наприклад, chr(33) = '!'. Замість цієї функції можна використовувати послідовність **#число**, де число - номер символу. Наприклад, #33= '**!**';

ord(X) - вивести код символу (**X** - дані типу char). Наприклад, ord('A') = 65;

succ(X) - функція, що знаходить елемент з наступним порядковим номером (**X** - ціле число, що вказує номер символу). Наприклад, succ(65) = 'B';

pred(X) - функція, що знаходить елемент з попереднім порядковим номером. Наприклад, pred(66) = 'A'.

В якості значень змінних типу char можна також вказувати спеціальні символи, які часто називають „керуючими кодами". Їх вказують з допомогою значка „^" і наступної літери. Керуючим кодам відповідають конкретні значення ASCII-коду, що дозволяє замість „^a" вводити „#1", так як код Ctrl+A дорівнює 1. Керуючі коди часто використовують у процедурах виведення інформації для форматованого виведення (переведення курсору на новий рядок). Крім цього, програміст може сам явно включати керуючі коди в потік інформації, що виводиться, керуючи її розміщенням на екрані, а також створювати „звукове супроводження".

4.4. Логічний тип

Логічний тип даних - це впорядкований тип даних, який може приймати одне з двох значень: **TRUE** (істина) - правдиве твердження або **FALSE** (не правда) - хибне значення. При цьому FALSE має порядковий номер „нуль", а TRUE - „один".

В Turbo Pascal логічні дані бувають чотирьох типів: **boolean** (1 байт в пам'яті), **ByteBool** (1 байт у пам'яті), **WordBool** (2 байти в пам'яті) та **LongBool** (4 байти в пам'яті).

Змінні типів ByteBool, WordBool, LongBool були введені в Turbo Pascal версії 7.0 лише для того , щоб забезпечити його сумісність з іншими мовами програмування й середовищем Windows. Відмінність між цими трьома типами та типом Boolean полягає в тому, що значення ідентифікатора константи TRUE для boolean завжди відповідає числу 1, тоді як для решти типів, це значення може бути іншим, але не рівним 0 (для всіх логічних типів значення FALSE завжди 0).

В Pascal-програмі всі змінні логічного типу потрібно описувати в розділі var:

var

змінна1,змінна2,... : логічний тип*;*

Наприклад:

```
var
      X,Y  : boolean;
```

Для даних логічного типу можна використовувати такі стандартні функції:

succ(X) - функція, що знаходить елемент з наступним порядковим номером, тобто succ(FALSE)=TRUE, але succ(TRUE) не існує (буде виводитись помилка);

pred(X) - функція, що знаходить елемент з попереднім порядковим номером, тобто pred(TRUE)= FALSE, але pred(FALSE) не існує.

Над даними логічного типу можна виконувати наступні операції:

= - рівність двох змінних;

<> - нерівність двох змінних;

or - логічне „АБО“ (наприклад, X or Y). Порівнюються значення двох логічних змінних, при цьому, якщо обидва FALSE, то результат FALSE (FALSE or FALSE = FALSE), у всіх інших випадках результат TRUE;

and - логічне „І“ (наприклад, X and Y). Порівнюються значення двох логічних змінних, при цьому, якщо обидва TRUE, то результат TRUE (TRUE and TRUE = TRUE), у всіх інших випадках результат FALSE;

xor - логічне заперечення „АБО“ (наприклад, X xor Y). Порівнюються значення двох логічних змінних, при цьому, якщо обидва TRUE або FALSE, то результат FALSE (TRUExorTRUE=FALSE або FALSExorFALSE=FALSE), у всіх інших випадках результат TRUE (TRUExorFALSE=TRUE або FALSExorTRUE=TRUE);

not - логічне заперечення (наприклад, not X). Логічне значення змінної змінюється на протилежне (not TRUE = FALSE або not FALSE = TRUE).

4.5. Інтервальний тип даних

При описуванні змінних у програмі, як правило, відомо, що вони будуть використовуватись для представлення підмножини значень деякого типу. Ця підмножина значень на мові Pascal може бути представлена з допомогою так званого інтервального типу даних.

Інтервальний тип - це впорядкований тип даних, елементи якого належать до певної множини (діапазону даних) простого типу. При заданні діапазону потрібно вказувати найменше та найбільше його значення, розділивши їх двома крапками (..). Елементами інтервального типу можуть бути:

- цілі числа (наприклад, 1..1000);
- дані символьного типу (наприклад, 'А'..'Я'');
- дані логічного типу (наприклад, FALSE..TRUE).

Дані інтервального типу потрібно описувати в Pascal-програмі в розділі type.

type

ім'я = поч.зн .. кін зн;

var

змінна1, змінна3,... : ім'я*;*

Наприклад:

```
type
       Dni = 1 .. 31;
       LatLit = 'A' .. 'Z'';
       Integ = 0 .. maxint;
var
       X,Y  : Dni;
       B,Z  : LatLit;
       C,A  : Integ;
```

Потрібно відмітити, що у Pascal-програмах допускається описувати дані інтервального типу безпосередньо в розділі var. Наприклад:

```
var
        X,Y  : 0 .. 356;
        Z,S  : 'а' .. 'я';
```

Для даних інтервального типу можна використовувати наступні функції:

succ(X) - функція, що знаходить елемент з наступним порядковим номером;

pred(X) - функція, що знаходить елемент з попереднім порядковим номером.

4.6. Перераховний тип даних

Перераховний тип визначається як впорядкований набір ідентифікаторів, що заданий шляхом їх перераховування. При цьому список ідентифікаторів, розділених комами, вказується в круглих дужках при описі типу в розділі type:

type

ім'я = (*знач1, знач2, знач3, ...)*;

var

змінна1, змінна2,... : ім'я*;*

Наприклад:

```
type
       Day = (Monday, Tuersday, Wednesday, Thursday, Friday, Saturday, Sunday);
       Int = (0, 2, 4, 6, 8, 10, 12);
var
       X,Y  : Day;
       B,Z  : Int;
```

Особливістю перераховного типу є те, що його елементи не можна вводити та виводити стандартними командами. Тому для введення або виведення даних перераховного типу потрібно використовувати проміжні змінні інших типів, наприклад, символьного чи стрічкового, а тоді з допомогою оператора перевірки умови вибирати відповідне значення перераховного типу.

Для даних перераховного типу можна використовувати лише операції порівняння: „=“ - рівність, „<>“ - нерівність, „<“ - менше, „>“ - більше (наприклад, X>Friday - будь-який день після п'ятниці (субота або неділя)).

Крім цього, для даних перераховного типу можна використовувати функції:

succ(X) - функція, що знаходить елемент з наступним порядковим номером;

pred(X) - функція, що знаходить елемент з попереднім порядковим номером.

Наприклад*:*

```
       succ (Friday) = Saturday.
```

Розділ 5. Оператори мови Turbo Pascal.

5.1. Введення та виведення інформації в Pascal-програмах

В Pascal немає операторів введення та виведення інформації. Ці операції в програмах виконуються через стандартні процедури.

Для виведення інформації можна використовувати процедури:

write(*вираз1*[,*вираз2*, ...]) - виводить значення виразів, після чого курсор залишається в тому ж самому рядку;

writeln(*вираз1*[,*вираз 2*, ...]) - виводить значення виразів, після чого курсор переводиться в наступний рядок.

Якщо в операторі виведення використовуються текстові константи, то їх записують в одинарних лапках.

В операторі виведення прийнято поєднувати текстові константи, що вказують назву змінної, що виводиться, та саму змінну (значення якої потрібно вивести). Наприклад:

```
writeln('X=',X);
writeln('Значення F=',F);.
```

Для введення інформації можна використовувати процедури:

read(*змінна1[, змінна2, ...]*) - вводить значення змінної, після чого курсор залишається в тому ж самому рядку;

readln(*змінна1[, змінна2, ...]*) - вводить значення змінної, після чого курсор переводиться в наступний рядок.

Потрібно відмітити, що в одній команді введення в Pascal прийнято вводити лише одну змінну, хоча (як показано вище) допускається вказання декількох змінних.

Якщо під час виконання програми в ній зустрічається процедура read або readln, то відбувається зупинка програми й користувачу потрібно в позицію курсору ввести значення змінної та натиснути Enter. Після чого введене значення присвоюється відповідній змінній. Основний недолік процедури read (readln) полягає в тому, що при введенні інформації користувач не бачить, значення якої змінної він вводить. Для усунення цього недоліку при введені інформації використовується одночасне поєднання процедур введення та виведення інформації.

Наприклад:

```
write('A='); readln(A);
write('X='); readln(X);
```

Форматоване виведення інформації

Потрібно відмітити, що процедура writeln (write) виводить значення змінних дійсного типу в експотенціальному (науковому) форматі: „**числоЕстепінь**“. Наприклад X=1.978500000E+00, що відповідає числу 1,9785.

Таке представлення чисел часто є незручним для сприйняття, тому в Pascal використовують форматований вивід інформації. Для цього записують такий формат:

```
writeln(вираз:n:m);
```

де **n** - кількість цифр, що відводиться для представлення числа;

m - кількість цифр, що відводиться для представлення дробової частини числа.

Наприклад

```
Writeln('X=',X:7:2);
```

В даному прикладі на число відводиться 7 позицій, з яких 2 - дробова частина числа, а 4 - ціла (7 позицій усього, 2 - дробова частина і 1 - крапка 7-2-1=4). При цьому необхідно пам'ятати, що Pascal не виводить „початкових нулів". Наприклад, для виведення числа 5 відведено 4 позиції, то на екран буде виведено 5, а не 0005.

З описаного вище можна зробити висновок, що при вказанні n та m для форматованого виведення інформації потрібно їх значення задавати таким, щоб різниця між ними була не менша 2 (n-m>=2).

Якщо при виведенні значення деякої змінної з допомогою writeln (write) число, що виводиться, не буде поміщатись у вказаний формат (тобто кількість дійсних розрядів числа буде більше за відведену кількість), то, всупереч указаному формату, частина значення змінної, що розміщена перед десятковою крапкою, буде повністю виводитись на екран. При цьому кількість позицій справа від десяткової крапки не змінюється. Якщо для значення, що виводиться, дробова частина не поміщається у вказану кількість позицій, то число заокруглюється.

5.2. Оператор присвоєння в Turbo Pascal

Оператор присвоєння - це один із найчастіше використовуваних операторів у Pascal-програмах. Він використовується для знаходження значення виразів, присвоєння змінним потрібних значень і ін.

Формат оператора присвоєння:

Змінна := вираз;

Наприклад

```
X := 5;
Y := sin(X)/ln(X) + exp(X);
```

Не потрібно плутати оператор присвоєння (:=) з операцією порівняння (=), що використовується при перевірці умови, а також при описі констант (розділ const) та типів даних (розділ type).

Ознайомившись з оператором присвоєння, а також процедурами введення та виведення інформації ви вже можете самостійно створити найпростішу програму на мові Pascal для обчислення значення виразів. Для цього спочатку рекомендую вам повторити загальну структуру Pascal-програми (див. розділ 3) та процедури, функції, що використовуються для цілого та дійсного типів даних (див. розділ 4.1. та 4.2).

Приклад 1

Скласти програму на мові Turbo Pascal для обчислення значення виразу $Y=A\cdot X^5 + B\cdot\sin(\alpha) - C\cdot X^2+X$. Будемо вважати, що X має додатне значення.

```
var
       X,A,B,C,Alpha,Y : real;
begin
       write('A=');  readln(A);
       write('B=');  readln(B);
```

```
        write('C=');  readln(C);
        write('X=');  readln(X);
        write('Alpha=');  readln(Alpha); {Кут вводимо в градусах}
        Y := A*exp(5*ln(X))+B*sin(Alpha*Pi/180)-C*sqr(X)+X;
        writeln('Y=',Y:7:3);
        readln {пауза в програмі}
end.
```

В кінці програми використано процедуру введення інформації без параметрів для того, щоб зробити в програмі паузу й користувач устиг прочитати виведені результати, оскільки оператор writeln паузи не робить.

Приклад 2

Скласти програму на мові Turbo Pascal для обчислення значення виразу

$$Y = \frac{\ln|X| + \sqrt{tgX} + \cos(X) - X^2}{\lg^3\left|\sqrt{tg(\cos(\sqrt{X}))}\right|}$$

```
var
        X,Y,Y1,Y2  : real;
begin
        write('X=');     readln(X);
        Y1 :=ln(abs(X))+sqrt(sin(X)/cos(X))+cos(X)-sqr(X);
        Y2 := ln(abs(sqrt(sin(cos(sqrt(X)))/cos(cos(sqrt(X))))))/ln(10);
        Y := Y1/(Y2*Y2*Y2);
        writeln('Y=', Y:7:4);
        readln
end.
```

В даній програмі для спрощення виразу в операторі присвоєння формулу розбито на дві частини (чисельник та знаменник). Тоді, з допомогою проміжних змінних Y1 та Y2 обчислено спочатку чисельник та частину знаменника. Знайдені проміжні результати підставлено в основну формулу.

Приклад 3

Скласти програму на мові Turbo Pascal для обчислення значення виразу

$$F = \frac{\sqrt[8]{\left|3 \cdot ctg|x| + Y^3 - B^Y\right| + X^2}}{B \cdot \log_B Y + Y}, \text{ де } Y=\arcsin(B \cdot X)$$

```
var
        X,B,Y,F,F1 :  real;
begin
        write('B=');  readln(B);
        write('X=');  readln(X);
        Y := arctan((X*B)/sqrt(1 - sqr(B*X)));
        F1 := exp((1/8)*ln(abs(3*cos(abs(X))/sin(abs(X))+Y*Y*Y-

                exp(Y*ln(B)))+sqr(X)));
        F  := F1/(B*ln(Y)/ln(B)+Y);
        writeln('F=',F:9:5);
        readln {пауза в програмі}
end.
```

5.3. Оператор безумовного переходу

Оператор безумовного переходу (**goto**) являє собою простий оператор, використовуючи який можна змінювати порядок виконання операторів у програмі. Загальний вигляд оператора безумовного переходу:

goto мітка;

При виконанні даного оператора відбувається перехід у те місце програми, де знаходиться вказана мітка. Нагадаємо, що всі мітки, що використовуються в програмі, потрібно описувати в розділі **Label**. В якості мітки можна використовувати будь-який ідентифікатор, що складається з латинський літер та цифр. Допускається використання цілих чисел від 0 до 9999.

Після вказання мітки в місці, куди потрібно здійснити перехід, потрібно вказати символ „двокрапка" (:). Наприклад A1:.

Нагадаємо також, що використання операторів безумовного переходу в Pascal-програмах є небажаним, адже це є ознакою безграмотності самого програміста. Вважається, що будь-яку програму в Pascal можна написати, обійшовши використання міток, тому оператору безумовного переходу в даному посібнику приділено дуже мало уваги.

5.4. Оператор перевірки умови (розгалуження)

Часто при створені програми необхідно щоб певна її частина виконувалась лише при виконані умови. В таких випадках програмісту необхідно застосувати оператор перевірки умови, який має наступний формат:

if *умова*
then *команда1*
else *команда2;*

Коли в програмі зустрічається така конструкція, то спочатку перевіряється умова. Якщо вона справджується, то виконується команда1, а тоді команда, що йде наступною після оператора if. У випадку, коли умова не справджується, то виконується команда2, а тоді наступна після оператора if команда.

Зверніть увагу, що перед словом else крапка з комою не ставиться.

Оператор перевірки умови може мати коротку форму:

if *умова*
then *команда1;*

В цьому випадку, якщо умова не виконується, то оператор if взагалі пропускається. При цьому відразу виконується наступна команда після оператора перевірки умови.

Потрібно відмітити, що в якості „команди1" або „команди2" може використовуватись лише одна команда (оператор). Якщо потрібно використати більше, то їх необхідно взяти в операторні дужки:

```
begin
        команда1;
        команда2;
        команда3;
        . . .
end;
```

Розглянемо декілька прикладів використання перевірки умови.

Приклад 1

Скласти програму, яка б знаходила корені квадратного рівняння $A \cdot X^2+B \cdot X+C=0$.

```
var
        X1,X2,A,B,C,D  :  real;
begin
        write('A=');   readln(A);
        write('B=');   readln(B);
        write('C=');   readln(C);
        D := sqr(B) - 4*A*C;  {визначення дискримінанту}
        If D<0
                then   writeln('Корені рівняння не існують')
                else
                if D=0
                        begin
                                X1 := -B/(2*A);
                                Writeln('Корінь рівняння  X=',X1:7:3)
                        end
                        else
                        begin
                                X1 := (-B + sqrt(D))/(2*A);
                                X2 := (-B - sqrt(D))/(2*A);
                                Writeln('Корені рівняння  X1=',
                                        X1:7:3,'X2=',X2:7:3)
                        end;
        readln
end.
```

Приклад 2

Скласти програму на мові Turbo Pascal для обчислення значення виразу

$$Y = \begin{cases} \left|\lg\left|X^3\right|\right| + \sqrt[3]{X}, & \text{при} \quad 0 < X < 45 \\ X^4 + \cos(10 \cdot X), & \text{у решті випадків} \end{cases}$$

```
var
        X,Y :  real;
begin
        write('X=');   readln(X);
        if  (X<45) and (X>0)
                then Y := ln(abs(X*X*X))+exp((1/3)*ln(X))
                else Y := sqr(sqr(X))+ cos(10*X*PI/180);
        writeln('Y=',Y:9:5);
        readln
end.
```

В даному випадку використано складну умову, що складається з двох простих (X>0 і X<45), оскільки в Pascal не можна використовувати складні вирази порівняння (типу 0<X<45).

Для об'єднання простих умов у складну використовуються логічні операції, такі як and, or або not. Оскільки, за правилами черговості виконання, ці операції виконуються в першу чергу, тому більш „слабші" операції порівняння слід брати в дужки.

Приклад 3

Дано три символи. Скласти програму, що визначає який із цих символів має найбільший код у ASCII-таблиці.

```
var
     A, B, C :  char;
begin
       write('Введіть перший символ  - ');  readln(A);
       write('Введіть другий символ  - ');   readln(B);
       write('Введіть третій символ  - ');    readln(C);
       if  (ord(A)>=ord(B)) and (ord(A)>=ord(C))
              then  writeln('Найбільший код має символ  - ', A)
              else
                     if  (ord(B)>=ord(A)) and (ord(B)>=ord(C))
                     then  writeln('Найбільший код має символ  - ', B)
                     else   writeln('Найбільший код має символ  - ', C);
       readln
end.
```

5.5. Оператор вибору

В Pascal-програмах досить часто використовується оператор вибору, що заміняє вкладені конструкції оператора перевірки умови. Оператор вибору є ідеальним засобом для обробки ситуацій, коли умова може приймати більше двох значень. Єдиний недолік цього оператора полягає в тому, що в умові можна використовувати лише дані впорядкованих типів (цілого, символьного, логічного, перераховного або інтервального).

Формат оператора вибору має наступний вигляд:

case *змінна* of
значення1 : *команда1*;
значення2 : *команда2*;
значення3 : *команда3*;
. . .
значенняN : *командаN*;
[else *командаX*]
end;

Коли в програмі зустрічається даний оператор, то перевіряється значення змінної (після case), якщо воно рівне „значенню1", то виконується „команда1", а тоді наступна команда після оператора вибору. Коли значення змінної не рівне „значеню1", то перевіряється, чи воно рівне „значенню2". Якщо так, то виконується „команда2", а якщо

ні, то знову перевіряється значення змінної і порівнюється з „значеннямЗ“ і т.д. Коли змінна не рівна ні одному з N значень, то виконується „командаХ“, що слідує після else.

Оператор case може мати скорочену форму, у цьому випадку в ньому відсутня конструкція „else командаХ“.

В якості значень змінної, в операторі case можуть бути не лише окремі значення, а й елементи перераховного та інтервального типів.

Наприклад:

```
case X of
1,3,5   : команда1;
10..17 : команда 2;
end;
```

Потрібно відмітити, що для кожного значення змінної можна використати лише по одній команді. У випадку, якщо в програмі потрібно більше команд, то їх слід взяти в операторні дужки (аналогічно тому, як показано в розділі 5.4.).

Зверніть увагу, що в операторі вибору перед словом else встановлюється крапка з комою (на відміну від аналогічної конструкції в операторі перевірки умови).

Приклад

Скласти програму, яка б по введеному номеру дня тижня виводила його назву.

```
var
        D  :  byte;
begin
        write('D=');   readln(D);
        case D of
        1 : writeln('Понеділок');
        2 : writeln('Вівторок');
        3 : writeln('Середа');
        4 : writeln('Четвер');
        5 : writeln('П''ятниця');
        6 : writeln('Субота');
        7 : writeln('Неділя');
        else  writeln('Не вірно введено номер дня тижня')
        end;
        readln
end.
```

5.6. Оператори циклу

Оператор циклу - це оператор, який повторно дозволяє виконувати один і той же набір команд певну кількість разів. При цьому не має необхідності записувати в тексті програми одинакові оператори декілька разів.

В Turbo Pascal є три типи оператора циклу:

- оператор циклу з наперед заданою кількістю повторень;
- оператор циклу з передумовою;
- оператор циклу з післяумовою.

5.6.1. Оператор циклу з наперед заданою кількістю повторень

Оператор циклу з наперед заданою кількістю повторень використовується в тому випадку, коли нам наперед відомо, скільки разів повинен повторюватись набір команд у циклі. Такий цикл ще називають **циклом з лічильником**.

В цьому операторі обов'язково потрібно вказувати наступні параметри:

- назву змінної впорядкованого типу, в якій зберігається кількість повторень циклу (лічильник циклу);
- початкове значення для змінної циклу (лічильника циклу);
- кінцеве значення для змінної циклу (лічильника циклу).

Оператор циклу з наперед заданою кількістю повторень поділяється на два типи:

- цикл по зростанню;
- цикл по спаданню.

Розглянемо формат запису оператора циклу з наперед заданою кількістю повторень по зростанню:

for *змінна* : = *значення1* to *значення2* do
команда;

У випадку, якщо в програмі зустрічається така конструкція, то змінній присвоюється „значення1" і виконується команда в циклі. Після цього значення змінної (лічильника) збільшується на 1 і перевіряється чи воно не перевищує „значеня2". Якщо ні, то команда виконується повторно і значення змінної (лічильника) знову збільшується на 1. Команда в циклі повторюється до тих пір, поки значення змінної (лічильника) не стане більшим за „значення2". Після чого дія циклу закінчується і виконання програми переходить до наступної команди, що слідує після оператора циклу. Необхідною передумовою для виконання циклу з наперед заданою кількістю повторень по зростанню є виконання умови „значення1"<"значення2".

Оператор циклу з наперед заданою кількістю повторень по спаданню має такий формат:

for *змінна* : = *значення1* downto *значення2* do
команда;

При цьому необхідно, щоб „значення2" було більше за „значення1". Цей цикл виконується аналогічно до циклу по зростанню, але значення лічильника не збільшується на 1, а зменшується. При цьому цикл повторюється до тих пір поки значення лічильника не стане меншим за „значення2".

Потрібно відмітити, що в циклі з наперед заданою кількістю повторень можна використовувати лише по одній команді. Якщо є необхідність помістити в цикл більше команд, то їх слід узяти в операторні дужки (begin ... end).

У випадку, якщо цикл потрібно завершити раніше останньої із команд, що знаходяться в операторних дужках, то можна використати одну з команд:

1. **break** - достроково завершує цикл і переводить виконання програми на наступну команду, що слідує після циклу;
2. **continue** - достроково завершує даний крок циклу і переводить виконання програми на початок циклу для виконання наступного кроку.

Потрібно відмітити, що дані команди з'явились лише в Turbo Pascal версії 7.0.

Приклад 1

Скласти програму для знаходження суми десяти введених з клавіатури чисел.

```
var
       X,S : real;
       I : byte;
begin
       S:=0; {вказуємо початкове значення суми}
       for I:=1 to 10 do
       begin
              write('X='); readln(X);       {вводимо число}
              S:=S+X;         {додаємо число до попередньої суми}
       end;
       writreln('S=',S:7:3);
       readln
end.
```

Приклад 2

Скласти програму для знаходження факторіалу числа N!

При цьому потрібно нагадати, що факторіал числа - це добуток типу 1·2·3·....·N. (Наприклад 5! = 1·2·3·4·5=120)

```
var
       F : LongInt;
       I, N : byte;
begin
       F:=1;   {вказуємо початкове значення добутку}
       write('N=');
       readln(N);
       for I:=2 to N do
       F:=F*I;  {множимо значення факторіала на наступний елемент}
       writreln('F=',F);
       readln
end.
```

5.6.2. Оператор циклу з передумовою

Інший варіант оператора циклу, що використовується в Turbo Pascal - це **оператор циклу з передумовою** (цикл „поки“). Для такого циклу наперед невідомо скільки разів повторяться команди в циклі. Оператор циклу з передумовою має такий формат запису:

while *умова* do
***команда*;**

Якщо в програмі зустрічається така конструкція, то відбувається перевірка умови, якщо умова виконується, то виконується команда в середині циклу (якщо ні, то цикл закінчується і виконується наступна команда після нього). Після того як команда виконалась, відбувається повернення на початок циклу і знову перевіряється умова. Таким чином команда в циклі повторюється до тих пір, поки виконується умова. Як тільки умова перестає виконуватись, то цикл закінчується й виконується наступна команда після циклу.

Потрібно відмітити, що в операторі циклу з передумовою може знаходитись лише одна команда. Якщо потрібно використати більше команд, то їх необхідно взяти в операторні дужки.

Даний цикл можна завершувати достроково з допомогою команд, описаних у попередньому підрозділі.

Приклад

Скласти програму знаходження факторіала числа N, використовуючи оператор циклу з передумовою.

```
var
       F : LongInt;
       I ,N : byte;
begin
       F:=1;  {вказуємо початкове значення добутку}
       I := 1;  {вказуємо початкове значення лічильника}
       write('N=');
       readln(N);
       while  I<=N do
       begin
       F:=F*I;  {множимо значення факторіала на наступний елемент}
       I := I+1;  {збільшуємо значення лічильника на 1}
       end;
       writreln('F=',F);
       readln
end.
```

5.6.3. Оператор циклу з післяумовою

Третій тип оператора циклу, що використовується в Turbo Pascal - це **оператор циклу з післяумовою**, який має наступний формат запису:

```
repeat
        команда1;
        команда2;
          . . .
until умова;
```

Оператор циклу з після умовою обов'язково виконається хоча б раз, адже в ньому спочатку виконуються команди, а вже тоді перевіряється умова (після until). При чому, якщо умова **не виконується**, то відбувається повернення до ідентифікатора **repeat** і команди в циклі повторяються ще раз (нагадаю, що в попередньому циклі, на відміну від даного, для повторного виконання команд, умова повинна виконуватись). Цикл повторюється до тих пір, поки не виконається умова. Як тільки умова виконалась, цикл закінчується й відбувається перехід на оператор, що знаходиться після циклу.

На відміну від попередніх операторів циклу, в цьому операторі можна використовувати необмежену кількість команд.

Даний цикл можна завершувати достроково з допомогою команд, описаних для циклу з наперед заданою кількістю повторень.

Приклад

Скласти програму знаходження факторіала числа N, використовуючи оператор циклу з післяумовою

```
var
        F : LongInt;
        I ,N : byte;
begin
        F:=1;  I := 1;  {вказуємо початкове значення добутку та лічильника}
        write('N=');
        readln(N);
        repeat
        F:=F*I;  {множимо значення факторіала на наступний елемент}
        I := I+1;  {збільшуємо значення лічильника на 1}
        until N<I;
        writreln('F=',F);
        readln
end.
```

Розділ 6. Робота з масивами в Turbo Pascal

6.1. Поняття про масиви. Класифікація масивів

Масив (Array) - це впорядкований набір даних однакового типу. Він належить до складного типу даних. Елементами масиву можуть бути дані будь-якого типу крім файлового.

Кожен елемент масиву характеризується **індексами** або **координатами**, тобто порядковими номерами, під якими вони знаходяться в масиві. Сам масив характеризується **іменем** та **розмірністю**. В якості імені масиву може бути будь-який набір латинських літер та цифр, але першою обов'язково повинна бути літера. Після імені у квадратних дужках записується розмірність масиву. Розмірність - це кількість координат для визначення місця, знаходження кожного елементу в масиві.

За розмірністю масиви поділяються на:

- одномірні масиви - характеризуються однією координатою. Їх ще називають **рядами** або **векторами**. Прикладом одномірного масиву може бути позначення A[5] - вектор „A" з п'яти елементів. Наприклад, [12, 3, 1, 5, 4] - це масив цілих чисел, ['a', '!', '3', 'd', 'x'] - масив символів;
- двомірні масиви - характеризуються двома координатами, номером рядка (перша координата) та номером стовпця (друга координата). Такі масиви ще називають **матрицями** або інколи **таблицями**. Прикладом двомірного масиву може бути позначення A[2,3] - матриця „A" з двох рядків та трьох стовпців. Наприклад

$$\begin{bmatrix} 2 & 3 & 2 \\ 1 & 5 & 3 \end{bmatrix}$$ - матриця цілих чисел $$\begin{bmatrix} 'd' & 'w' & '!' \\ '4' & '+' & '@' \end{bmatrix}$$ - матриця символів

- тримірні масиви - характеризуються трьома координатами - довжина (перша координата), ширина (друга координата) та висота (третя координата). Такі масиви називають **просторовими**. Приклад просторового масиву - A[3, 4, 2];
- чотиримірні масиви і т.д. (у даному курсі розглядатись не будуть).

Доступ до елементів масиву здійснюється через його координати, наприклад запис B[2, 3] := 35 означає, що елемент, який знаходиться в другому рядку та третьому стовпці присвоює значення 35.

6.2. Операції над одномірними масивами (рядами)

У програмі Turbo Pascal масиви спочатку потрібно описати в розділі type. Для цього використовують таку конструкцію:

```
type
    ім'я = array [поч.інд...кін.інд] of тип;
var
   змінна1, змінна2... : ім'я;
```

де ***ім'я*** - назва типу, що отримують масиви;

тип - тип елементів масиву;

поч.інд - номер першого елементу масиву (переважно приймають 1);

кін.інд - номер останнього елементу масиву.

Приклад:

```
type
        mas=array [1..7] of  real; {масив з 7 дійсних чисел}
var
        A,B : mas;
```

Допускається опис рядів в розділі var, наприклад,

```
var
        A, B : array [1..7] of  real;
```

Для введення елементів ряду використовують оператор циклу з наперед заданою кількістю повторень. Для цього записують таку конструкцію:

```
for I:=M to N do
begin
write ('A[', I, ']=');
readln (A[ I ])
end;
```

де M, N - номери першого та останнього елементів ряду;

I - лічильник, що відповідає номеру елемента ряду, що вводиться;

A - назва ряду.

Для виведення елементів ряду також використовується оператор циклу. Для цього записують таку конструкцію:

```
for I := M to N do
writeln('A[', I ,']=', A [ I ]);
```

Розглянемо деякі основні операції над рядами. Для того, щоб знайти суму (різницю) двох рядів A та B, потрібно попарно додати (відняти) всі їх, елементи. Тому потрібно використати оператор циклу:

```
for I := M to N do C[ I ] := A[ I ] + B[ I ];
```

а для різниці

```
for I := M to N do C[ I ] := A[ I ] - B[ I ],
```

Для знаходження добутку двох рядів потрібно попарно перемножити їх елементи:

```
for I := M to N do C[ I ] := A[ I ] * B[ I ],
```

так само, щоб поділити два ряди, потрібно попарно поділити їх елементи.

При множені (ділені) ряду на число перемножуються (діляться) усі елементи ряду на це число:

```
for I := M to N do C[ I ] := K * B[ I ],
```

Якщо потрібно знайти суму елементів в одному ряді, потрібно використати наступну конструкцію:

```
S := 0; {задаємо початкове значення суми}
for I := M to N do S := S+ A[ I ], {до суми додаємо наступний
елемент}
```

Для знаходження добутку елементів використовують таку ж конструкцію, але початкове значення добутку встановлюють не 0 (нуль), а 1 (одиниця), і в циклі замість символу "+" записують „*".

Якщо потрібно знайти максимальний (мінімальний) елемент ряду, то спочатку

присвоюють йому значення початкового елементу, а тоді всі наступні порівнюють з цим максимумом (мінімумом). У випадку, коли поточний елемент більший (менший) за максимум (мінімум), то йому присвоюють значення максимального. Отримуємо такий програмний блок:

```
max := A[1];
for I := 12 to N do
if  max<A[ I ]  then max := A[ I ];
```

Інколи для пошуку найбільшого (найменшого) спочатку йому присвоюють дуже мале (велике) число, а тоді порівнюють кожен елемент ряду з цим значенням, наприклад:

```
max := -1e9;
for I := 1 to N do
if   max<A[ I ]   then max := A[ I ];
```

Розглянемо ще декілька прикладів програм.

Приклад 1

Дано масив A з 15 елементів цілого типу. Порахувати середнє арифметичне додатних чисел.

```
type
       mas = array [1..15] of integer;
var
       A: mas;
       S, N, I: byte;
       X: real;
begin
       for I := 1 to 15 do
       begin
              write (`A[`, I ,`]=`);
              readln (A[ I ])
       end;
       S:= 0; N := 0;
       for I:= 1 to 15 do
       if (A[ I ]  mod  2)=0 {націло ділиться на два, тобто парне число}
              then
              begin
                     S:= S+A[ I ]; {знаходження суми парних чисел}
                     N:= N+1  { знаходження кількості парних чисел}
              end;
       X:= S/N;  {середнє арифметичне – сума поділена на кількість}
       writeln (`X=`,  X:7:3);
       readln
  end.
```

Приклад 2

Дано масив A з 10 елементів цілого типу. Знайти найменше від'ємне число, що не кратне 3.

```
type
       mas = array [1..10] of integer;
var
       A  :  mas;
       I  :  byte;
       MIN  :  integer;
```

```
begin
        for I := 1 to 10 do
        begin
                write (`A[`, I ,`]=`);
                readln (A[ I ])
        end;
        MIN  := maxint; {мінімум присвоїти найбільше можливе ціле число}
        for I  := 1 to 10 do
        if ((A[ I ]  mod  3)<>0)  and (A[ I ]<0) and (MIN<A[ I ])
                then MIN := A[ I ];
        if  min=maxint  {перевіряється чи змінилось значення min}
                then  writeln(`Таких чисел у масиві не існує`)
                else  writeln (`MIN=`,  MIN);
        readln
end.
```

Потрібно відмітити, що дану програму можна було виконувати іншим шляхом. Для встановлення початкового значення мінімального елементу, замість оператора присвоєння MIN := maxint, можна було використати конструкцію:

```
N := 0;   {у розділі var дану змінну слід описати типом byte}
for I := 1 to 10 do
if  ((A[ I ] mod 3)<>0) and (A[ I ]<0)
        then
        begin
                N := I;
                MIN :=A[ I ];
                break {вихід з незакінченого циклу}
        end;
```

Після цього потрібно використати таку ж конструкцію оператора циклу, що показана у вищеописаному прикладі. Але початкове значення лічильника потрібно встановити N (for I := N to 10 do). В кінці програми для виведення результату потрібно перевірити умову „if N=0“ (замість „if min=maxint“).

6.3. Операції над двомірними масивами (матрицями)

Матриці описуються подібно до рядів, для чого потрібно записати таку конструкцію:

type

ім'я = array [*поч.інд1...кін.інд1, поч.інд2...кін.інд2*] of *тип*;

var

змінна1, змінна2... : ***ім'я*;**

де ***ім'я*** - назва типу, що отримують матриці;

тип - тип елементів матриці;

поч.інд 1 - номер першого рядка матриці (переважно приймають 1);

кін.інд1 - номер останнього рядка матриці;

поч.інд2 - номер першого стовпця матриці (переважно приймають 1);

кін.інд2 - номер останнього стовпця матриці;

Приклад*:*

```
type
        matr = array [1..3, 1.. 4]  of  real;
var
        A, B, C : matr;
```

В даному прикладі описано матрицю дійсних чисел з 3 рядків та 4 стовпчиків. Допускається опис матриць в розділі var, наприклад

```
var
        A, B : array [1 .. 2,1 .. 2] of  real;
```

Для введення елементів матриці використовують оператор циклу з наперед заданою кількістю повторень. Для цього записують таку конструкцію:

```
for I := M1 to N1 do
for J := M2 to N2 do
begin
write ('A[', I, ',' , J ,']=');
readln (A[ I , J ])
end;
```

де M1, N1 - номери першого та останнього рядків матриці;

M2, N2 - номери першого та останнього стовпців матриці;

I - лічильник, що відповідає номеру елемента рядку матриці, що вводиться;

J - лічильник, що відповідає номеру елемента рядку матриці, що вводиться;

A - назва матриці.

Для виведення елементів матриці також використовується оператор циклу. Для цього записують таку конструкцію:

```
for I := M1 to N1 do
for I := M2 to N2 do
writeln('A[', I , ',' , J, ']=', A [ I, J ]);
```

Якщо в матриці кількість рядків відповідає кількості стовпців, то матриця називається **квадратною**.

Квадратна матриця називається **симетричною** відносно діагоналі, якщо виконується рівність A[I, J]=A[J, I].

Визначник матриці - це число, яке встановлюється як сума добутків відповідних елементів матриці. Потрібно відмітити, що визначник може існувати лише у квадратній матриці. Для матриці розміром 2X2 визначник знаходиться за формулою:

|A[2,2]| = A[1,1] * A[2,2] - A[1,2] *A[2,1]

Для матриці 3X3 визначник знаходиться за формулою:

|A[3,3]| = A[1,1] * A[2,2] * A[3,3] +A[1,2] * A[2,1] * A[3,1] + A[2,1] * A[3,2] * A[1,3]- A[1,3] * A[2,2] * A[3,1] - A[2,1] * A[1,2] * A[3,3] - A[3,2] * A[2,3] * A[1,1]

Визначники матриць вищих порядків визначаються шляхом розбиття цих матриці на декілька матриць третього порядку і сумування їх визначників. Наприклад, для матриці 4x4 визначник шукається за формулою:

$$\begin{vmatrix} A_{1.1} & A_{1.2} & A_{1.3} & A_{1.4} \\ A_{2.1} & A_{2.2} & A_{2.3} & A_{2.4} \\ A_{3.1} & A_{3.2} & A_{3.3} & A_{3.4} \\ A_{4.1} & A_{4.2} & A_{4.3} & A_{4.4} \end{vmatrix} = A_{1.1} * \begin{vmatrix} A_{2.2} & A_{2.3} & A_{2.4} \\ A_{3.2} & A_{3.3} & A_{3.4} \\ A_{4.2} & A_{4.3} & A_{4.4} \end{vmatrix} - A_{1.2} * \begin{vmatrix} A_{2.1} & A_{2.3} & A_{2.4} \\ A_{3.1} & A_{3.3} & A_{3.4} \\ A_{4.1} & A_{4.3} & A_{4.4} \end{vmatrix} +$$

$$+ A_{1.3} * \begin{vmatrix} A_{2.1} & A_{2.2} & A_{2.4} \\ A_{3.1} & A_{3.2} & A_{3.4} \\ A_{4.1} & A_{4.2} & A_{4.4} \end{vmatrix} - A_{1.4} * \begin{vmatrix} A_{2.1} & A_{2.2} & A_{2.3} \\ A_{3.1} & A_{3.2} & A_{3.3} \\ A_{4.1} & A_{4.2} & A_{4.3} \end{vmatrix}$$

Транспонована матриця A^T - це матриця, в якій рядки та стовпці поміняні місцями. Для виконання такої операції потрібно в тексті програми записати таку конструкцію:

```
for I := M1 to N1 do
for J := M1 to N1 do
AT[ J, I ] := A[ I, J];
```

Дві матриці називаються **рівними**, якщо в них однакова кількість рядків та стовпців і відповідні елементи рівні між собою.

Елементи, в яких номер рядка та номер стовпця співпадають, називаються **головною діагоналлю матриці**. Якщо в матриці всі елементи, крім головної діагоналі рівні 0 (нуль), то матриця називається **діагональною**.

Діагональна матриця, в якій усі елементи головної діагоналі рівні 1, називається **одиничною**.

Якщо при множені матриці „A“ на матрицю „B“ отримується одинична матриця, то матрицю B називають **оберненою до A**. Для того щоб знайти обернену матрицю, потрібно спочатку знайти визначник цієї матриці (якщо визначник рівний 0, то оберненої матриці не існує), а тоді записати в програмі таку конструкцію:

```
for I := M1 to N1 do
for J := M2 to N2 do
B[ I, J]:=A[J, I] / D;
```

де D - визначник матриці „A“.

Розглянемо деякі основні операції над матрицями. Для того, щоб знайти суму (різницю) двох матриць A та B потрібно попарно додати (відняти) всі їх елементи. Тому потрібно використати оператор циклу:

```
for I := M1 to N1 do
for J := M2 to N2 do
C[ I, J] := A[ I, J] + B[ I, J];
```

При множені (ділені) матриці на число потрібно перемножити (поділити) усі елементи матриці на це число:

```
for I := M1 to N1 do
for J := M2 to N2 do
C[ I, J] := K * B[ I, J];
```

Для знаходження добутку двох матриць потрібно попарно перемножити кожен елемент рядка першої матриці на кожен елемент стовпця другої та додати їх. І таку операцію виконати над кожним рядком та стовпцем.

```
for I := M1 to N1 do
for J := M2 to N2 do
begin
       C[ I, J ] :=0;
       for K := M2  to N2 do
       C[ I, J] :=  C[ I, J] + A[ I, K] * B[ K, J]
end;
```

Якщо кількість рядків першої матриці не співпадає з кількістю стовпців другої, то перемножити такі матриці неможливо.

При ділені матриці „А“ на матрицю „В“ потрібно знайти обернену матрицю до „В“, а тоді матрицю „А“ перемножити на обернену до “В“.

Приклад 1

Дано матрицю дійсних чисел 3х3. Знайти добутки чисел більших 1, але менших 4 в кожному рядку окремо.

```
type
       matr=array [1..3, 1..3] of real;
var
       A   :  matr;
       I, J :  byte;
       D   :  real;
begin
       for  I  := 1  to  3   do
       for  J  := 1  to  3  do
       begin
              write ('A[',I , ',' , J, ']=');
              readln (A[I, J])
       end;
       for   I := 1  to  3  do
       begin
              for  J := 1 to  3  do
              if   (A[ I, J ] > 1)  and  (A[ I, J ] < 4)
                     then  D:= D*A[ I, J ];
              writeln (`D=`, D:7:3);
       end;
       readln
end.
```

Приклад 2

Дано матрицю цілих чисел 3х3. Поміняти в ній 1 та 3 рядки місцями.

```
type
       matr=array [1..3, 1..3]  of  integer;
var
       A    :  matr;
       I, J  :  byte;
       M    :  integer;
begin
       for I := 1  to  3  do
       for J := 1  to  3  do
       begin
              write ('A[', I, ',' J, ']=');
              readln (A[I, J])
```

```
        end;
        for I := 1  to  3  do
        begin
               M := A[ I, 1];
               A[ I, 1]  :=  A[ I, 3];
               A[ I, 3] := M
        end;
        for I := 1  to  3  do
               for J := 1  to  3  do
               write ('A[', I, ',' J, ']=', A[I, J]);
        end;
        readln
end.
```

В даній програмі використано проміжну змінну „M“ для того, щоб не втратити значення першого елемента під час переприсвоєння.

Розділ 7. Використання підпрограм у Turbo Pascal

7.1. Класифікація підпрограм. Глобальні та локальні ідентифікатори. Параметри значення та параметри змінні

В попередніх розділах посібника вже зустрічались терміни „процедура“ та „функція“. Їх у загальному можна назвати підпрограмою.

Підпрограма - це поіменований набір операторів та команд, які викликаються в потрібних місцях програми по імені і виконують у ній певну послідовність дій.

Підпрограми доречно використовувати, якщо в різних місцях програми потрібно використати одні й ті ж послідовності операторів (програмні блоки). Тоді їх краще об’єднати в підпрограму, записавши лише один раз. Для виконання цих програмних блоків у програмі достатньо вказати ім’я підпрограми.

Як вже відмічалось, в Turbo Pascal, підпрограми поділяються на процедури та функції.

Процедура - це підпрограма, що виконує в програмі певну послідовність дій або операцій.

Функція - це підпрограма, що виконує в програмі певні операції або обчислення і передає в точку виклику єдиний результат.

Основна відмінність між процедурами та функціями полягає в тому, що в результаті виконання процедури можна отримати цілий набір даних (наприклад матрицю чисел), тоді як у результаті виконання функції отримуємо єдиний результат. Крім цього процедури та функції по різному викликаються з основної програми. Виклик процедури можна розглядати як окрему команду в програмі, що виділяється від інших команд крапкою з комою. Для виклику в основній програмі процедури потрібно вказати таку конструкцію:

ім’я.процедури (*параметр1*, *параметр2*, ...);

Функції викликаються через оператор присвоєння всередині математичних виразів (як це було показано у вищеописаних прикладах з такими функціями як sin, cos і т.д.), в якості змінних в командах виведення інформації (write та writeln) або замість змінних в логічних виразах (наприклад, для перевірки умови в операторі if). Наприклад, для виклику функції в операторі присвоєння потрібно записати:

значення := *ім’я.функції* (*параметр1*, *параметр2*, ...);

Як видно з вищеописаного, при виклику процедур та функції часто використовуються **параметри**, що записуються після імені підпрограми в дужках через кому. Потрібно відмітити, що в Turbo Pascal можуть застосовуватись підпрограми без використання параметрів. В такому випадку, дужки після імені підпрограми також не використовуються.

Параметри - це вхідні дані для виконання підпрограми або значення, що отримується після її виконання (стосується лише процедур). Параметри в підпрограмах поділяються на формальні та фактичні.

Формальні параметри - це змінні, що використовуються в тексті підпрограми. Їх список записують після імені підпрограми в розділі, де вона створюється.

Фактичні параметри - це конкретні значення, що підставляються замість відповідних формальних параметрів при виконанні підпрограми. Їх записують після імені підпрограми в позиції її виклику в основній програмі (як показано вище). При вказанні параметрів потрібно враховувати, що кількість формальних та фактичних параметрів (для даної підпрограми) повинна співпадати.

Формальні параметри в Turbo Pascal поділяються на параметри значення та параметри змінні.

Параметри значення - це параметри, значення яких не змінюється після завершення виконання підпрограми. Тобто, по завершенні підпрограми в основній програмі ці параметри будуть мати ті ж значення, що й до її виконання.

Параметри змінні - це параметри, значення яких змінюється після завершення виконання підпрограми. Ці параметри зустрічаються лише в процедурах. Для того щоб відповідні параметри були параметрами змінними, потрібно в розділі створення підпрограми перед ними вказати службове слово **var**.

В якості параметрів у підпрограмах Turbo Pascal можна використовувати змінні, константи, вирази або інші функції. Змінні та константи можна ще назвати єдиним терміном „ідентифікатори". Потрібно відмітити, що ідентифікатори, в залежності від того, в якій частині програми вони описані, бувають локальні та глобальні.

Локальні - це ідентифікатори, які описані в середині даної підпрограми і діють лише в її межах.

Глобальні - це ідентифікатори, які описані в зовнішньому програмному модулі і діють як у ньому, так і у всіх його підпрограмах. Так, якщо ідентифікатор описаний в основній програмі, то він поширюється як у тексті основної програми, так і в підпрограмі.

Якщо імена локальних та глобальних ідентифікаторів співпадають, то в середині даної підпрограми діють локальні ідентифікатори, а за її межами - глобальні.

Потрібно відмітити, що в Turbo Pascal допускається використання вкладених структур, коли в середині однієї підпрограми описується інша. Тоді ця підпрограма буде діяти лише в межах зовнішньої підпрограми і її не можна використовувати в інших програмних блоках (наприклад, в основній програмі). В такому випадку ідентифікатори, описані в зовнішній підпрограмі, будуть глобальними для внутрішньої підпрограми і локальними для основної програми.

7.2. Підпрограми-функції. Приклади створення

В Turbo-Pascal підпрограми-функції бувають стандартні (наприклад sin, cos, odd, chr і т.д.) та функції, створені користувачем.

Для створення підпрограми-функції потрібно записати наступну конструкцію:

function *ім'я.функ*(*пар1*,*пар2*,...: тип1; *пар11*,*пар12*,... : тип2;...) : *тип.резул*;
[label *мітка1, мітка2 ...*;]
[const
ім'я = значення;
,,,]
[type
тип = опис типу;
,,,]
var
змінна1, змінна2, ... : тип;
,,,
begin
тіло функції
end;

В цій конструкції:

ім'я.функ - назва створюваної функції;
пар1,пар2,пар11,пар12,...- формальні параметри функції;
тип1, тип2 - тип відповідних формальних параметрів;
тип.резул - тип, що отримує результат виконання функції.

Як видно з вищеописаного, структура функції практично нічим не відрізняється від загальної структури Pascal-програми (див. розділ 3), тому функцію можна розглядати, як окрему завершену програму, що вирішує конкретну локальну задачу.

Функції можна створювати, як у самій програмі (після розділу **var**), так і в спеціальному файлі, що називають модулем (див. розділ 14).

Інколи виникає потреба завершити роботу функції раніше виконання останнього оператора в її тілі. В таких випадках слід використати спеціальний оператор передчасного завершення підпрограми. Для цього потрібно вказати оператор:

exit;

Розглянемо приклад програми, в якій використовуються функції створені користувачем.

Приклад

Дано матрицю цілих чисел 3х6. Знайти, в якому рядку найбільше чисел кратних 3.

```
type
       matr=array [1..3, 1..6] of integer;
var
       A   :  matr;
       I, J :  byte;

function  count(I : byte; A : matr) : byte;
       {створюємо функцію, що підраховує кількість чисел кратних 3}
var
       K : byte; {проміжна змінна, що буде визначати кількість чисел}
```

```
begin
       K := 0;
       for J := 1 to 6 do
       if  (A[I,J] mod 3)   then K := K+1;
       count := K   {значення проміжної змінної присвоюємо самій функції}
end;

begin
       for  I  := 1  to  3   do
              for  J  := 1  to  6  do
              begin
                     write ('A[',I , ',' ,J, ']=');
                     readln (A[I, J])
              end;
       if  (count(1,A)>=cont(2,A)) and (count(1,A)>=cont(3,A))
              then  writeln(' Найбільше чисел, що діляться на 3 в 1-му рядку')
              else
              if  (count(2,A)>=cont(1,A)) and (count(2,A)>=cont(3,A))
              then  writeln(' Найбільше чисел, що діляться на 3 в 2-му рядку')
              else  writeln(' Найбільше чисел, що діляться на 3 в 3-му рядку');
       readln
end.
```

7.3. Підпрограми-процедури. Приклади створення

В Turbo-Pascal підпрограми-процедури бувають стандартні (наприклад, writeln, readln, inc, dec і т.д.) та процедури, створені користувачем.

Для створення підпрограми-процедури потрібно записати наступну конструкцію:

procedure *ім'я.проц*(*пар1*,*пар2*,...:тип1; *пар11*,*пар12*,...: тип2; var *пар21*,*пар22*,...: тип3);
[label *мітка1, мітка2* ...;]
[const
ім'я = значення;
,,,]
[type
тип = опис типу;
,,,]
var
змінна1, змінна2, ... : тип;
,,,
begin
тіло процедури
end;

В цій конструкції:

ім'я.проц - назва створюваної процедури;

пар1,пар2,пар11,пар12,...- формальні параметри-значення;

тип1, тип2 - тип відповідних формальних параметрів-значень;

пар21,пар22... - формальні параметри-змінні. Потрібно відмітити, що при описі формальних параметрів-змінних перед ними слід указувати службове слово **var**;

тип1, тип2 - тип відповідних формальних параметрів-значень.

Як видно з вищеописаного, структура процедури практично нічим не відрізняється від загальної структури Pascal-програми (див. розділ 3), тому процедуру можна розглядати, як окрему завершену програму, що вирішує конкретну локальну задачу.

Процедури можна створювати, як у самій програмі (після розділу **var**), так і в спеціальному файлі, що називають модулем (див. розділ 14).

Інколи виникає потреба завершити роботу процедури раніше виконання останнього оператора в її тілі. В таких випадках слід використати спеціальний оператор передчасного завершення підпрограми. Для цього потрібно вказати:

exit;

Розглянемо приклад програми, в якій використовуються процедури створені користувачем.

Приклад

Дано матриці дійсних чисел A, B, C, D розміром 3x3. Обчислити вираз:

Z=(A+B)*C-A*B.

```
type
       matr=array [1..3, 1..3] of real;
var
       A, B, C, D, Z, M1, M2, M3 :  matr;
       I, J :  byte;
procedure  mkmatr(C : char; var A : matr); {процедура створення матриці}
       begin
              for I := 1 to 3 do
              for J := 1 to 3 do
              begin
                     write(C,'[', I, ',', J, ']=');
                     readln(A[I, J])
              end
       end;
procedure printmatr(C : char; A : matr);
       {процедура виведення результуючої матриці}
       begin
              for  I := 1 to 3  do
              for  I := 1 to 3  do
              writeln(' C, '[', I, ',', J, ']=', A[I, J]:7:3);
       end;
procedure  sum(A, B  matr; var C :matr);
       {процедура знаходження суми двох матриць}
       begin
              for I := 1 to 3 do
              for J := 1 to 3 do
              C[ I, J] := A[ I, J] + B[ I, J];
       end;
```

```
procedure  product(A, B  matr; var C :matr);
       {процедура знаходження добутку двох матриць}
       begin
              for I := M1 to N1 do
              for J := M2 to N2 do
              begin
                     C[ I, J ] :=0;
                     for K := M2  to N2 do
                     C[ I, J] :=  C[ I, J] + A[ I, K] * B[ K, J]
              end
       end;
procedure  negativ(A, B  matr; var C :matr);
       {процедура знаходження різниці двох матриць}
       begin
              for I := 1 to 3 do
              for J :-= 1 to 3 do
              C[ I, J] := A[ I, J] - B[ I, J];
       end;

begin
       mkmatr('A',A);
       mkmatr('B', B);
       mkmatr('C', C);
       mkmatr('D', D);
       sum(A, B, M1);
       product(M1, C, M2);
       product(A, B, M3);
       negativ(M2, M3, Z);
       printmatr('Z', Z);
       readln
end.
```

Turbo Pascal підтримує дві моделі виклику процедур - ближню **near** та віддалену **far**. Між цими моделями існує ряд відмінностей.

Процедури, створені з використанням моделі **near**, виявляються більш швидкодіючими (ефективними), але їх використання має ряд обмежень. Вони можуть бути викликані лише з модуля, в якому вони описані.

Процедури, створені з використанням моделі **far**, можуть бути викликані з будь-якого місця програми. Недоліком цих процедур є їх повільне виконання.

Компілятор Turbo Pascal 7.0 на основі опису процедури може автоматично вибрати оптимальну модель виклику. Якщо при розробці програми виникла необхідність вказати процедурі конкретну модель виклику, то в описі процедури, перед її основним блоком необхідно вказати команду **near** або **far**.

7.4. Рекурсивні підпрограми

В Turbo Pascal існує особлива група підпрограм, які називаються рекурсивними.

Рекурсивні підпрограми - це підпрограми, які викликають самі себе на виконання певну кількість разів (при цьому не використовується оператор циклу). Потрібно відмітити, що використання рекурсивної підпрограми значно скорочує текст програми.

Будь-яка рекурсивна підпрограма виконується у два етапи:

1-ий етап - „розгортання" підпрограми, коли підпрограма викликає сама себе до тих пір, поки не виконується певна умова. Після виконання умови виконується сама внутрішня підпрограма;

2-ий етап - зворотне „згортання" підпрограми, коли результат роботи внутрішньої підпрограми є вхідною інформацією для підпрограми, яка її викликала.

Розглянемо порядок роботи рекурсивної підпрограми на прикладі функції знаходження факторіала числа N (N!=1*2*3*4*...*N).

```
function  fact(N : byte) : longint;
        begin
                if  N=1 then  fact := 1
                else  fact := N*fact(N-1)
        end;
```

Для прикладу візьмемо, що N=4, тоді на першому етапі перевіряється умова „if 4=1". Ця умова не виконується, тому виконується команда „fact := 4*fact(4-1)". Отже підпрограма-функція викликала сама себе, але вже з N=4-1=3.

Знову перевіряється умова. Вона не виконується, отже виконується команда „fact := 3*fact(3-1)". Функція знову викликала сама себе, але вже з параметром N=2. Цей процес повторяється, поки N не досягне 1. Після цього виконується команда „fact := 1". На цьому перший етап роботи підпрограми завершився.

На другому етапі знайдене значення функції підставляється у функцію, що її викликала, тобто fact(2)=2*fact(1)=2*1=2. Це значення підставляється в наступну зовнішню підпрограму fact(3)=3*fact(2)=3*2=6, тоді - fact(4)=4*fact(3)=4*6=24. На цьому робота рекурсивної підпрограми завершується.

Приклад

Знайти значення виразу $Y=A_0 \cdot X^{15} + A_1 \cdot X^{14} + A_2 \cdot X^{13} + ... + A_{13} \cdot X^2 + A_{14} \cdot X + A_{15}$

Цей вираз можна математично перетворити, винісши по черзі за дужки X. Отримаємо $Y=(((...(((A_0 \cdot X+A_1) \cdot X+A_2) \cdot X + A_3)...) \cdot X + A_{14}) \cdot X)+A_{15}$. Тепер можна застосувати рекурсивну підпрограму.

```
function  binom(X : real; A : mas; N : byte);
        begin
                if  N=0  then  binom := A[0]
                        else  binom :=  binom(X, A, N -1)*X+A[N]
        end;
```

В даному прикладі змінна A описана як масив з 16-ти елементів, що нумеруються від 0 до 15.

Розділ 8. Робота з множинами в Turbo Pascal

8.1. Поняття про множини. Операції над множинами

Множина - це не впорядкований набір даних простого типу, які не повторюються. Елементами множини можуть бути символи, логічні дані або цілі числа в діапазоні від 0 до 255. Отже, кількість елементів будь-якої множини не може перевищувати 256. Елементи множини в Turbo Pascal беруться в квадратні дужки і відділяються один від одного комою.

Приклади множин:

[1, 4, 54, 3, 67] - множина з п'яти цілих чисел;

['Ф', 'в', 'd', '5', '!', 'y'] - множина з шести символів типу **char**;

['A'..'Z', 'a'..'z'] - множина з великих та малих латинських літер (задана як діапазони символів);

[ ] - порожня множина (не містить ні одного елемента).

В Turbo Pascal множини описуються в розділі **type**. Для цього необхідно вказати наступну конструкцію:

```
type
        ім'я = set of тип;
var
        змінна1, змінна2, ... : ім'я;
```

де

ім'я - назва типу множини;

змінна1, змінна2 - назви змінних, типу множини;

тип - тип елементів множини.

Приклад:

```
type
     Mn=set  of  byte;  {множина цілих чисел}
var
     A, B, C : Mn;
```

Над множинами можна виконувати наступні операції.

1. **A<=B - входження однієї множини в іншу**. Дана операція дає результат TRUE, якщо всі елементи множини A входять у множину B. Наприклад, якщо A:=[1,2,5], а B:=[3,7,1,5,2], то буде отримано результат TRUE. У випадку, якщо A:=[1,2,5,4], а B:=[3,7,1,5,2], то результат буде FALSE, оскільки цифра 4 не входить у множину B;
2. **A=B - рівність множин**. Дана операція дає результат TRUE, якщо всі елементи множин A та B одинакові. Наприклад, множини [3,7,5] та [5,7,3] рівні між собою (результат TRUE);
3. **A<>B - не рівність множин**. Дана операція обернена до попередньої, тобто, якщо множини не рівні, то отримуємо результат TRUE;
4. **C:=A+B - об'єднання множин**. В даному випадку утворюється множина C, в яку входять всі елементи множин A і B. Наприклад, якщо A:=[9,2,5], а B:=[3,7,1,5,2],

то буде отримано множину C:=[9,3,7,1,5,2] (нагадаю, що елементи в множину можуть входити лише по разу);

5. **C:=A-B - виключення множини**. В даному випадку утворюється множина C, в яку входять всі елементи множини A, які не входять в B. Наприклад, якщо A:=[9,2,5], а B:=[3,7,1,5,2], то буде отримано множину C:=[9] (якщо знайти C:= B-A, то отримаємо B:=[3,7,1]);
6. **C:=A*B - переріз множин**. Утворюється множина C з елементів, які одночасно входять в A і B. Наприклад, якщо A:=[9,2,5], а B:=[3,7,1,5,2], то буде отримано множину C:=[2,5];
7. **X in A - входження елемента в множину**. Дана операція встановлює результат TRUE, якщо значення змінної X міститься серед елементів множини A. Наприклад, якщо значення змінної X рівне 5, а множина A містить елементи [6,4,5,7,12], то результат операції буде TRUE.

8.2. Приклади типових програм по обробці множин

Особливістю множин є те, що в Pascal-програмі не можна вводити та виводити елементи множини безпосередньо процедурами **write** та **read**. Тому, щоб ввести елементи у множину слід скористатись „штучним“ способом, ввівши додатково проміжну змінну, наприклад,:

```
write(`X=`);
readln(X);
A := A + [X];
```

Потрібно відмітити, що перед тим, як вводити перший елемент у множину потрібно створити порожню множину A:=[ ]. Після цього вводити всі наступні елементи в циклі, задавши певну умову, наприклад, поки не буде введено число 0 або символ „пропуск“ і т.д.

Щоб вивести елементи множини, потрібно також скористатись „штучним“ способом. Для цього в циклі необхідно міняти значення проміжної змінної так ,щоб вона набула всіх можливих значень. При цьому потрібно перевіряти входження цієї змінної у множину. Наприклад, для виведення елементів множини цілих чисел можна записати:

```
for  X := 0  to  255 do
       if  X   in  A
       then  writeln(X);
```

Для виведення елементів множини символів можна записати:

```
       for   J := 0  to  255 do
       {знаходження символу по коду й перевірка входження символу}
       if  chr(J)  in  A
              then  writeln(chr(J));
```

Розглянемо приклад програми, в якій використовується множина.

Приклад

Дано ціле натуральне число N. Перерахувати, які цифри входять у нього.

```
type
        Mn = set of  0..9;  {опис множини цифр, що можуть входити в число}
var
        A : Mn;
        N : Word;
        K : byte;
begin
        A := [ ]; {створення порожньої множини}
        write ('Введіть N=');
        readln (N);
        while N > 0  do
        begin
                K := N  mod 10; {відділення від числа останньої цифри}
                A:=A+[K];  {добавлення до множини цієї цифри}
                N:=N div 10  {відкидання від числа останньої цифри}
        end;
        for K :=0  to  9  do
                if  K  in  A then
                        writeln('У число входить цифра', K);
        readln;
end.
```

Надалі розглянемо декілька прикладів типових підпрограм по обробці множин.

Підпрограма 1

Створення множини цілих чисел

```
procedure  mk_mn (var A : mn);  {type  mn=set  of  byte}
var
        X : integer;
begin
        A:=[];   {створення порожньої множини}
        write ('X=');  {введення першого проміжного елемента}
        readln(X);
        {вказання умови поки вводити елементи в множину}
        while  (X>255) and (X<0)  do
                begin
                        A := A + [X]; {введення елемента в множину}
                        write ('X='); {введення наступного проміжного елемента}
                        readln(X)
                end
end;
```

Підпрограма 2

Створення множини символів

```
procedure  mk_mn (var A : mn);  {type  mn=set  of  char}
var
        X : char;
begin
        A:=[];  {створення порожньої множини}
        write ('X='); {введення першого проміжного елемента}
        readln(X);
        {вказання умови поки вводити елементи в множину}
        while  X<>' '  do  {поки не буде введено пропуск}
```

```
                begin
                        A := A + [X]; {введення елемента в множину}
                        write ('X='); {введення наступного проміжного елемента}
                        readln(X)
                end
end;
```

Підпрограма 3

Виведення елементів множини, що складається з малих українських літер

```
procedure  print_mn (A : mn); {type  mn=set  of  'a'..'я'}
var
       X : char;
begin
       for   X := 'a'  to 'я'  do
       if   X  in  A
              then
              write (X,' ');
       readln
end;
```

Підпрограма 4

Вивести елементи, що входять одночасно у дві множини A та B.

```
procedure  print_mn (A, B : mn); {type  mn=set  of  byte}
var
       X : byte;
begin
       for   X := 0  to  255  do
       if   X  in  (A * B)
              then
              write (X,' ');
       readln
end;
```

Підпрограма 5

Визначити, чи в множину символів входять цифри.

```
function  digit (A : mn) : boolean;  {type  mn=set  of  char}
var
       B, C : mn;
       X : boolean;
begin
       B := ['0'..'9'];
       C := A * B;
       if   C=[]  then X := false else X := true;
       digit := X;
end;
```

Розділ 9. Стрічки в Turbo Pascal

9.1. Поняття про стрічки. Стандартні процедури та функції для роботи із стрічками

Стрічка (String) - це складний тип даних, елементами якого є впорядкований набір символів до 255. Стрічкову константу беруть в одинарні лапки. Змінна типу **string** складається з одномірної послідовності символів типу **char**. Фактично стрічку можна розглядати як одномірний масив символів. Але стрічка це окремий тип даних, який описується в розділі **type**. Для цього потрібно вказати наступну конструкцію:

type

***ім'я* = string[N];**

var

***змінна1, змінна2 ...* : *ім'я*;**

де

ім'я - назва типу множини;

змінна1, змінна2 - назви змінних, типу множини;

тип - тип елементів множини.

N - довжина стрічки в символах. Якщо цей параметр не вказати (разом із квадратними дужками), то довжина стрічки сприймається 255 символів.

Потрібно відмітити, що стрічки допускається описувати безпосередньо в розділі **var** (автори посібника радять надавати цьому способу перевагу). Для цього потрібно вказати конструкцію:

var

***змінна1, змінна2 ...* : string[N];**

Наприклад

```
var
       S1, S2  : string;  {стрічки довжиною 255 символів}
       S3, S4  : string[50]; {стрічки довжиною 50 символів}
```

Як вже відмічалось, стрічки подібні на одномірні масиви символів, тобто кожен їх, елемент (символ) має свій порядковий номер. Щоб викликати символ із певним номером, слід вказати назву змінної, а після неї у квадратних дужках номер елемента. Потрібно відмітити, що в кожній стрічці під порядковим номером 0 знаходиться число, що вказує довжину даної стрічки. Наприклад, щоб знайти довжину стрічки можна вказати S[0], де S - назва змінної.

Змінні стрічкового типу в Turbo Pascal можна вводити та виводити стандартними процедурами **read** (**readln**) та **write** (**writeln**). Крім цього для змінних стрічкового типу можна використовувати оператор присвоєння та операцію **об'єднання стрічок**. Для об'єднання стрічок між ними достатньо вказати операнд **+**, наприклад:

S := S1 + S2 + S3 + 'слово';

В даному випадку в кінець стрічки S1 "доклеюється" стрічка S2, а тоді в її кінець - S3. Після цього в кінець результуючої стрічки "доклеюється" стрічкова константа 'слово' і ця результуюча стрічка присвоюється змінній S.

Крім цього в Turbo Pascal є цілий ряд стандартних процедур та функцій для обробки стрічок.

***Функція* Length(S)**. Дана функція дозволяє визначити довжину стрічки S. Результат виконання даної функції - величина цілого типу (**byte**). Нагадаю, що довжину стрічки можна також визначити за її елементом з нульовим порядковим номером.

***Функція* Pos(S1,S2)** - визначає позицію, з якої підстрічка S1 входить в стрічку S2. В результаті виконання даної функції, отримується величина цілого типу (**byte**). Наприклад, результатом виконання функції pos('тон','картон') буде число 4, оскільки підстрічка 'тон' входить у стрічку 'картон' з 4-ої позиції. Функція **pos** вимагає повної відповідності шуканої підстрічки і фрагменту стрічки, в якій проходить пошук. Причому, великі та малі літери вважаються різними літерами. Якщо фрагмент у стрічці не знайдено, то функція видає результат 0 (нуль).

***Функції* UpCase(S[N])** - перетворює в стрічці S символ, що знаходиться на позиції N у цій стрічці у велику літеру. Потрібно відмітити, що дана функція не діє на символи кирилиці. Результатом виконання функції буде стрічка, в якій змінено регістр одного символу (з номером N). Наприклад, результатом виконання функції UpCase(S[6]), де S='Microsoft' буде стрічка 'MicroSoft'.

***Функції* Concat(S1,S2,S3, ...)** - об'єднує стрічки S1, S2, S3 і т.д. В результаті виконання функції буде стрічка, в якій по порядку „склеєно" всі стрічки, які є параметрами функції. Наприклад, результатом виконання функції S:=Concat('abc','def') буде стрічка S = 'abcdef'.

***Функції* Copy(S,M,N)** - копіює зі стрічки S, починаючи з позиції M, N символів підряд. Наприклад, результатом виконання функції S:=Copy('абракадабра',2,4) буде стрічка S='брак'.

***Процедура* Insert(S1,S,N)**. Дана процедура дозволяє вставити в стрічку S починаючи з позиції N підстрічку S1. Наприклад, якщо стрічка S='баран', а S1='*аб*', то результатом виконання процедури Insert(S1,S,4) буде стрічка 'бар*аб*ан'.

***Процедура* Delete(S,M,N)** - дозволяє знищити в стрічці S, починаючи з позиції M, N символів. Наприклад, якщо стрічка S='робота', то результатом виконання процедури Delete(S,3,2) буде стрічка 'рота' (знищено з 3-ої позиції два символи).

***Процедура* Str(X:N:M,S)**. Дана процедура перетворює число X (дійсного або цілого типу) у стрічку S. Параметр N вказує скільки символів виділяти для числа, а M - кількість знаків після коми. Потрібно відмітити, що параметри N та M є необов'язковими (для цілих чисел їх узагалі вказувати не бажано).

***Процедура* Val(S,X,C)**. Дана процедура перетворює стрічку S у число X (дійсного або цілого типу). Параметр C (тип Integer) указує код успішності виконання операції. Цей параметр є вихідним і по ньому можна визначити на скільки успішно стрічка перетворилась в число, адже не кожну стрічку можна перетворити в число ('3452' можна перетворити, тоді як '345w' - не можливо). Якщо стрічку успішно перетворено в число, то код успішності рівний нулю (C=0). При невдалому перетворені код успішності рівний числу, що показує номер позиції у стрічці, з якої не можливе перетворення (для стрічки '345w' код успішності - 4).

9.2. Приклади типових програм по обробці стрічок

Розглянемо декілька прикладів типових підпрограм для обробки стрічок.

Підпрограма 1

Порахувати кількість українських голосних літер у стрічці.

Для розв'язку даної задачі можна скористатись множиною голосних літер та операцією перевірки входження елемента (символу стрічки) у множину (див. розділ 8).

```
function count_gol(S : string) : byte;
var
     A : set of char;
     I, K : byte;
begin
       A := ['А', 'О', 'У', 'Е', 'И', 'І', 'а', 'о', 'у', 'е', 'и', 'і'];
       K := 0;
       {організовуємо цикл від першого до останнього символу стрічки}
       for I := 1 to length(S) do
       if S[I] in A then K := K + 1;
       count_gol := K
end;
```

Підпрограма 2

Записати стрічку символів у зворотному порядку.

```
function retr(S : string) : string;
var
       I : byte;
       S1 : string;
begin
       S1 := ''; {створення порожньої стрічки}
       {організовуємо цикл від останнього елемента стрічки до першого}
       for I := length(S) downto 1 do
       S1 := S1 + S[I]; {доклеюємо до нової стрічки по одному символу}
       retr := S1 {утворену стрічку присвоюємо функції}
end;
```

Цю підпрограму можна організувати, по іншому вказавши замість циклу по спаданню цикл по зростанню. Для цього записуємо

```
              ...
       for I := 1 to length(S) do
       S1 := S[I] + S1;
              ...
```

Підпрограма 3

Знищити в стрічці S символів усі пропуски.

Для виконання даної підпрограми введемо додаткову стрічку S1, куди будемо копіювати фрагмент стрічки S від першого символу до першого пропуску (не включно). Після цього, у стрічці S, знищуємо всі символи від першого символу до першого пропуску (включно). Так слід повторювати ці дії поки в стрічці S будуть пропуски.

```
function  del_space(S : string) : string;
var
       K  :  byte;
       S1  : string;
begin
       S1 := '';  {створення порожньої стрічки}
       K := pos(' ',S);  {шукаємо пропуск у стрічці S}
       while  K<>0  do   {цикл поки в стрічці є пропуски}
       begin
              S1 := S1 + copy(S,1,K-1);
              delete(S,1,K)
       end;
       S1 := S1 + S;  {доклеїти до стрічки S1 останнє слово}
       del_space := S1
end;
```

Підпрограма 4

Як відомо, функція UpCase не може перетворити малі символи кирилиці у великі. Створити функцію, яка буде перетворювати всі літери стрічки S (як латинські так і кириличні) у великі.

В середині створюваної функції створимо внутрішню підпрограму, що буде перетворювати лише одну літеру українського алфавіту у велику. В основній підпрограмі організуємо цикл від першого до останнього символу стрічки, де будемо викликати внутрішню підпрограму.

```
function  UpCaseAll(S : string) : string;
var
       I  : byte;
function UpCaseOne(S : string; I : byte) : string; {перетворення одного
символу}
       begin
       S[I] := UpCase(S[I]); {перетворення малої латинської літери у велику}
       {якщо дана літера латинська, то наступний оператор ігнорується}
       case  S[I]  of
              'a' : S[I] := 'A';
              'б' : S[I] := 'B';
              'в' : S[I] := 'B';
              ... {таким самим чином слід описати всі літери кирилиці}
              end;
       UpCaseOne := S
       end;
begin
       for I := 1  to  length(S)  do
       S[I] :=  UpCaseOne(S[I]);
       UpCaseAll := S
end;
```

Розділ 10. Записи в Turbo Pascal

Розглянемо ще один складний тип даних, так звані **записи (record)**, які дозволяють зберігати разом змінні, що належать до різних типів даних (будь-яких простих, стрічок або масивів).

За допомогою одного запису представляється деяка структура статичних даних, яку можна розглядати, як картку в каталозі або деякий бланк. На кожній картці в каталозі (картотеці) записана деяка інформація. Ця інформація неоднорідна, і окремі її частини мають різний вміст, оформляються різними способами і зберігаються в окремих полях картки. Так при обліку працівників підприємства картка буде містити поля: імена, прізвища, дати народження, стаж роботи і т.д. Отже записи в основному використовуються для організації **баз даних (database)**.

Як уже відмічалось, кожен запис складається з окремих елементів різних типів, що називаються **полями** (**Field**). При записуванні в тексті програми записи та поля розділяються між собою крапкою, тобто записуємо **record.field**. Наприклад, вираз kadry.name , означає, що ми маємо справу із записом kadry, в якому є поле name.

Тип даних записи описують у розділі **type**. Для записуємо наступну конструкцію:

type
ім'я = record
поле1 : *тип1*;
поле2 : *тип2*;
поле3 : *тип3*;
... *{і так далі }*
end;
var
змінна1, *змінна2*, ... : *ім'я*;

де

ім'я - назва типу записів;

поле1, поле2, ... - назви полів запису;

тип1, тип2, ... - типи полів запису;

змінна1, змінна2, ... - назви змінних типу записи.

Для прикладу спробуємо описати тип даних записи, для бази даних працівників підприємства:

```
type
       kadry = record
       fam : string[15];  {прізвища працівників}
       name : string[15];  {імена працівників}
       fath : string[20];  {по-батькові працівників}
       date : string[10];  {дата народження}
       stan : boolean;  {сімейний стан}
       stag : byte;  {стаж роботи}
       zarp : real;  {середня зарплата}
end;
var
       Prac : kadry;
```

В якості іншого прикладу використання записів можна представити опис комплексних чисел (числа, що складаються з двох частин: дійсної та уявної).У такому випадку можна записати таку конструкцію:

```
type
       compl = record
       re  : real;  {дійсна частина}
       imp : real;  {уявна частина}
end
```

Для введення та виведення записів можна скористатись звичайними процедурами введення та виведення інформації: **read** (**readln**) та **write** (**writeln**). Але при введенні або виведенні інформації потрібно окремо звертатись до кожного поля. Наприклад, щоб увести один запис бази даних працівників, тип якого описаний вище потрібно вказати таку конструкцію:

```
write ('Прізвище : ');
readln (Prac.fam);
write ('Ім''я  : ');
readln (Prac.name);
write ('По батькові : ');
readln (Prac.fath);
       ...   {і так далі}
```

З вищесказаного можна зробити висновок, що звертання до конкретних полів запису пов'язані з деякою незручністю (утворюються довгі складові ідентифікатори). Але при спільній обробці декількох полів запису (так як це показано вище) довжину ідентифікатора можна скоротити. Для цього використовується спеціальний оператор, який називають оператором приєднання.

Оператор приєднання має наступний формат запису:

with ***змінна*** **do**
begin
    команди
end;
де

змінна - назва змінної, що визначає запис;

команди - список команд, в яких спільно обробляються поля одного запису.

Наприклад, для вищеописаної послідовності процедур введення-виведення інформації можна записати:

```
With  Prac  do
begin
       write('Прізвище : ');
       readln(fam);
       write('Ім''я  : ');
       readln(name);
       write('По батькові : ');
       readln(fath);
              ...   {і так далі}
end;
```

Особливу групу записів становлять **записи з варіантами**, які описуються з використанням оператора **case**. За допомогою записів з варіантами можна одночасно зберігати різні структури даних, що мають велику спільну частину, однакову у всіх

структурах, і невеликі частини, які відрізняються в різних структурах. Наприклад, розглянемо опис запису працівників Kadry, ввівши в нього ще одне поле Hobby:

```
type
        kadry = record
        fam : string[15];
        name : string[15];
        fath : string[20];
                ...  {і так далі}
        case Hobby : (Computer, Music)  of
                Computer : (Type_computer :  string[20];
                        Mbyte : byte;
                        Compatible :  boolean);
                Music  :  Instrument : array[1..3]  of   string[10];
        end;
var
        Prac : kadry;
```

Додаткове поле Hobby визначає хобі працівника і може мати різну структуру в залежності від його інтересів. Так, наприклад, для працівника, що цікавиться комп'ютерами Hobby має три поля, що призначені для збереження інформації про тип його комп'ютера (Type_computer), об'єм оперативної пам'яті (Mbyte) та сумісність з вашим ПК. Якщо ж працівник цікавиться музикою, то в масиві стрічок Instrument перераховуються інструменти, на яких він грає.

Автори посібника не будуть в цьому розділі розгорнуто розглядати приклади програм із записами, відклавши їх до завершення вивчення теми обробки файлів.

Розділ 11. Тип даних, файли

11.1. Класифікація файлових типів у Turbo Pascal

Файл (file) - це пойменоване місце на диску, де зберігається набір однотипної інформації. Завдяки файловому типу дані можна зберігати на диску і використовувати кожен раз під час виконання програми. Дані, що зберігаються у файлі називаються його компонентами. Для більшості типів файлів кожен з компонентів має свій порядковий номер. Як правило. нумерація компонентів починається з нуля.

При обробці файлів один із компонентів (над яким виконуємо операції зчитування або запису) повинен бути активним. На активність компоненту вказує спеціальний мітчик, який називають **маркером**.

В Turbo Pascal розрізняють три файлових типи даних:

- **файли вказаного типу** - файли, в яких всі компоненти належать до одного із стандартних у Turbo Pascal типів даних;
- **текстові файли** - файли, в яких компоненти є текстовими стрічками;
- **файли без типу (двійкові файли)** - файли, в яких компоненти не належать ні до одного зі стандартних типів, а являються двійковими кодами певної інформації.

Крім цього, файли поділяють за типом доступу до їх даних. За цією ознакою можна виділити:

- **файли з прямим (безпосереднім) доступом** - це файли, в яких можливий безпосередній доступ до будь-яких їх компонентів. Для цього потрібно вказати лише координати відповідного компоненту файлу;
- **файли з послідовним доступом** - це файли, в яких, щоб зчитати будь-який компонент з номером N, потрібно спочатку послідовно зчитати всі N-1 компоненти.

До файлів з прямим доступом належать файли вказаного типу та файли без типу, тоді як текстові файли належать до файлів з послідовним доступом.

Для роботи із файлами будь-якого типу їх спочатку потрібно описати в розділі **type**, а відповідні їм змінні в розділі **var** (про що описано в розділах 11.2,11.3, 11.4 даного посібника).

На початку тексту програми змінні файлового типу потрібно співставити (асоціювати) із відповідними іменами файлів на диску. Для чого використовується процедура

assign(*змінна_файл*, *ім'я файлу*);

де

змінна_файл - змінна, що відповідає даному файлу;

ім'я файлу - текстова стрічка (змінна або константа) яка визначає ім'я файлу на диску. Потрібно нагадати, що для ОС MS-DOS ім'я файлу не повинно перевищувати 8 символів, крім цього через крапку бажано вказувати розширення (до 3 символів).

Для того щоб виконувати будь-які операції над компонентами файлів, потрібно спочатку їх відкрити (див. нижче). В кінці програми всі файли потрібно закрити, для чого використовується процедура:

close(*змінна_файл*);

де

змінна_файл - змінна, що відповідає даному файлу.

В Turbo Pascal є стандартний набір процедур, що дозволяють виконувати операції не лише над файлами, але й над каталогами, в яких ці файли розміщені. При цьому можна виділити наступні процедури:

- процедура **ChDir(*стр*)** - встановлює активний каталог. В якості параметра ***стр*** встановлюють текстову стрічку, в якій указано шлях до каталогу, що потрібно зробити активним;
- процедура **GetDir(*N, стр*)** - встановлює поточний каталог вказаного диска. Значення ***N*** - це число типу byte, що вказує диск (0 - диск по замовчуванню, 1 - диск A, 2 - диск B, 3 - диск C і т.д.). В якості параметра ***стр*** встановлюють текстову стрічку, в якій вказано шлях до каталогу, що потрібно зробити активним;
- процедура **MkDir(*стр*)** - створює новий підкаталог. В якості параметра ***стр*** встановлюють текстову стрічку, в якій вказано ім'я створюваного підкаталогу;
- процедура **RmDir(*стр*)** - знищує порожній підкаталог. В якості параметра ***стр*** встановлюють текстову стрічку, в якій указано ім'я потрібного підкаталогу;

11.2. Файли вказаного типу. Стандартні процедури та функції для роботи з ними

Файли вказаного типу є одним із найпоширеніших файлових типів даних, які використовуються в Pascal-програмах. Їх компонентами можуть бути дані будь-яких інших типів, що використовуються в Turbo Pascal, але найчастіше - це записи.

Файловий тип даних описується в розділі **type**. Для цього необхідно записати наступну конструкцію:

type

ім'я = file of *тип*;

var

змінна_файл1, *змінна_файл2*, ... : *ім'я*;

де

ім'я - назва описаного файлового типу ;

тип - типи компонентів файлу;

змінна_файл1, змінна_файл2, ... - назви змінних, що містять дані файлового типу

Для прикладу опишемо файли цілих чисел, символів та стрічок:

```
type
       file_int = file of  integer;
       file_chr = file of char;
       file_str =  file of string;
var
       F1, F2 : file_int;
       A, B    : file_chr;
       F3, Z   : file_str;
```

Як вже відмічалось, змінні файлового типу спочатку необхідно співставити з відповідним іменем на диску (процедура **assign**). Після чого файли потрібно відкрити, для чого існує два способи:

- процедура **rewrite(*змінна_файл*)** - відкриває файл для запису в нього даних (змінна вказує на потрібний файл), а маркер розміщується на початок файлу. При цьому знищуються всі дані, що існували у файлі до цього моменту. Дану процедуру використовують переважно для створення нового файлу;
- процедура **reset(*змінна_файл*)** - відкриває файл для запису або зчитування даних (змінна вказує на потрібний файл). При цьому маркер розміщується на початок файлу. Потрібно відмітити, що для використання даної процедури необхідно, щоб такий файл уже існував. Якщо такий файл не існує, то виникає помилкова ситуація. Щоб усунути зависання програми в цій ситуації використовують директиву **{$I-}**, яка блокує цю помилку (про що буде описано нижче);

Файли вказаного типу належать до файлів з прямим доступом. Щоб звернутись до будь-якої компоненти такого файлу необхідно знати її номер. Якщо номер відомий, то можна скористатись процедурою:

seek(*змінна_файл*, *номер*);.

Наприклад, процедура seek(F, 5) переміщає маркер у файлі F на компоненту з номером 5. Нагадаємо, що нумерація компонентів у файлі починається з нуля.

Для того, щоб зчитати активну компоненту із файлу потрібно скористатись процедурою:

read(*змінна_файл*, *змінна*);.

Наприклад:

```
read(F, X);
```

При виконанні цієї процедури зчитується активна компонента з файлу F і її значення присвоюється змінній X. При цьому маркер автоматично переміщається на наступну компоненту.

Для запису у файл нової компоненти потрібно скористатись процедурою:

write(*змінна_файл*, *змінна*);.

Наприклад

```
write(F, X);
```

При виконанні даної процедури значення змінної X записується в позицію маркера файлу F. При цьому маркер переміщається на наступний номер компоненти. Потрібно відмітити, що для введення нових компонентів у файл потрібно помістити маркер на останню компоненту файлу.

При роботі з файлами вказаного типу можна також скористатись наступними процедурами та функціями:

- процедура **rename(*змінна_файл, ім'я_файлу*)** - дозволяє перейменувати файл, зв'язаний із змінною ***змінна_файл***, співставивши його з іменем ***ім'я_файлу***. При цьому в якості ***ім'я_файлу*** може бути змінна або константа стрічкового типу, яка призначить файлу нове ім'я (наприклад, rename(F,'test.dat););
- процедура **erase(*змінна_файл*)** - знищує існуючий файл з диску. При цьому

файлова змінна повинна бути перед тим зв'язаною з існуючим іменем на диску (наприклад, erase(F););

- процедура **Truncate(*змінна_файл*)** - обмежує розмір файлу активною позицією маркера. В результаті виконання цієї процедури всі дані у файлі, що знаходяться після активної компоненти знищуються;
- функція **eof(*змінна_файл*)** - дозволяє визначити чи досяг маркер кінця файлу. При цьому, якщо маркер знаходиться в кінці файлу, то функція видає результат **TRUE**. В усіх інших випадках отримаємо результат **FALSE**;
- функція **FileSize(*змінна_файл*)** - визначає розмір файлу ***змінна_файл***. Результатом роботи цієї функції є число типу **LongInt**, яке вказує на кількість компонентів у файлі;
- функція **FilePos(*змінна_файл*)** - визначає номер активної компоненти у файлі ***змінна_файл***. Результатом роботи цієї функції є число типу **LongInt**, яке показує номер компоненти, на якій знаходиться маркер;
- функція **IOResult** - призначена для пошуку помилок, що виникають при роботі з файлами. Ця функція видає результат останньої операції введення-виведення, якщо директивою **{$I-}** відключено автоматичний контроль за помилками, що виникають при виконанні операцій введення-виведення. При безпомилковому виконанні операцій введення-виведення функція IOResult встановлює результат рівний нулю. Тому, цю функцію, як правило, використовують в операціях порівняння з нулем (на рівність або нерівність). Нижче наведено приклад фрагменту програми, де функція IOResult використовується для перевірки достовірності файлу test.dat.

```
assign(F, 'test.dat');
{$I-} {відключення контролю помилок введення-виведення}
        reset(F);
        if  IOResult <> 0
              then  write('помилка при відкритті файлу');
{$I+} {включення контролю помилок  введення-виведення}
```

Введення даних у файл (створення файлу) здійснюється з використанням оператора циклу (як правило, з передумовою або післяумовою). При цьому використовують два способи. Перший спосіб - це введення даних до тих пір, поки не буде введено певне значення. Для прикладу розглянемо процедуру створення файлу цілих чисел, в якому не повинно бути значень рівних 0.

```
procedure  mk_file(var  F : file  of  integer);
var
        S : string;
        X : integer;
begin
        write ('Введіть ім'я створюваного файлу  -  ');
        readln(S);
        assign(F, S);
        rewrite (F);
        write('Введіть компоненту файлу  X=');
        readln(X);
        while  X<>0 do
```

```
                begin
                     write (F,X);
                     write('Введіть компоненту файлу  X=');
                readln(X)
        end;
        close(F)
end;
```

Для створення файлу символів або стрічок у якості умови можна задати „поки не буде введено пропуск“ (while X <> ' ').

Інший спосіб створення файлів - це використання діалогового режиму. При цьому потрібно ввести додаткову змінну символьного типу, яка буде відповідати за організацію діалогового режиму.

```
procedure mk_file(var F : file of integer);
var
        S : string;
        C : char;
        X : integer;
begin
        write ('Введіть ім'я створюваного файлу - ');
        readln(S);
        assign(F, S);
        rewrite (F);
        repeat
                write('Введіть компоненту файлу X=');
                readln(X);
                write(F, X);
                write('Чи будете вводити дані ще? (Y/N) - ');
                readln(C);
        until (C='N') or (C='n');
        close(F)
end;
```

При зчитуванні даних із файлу, як правило, організовують цикл з передумовою, де в якості умови визначають чи досягнуто кінця файлу, для чого використовують конструкцію **while not eof(F) do** („поки не досягнуто кінця файлу F).

Як вже відмічалось, в якості файлів вказаного типу найчастіше використовуються файли записів. Розглянемо декілька типових процедур обробки файлів типу запис, на прикладі бази даних працівників підприємства, що описана в розділі 10 даного посібника.

Для опису такого файлового типу в розділі **type** потрібно ввести наступну конструкцію:

```
        type
                kadry = record
                fam  : string[15];  {прізвища працівників}
                name : string[15];  {імена працівників}
                fath  : string[20];  {по-батькові працівників}
                date  : string[10];  {дата народження}
                stan  : boolean;    {сімейний стан}
                stag  : byte;         {стаж роботи}
                zarp  : real;          {середня зарплата}
                end;
                file_kadr : kadry;
        var
                F : file_kadry;
```

Розглянемо процедуру створення файлу бази даних працівників підприємства:

```
procedure  mk_bd(var F :  file_kadry);
var
       S : string;
       C : char;
       X : kadry;
begin
       write (’Введіть ім’я створюваного файлу  -  ’);
       readln(S);
       assign(F, S);
       rewrite (F);
       repeat
              with  X  do
              begin
                     write(’Прізвище : ’);
                     readln(fam);
                     write(’Ім’’я  : ’);
                     readln(name);
                     write(’По-батькові : ’);
                     readln(fath);
                     ...    {і так далі}
              end;
              write(F,X);   {запис компоненти у файл}
              write(’Чи будете вводити дані ще? (Y/N) – ’);
              readln(C);
       until  (C=’N’) or (C=’n’);
       close(F)
end;
```

Процедура перегляду бази даних буде мати вигляд:

```
procedure  list_all(var  F : file_kadry);
var
       S : string;
       X : kadry;
begin
       write (’Введіть ім’я файлу бази даних -  ’);
       readln(S);
       assign(F, S);
       {$I-}    {відключення контролю помилок введення-виведення}
       reset(F);
       if  IOResult <> 0
              then
              begin
                     write(’Файл з таким іменем відсутній або не можливо
                            відкрити’);
                     halt  {припинити виконання програми}
              end;
       {$I+}     {включення контролю помилок  введення-виведення}
       writeln(’| Прізвище     |  Ім’я       |  По батькові    |  ...’);
       while not eof(F) do
              begin
                     read(F, X); {зчитування активної компоненти з файлу}
                     with  X  do
       writeln(’ | ’, fam, ’ | ’, name, ’ | ’, fath, ’ | ’,  ...);
              end;
      close(F)
end;
```

Тепер розглянемо процедуру доповнення бази даних новими записами:

```
procedure  append_bd(var  F : file_kadry);
var
       S : string;
       C : char;
       X : kadry;
begin
       write ('Введіть ім'я файлу бази даних -  ');
       readln(S);
       assign(F, S);
       {$I-}    {відключення контролю помилок введення-виведення}
       reset(F);
       if  IOResult <> 0
              then
              begin
                     writeln('Файл з таким іменем відсутній або
                     неможливо відкрити');
                     write('Будемо створювати нову БД? (Y/N) -');
                     if   (C='y')  or (C='Y')
                            then rewrite(F)
                            else halt  {припинити виконання програми}
              end;

       {$I+}     {включення контролю помилок  введення-виведення}
       seek (F, FileSize(F));    {переміститись в кінець файлу}
      repeat
              with  X  do
                     begin
                     write('Прізвище : ');
                     readln(fam);
                     write('Ім''я  : ');
                     readln(name);
                     write('По-батькові : ');
                     readln(fath);
                     ...    {і так далі}
                     end;
              write(F,X);  {запис компоненти у файл}
              write('Чи будете вводити дані ще? (Y/N) - ');
              readln(C);
       until  (C='N') or (C='n');
       close(F)
end;
```

11.3. Текстові файли. Стандартні процедури та функції для роботи з ними

Одним із окремих підвидів файлових типів даних у Turbo Pascal є текстові файли. Зовні цей тип можна порівняти з файлами вказаного типу - файли типу стрічки. Але між ними є суттєві відмінності. Файли стрічок є файлами з прямим доступом до даних, тоді як текстові файли являються файлами з послідовним доступом. Тобто, для того, щоб зчитати N-ну стрічку в текстовому файлі потрібно послідовно зчитати всі попередні N-1 стрічки. Ця властивість є, звичайно, недоліком текстового файлу.

Перевагою текстових файлів є необмеженість довжини стрічок в тексті та їх, нефіксована довжина. Тобто, якщо розглядати файл стрічок, то для кожної з них у файлі відведено однаковий об'єм пам'яті (якщо в описі типу не вказано обмеження довжини стрічки, то кожна з них буде займати по 255 символів). Тоді як у текстових файлах, кожна стрічка буде займати стільки пам'яті, скільки символів реально знаходиться в ній.

Для опису текстових файлів у розділі **type** потрібно вказати наступну конструкцію:

```
type
        ім'я = text;
var
        змінна_файл1, змінна_файл2, ... : ім'я;
```

де

ім'я - назва описаного файлового типу;

змінна_файл1, змінна_файл2, ... - назви змінних, що містять дані текстових файлів.

Наприклад:

```
type
        Txt = text;
var
        T, T1, T2 : txt;
```

Як і для файлів вказаного типу, змінні типу текстових файлів на початку програми потрібно асоціювати з відповідним іменем на диску. Для чого використовується процедура **assign**. Після цього текстовий файл необхідно відкрити.

Існують три способи відкриття текстових файлів:

- процедура **rewrite(*змінна_файл*)** - відкриває файл для запису в нього даних, а маркер розміщується на початок файлу. При цьому знищуються всі дані, що існували у файлі до цього моменту. Дану процедуру використовують для створення нового текстового файлу;
- процедура **reset(*змінна_файл*)** - відкриває файл для зчитування даних із нього. При цьому маркер розміщується на початок файлу. Потрібно відмітити, що як і для файлів вказаного типу, для текстових файлів необхідно, щоб такий файл уже існував (для перевірки використовують директиву **{$I-}** (див. вище));
- процедура **append(*змінна_файл*)** - відкриває текстовий файл для його доповнення новими даними. При цьому маркер установлюється в кінці файлу.

Для зчитування даних з текстових файлів існує дві процедури:

- процедура **read(*змінна_файл, змінна*)** - зчитує з текстового файлу компоненти посимвольно. Наприклад, процедура read(T, C) зчитує з текстового файлу T активний символ і присвоює його значення змінній C (символьного типу). Після чого маркер переводиться на наступний символ;
- процедура **readln(*змінна_файл, змінна*)** - зчитує з текстового файлу компоненти построчково. Наприклад, процедура readln(T, S) зчитує з текстового файлу T активну стрічку і присвоює її значення змінній S. Після чого маркер переводиться на наступну стрічку.

Запис даних у текстові файли також можна здійснювати двома способами:

- процедура **write(*змінна_файл, змінна*)** - записує в текстовий файл компоненти посимвольно. Наприклад, процедура write(T, C) записує в тестовий файл T значення змінної C (символьного типу) і маркер у файлі залишає в тій ж самій стрічці;
- процедура **writeln(*змінна_файл, змінна*)** - записує в текстовий файл компоненти построчково. Наприклад, процедура writeln(T, S) записує в тестовий файл T значення змінної S і маркер у файлі переводить на нову стрічку.

При роботі із текстовими файлами можна використовувати більшість процедур та функцій, що використовуються для обробки файлів вказаного типу (див. розділ 11.2). До них відносяться процедури **rename(*змінна_файл, ім'я_файлу*)**, **erase(*змінна_файл*)**, **Truncate(*змінна_файл*)** та функції **eof(*змінна_файл*)**, **FileSize(*змінна_файл*)**, **FilePos(*змінна_файл*)**, **IOResult** (не можливо лише використати процедуру **seek**).

Крім цього для текстових файлів можна використати ще такі процедури та функції:

- **flush(*змінна_файл*)** - процедура, що очищає буфер введення-виведення текстового файлу;
- **SetTextBuf(*змінна_файл, номер, розмір*)** - встановлює буфер введення-виведення текстового файлу. Параметр ***номер*** встановлює номер байта в пам'яті, з якого буде починатись буфер введення-виведення, а параметр ***розмір*** - вказує розмір цього буфера (цей параметр є необов'язковим);
- **seekeof(*змінна_файл*)** - функція, аналогічна функції **eof(*змінна_файл*)**;
- **eoln(*змінна_файл*)** - функція, що дозволяє визначити чи знаходиться маркер у файлі в кінці стрічки. При цьому, якщо маркер знаходиться в кінці стрічки, то функція видасть результат **TRUE**. В іншому випадку функція видає результат **FALSE**;
- **seekeoln(*змінна_файл*)** - функція, аналогічна функції **eoln(*змінна_файл*)**.

Процес введення даних в текстові файли та зчитування даних із них нічим не відрізняється від аналогічних операцій для файлів вказаного типу.

Наприклад, розглянемо процедуру доповнення до текстового файлу нової інформації. При цьому дані будемо вводити до тих пір, поки не буде введено стрічку із трьох пробілів.

```
procedure  append_file(var  T : text);
var
        S : string;
        C : char;
begin
        write (’Введіть ім’я створюваного файлу  -  ’);
        readln(S);
        assign(T, S);
        {$I-}  {відключення контролю помилок введення-виведення}
        append(T);
        if  IOResult <> 0 then
                begin
                write(’Файл з таким іменем відсутній або не можливо відкрити’);
                write(’Будемо створювати нову БД? (Y/N) –’);
                if   (C=’y’)  or (C=’Y’)
                        then rewrite(T)
                        else halt    {припинити виконання програми}
                end;
        {$I+}  {включення контролю помилок  введення-виведення}
        write(’Введіть наступну стрічку файлу  S=’);
        readln(S);
        while  S <> ’    ’  do
        begin
                writeln(T, S);  {запис у файл стрічки}
                write(’Введіть наступну стрічку файлу  S=’);
                readln(S)
        end;
        close(F)
end;
```

В якості прикладу розглянемо процедуру, яка здійснює знищення з текстового файлу порожніх рядків. Для розв’язку цієї задачі введемо проміжну змінну файлового типу (тимчасовий файл), куди будемо поміщати зчитані з вхідного файлу не порожні стрічки. Якщо стрічка порожня, то її не будемо поміщати в тимчасовий файл.

```
pocedure  delete_space(var T : text);
var
        S, S1 : string;
        T1 : text;
begin
        write (’Введіть ім’я потрібного файлу  -  ’);
        readln(S);
        assign(T, S);
        {$I-}  {відключення контролю помилок введення-виведення}
        reset(T);
        if  IOResult <> 0
                then begin
                write(’Файл з таким іменем відсутній або неможливо відкрити’);
                halt
                end;
        {$I+}  {включення контролю помилок  введення-виведення}
        assign(T1,’temp0001.tmp’);
        rewrite(T1);
        while  not  eof(T) do
                begin
                        readln(T,S1);
                        if  S1<>’ ’ then  writeln(T1,S1)
                end;
```

```
        close(T);
        close(T1);
        erase(T);   {знищуємо набір даних вхідного файлу}
        rename(T1,S);   {надаємо тимчасовому файлу ім'я вхідного оригіналу}
        erase('temp0001.tmp')  {знищуємо з диску тимчасовий файл}
end;
```

11.4. Файли без типу. Стандартні процедури та функції для роботи з ними

Особливим типом файлів, що використовуються в Turbo Pascal є файли без типу або двійкові файли. В цих файлах дані не можна віднести ні до одного зі стандартних типів даних, оскільки вони являються звичайними двійковими кодами.

Для опису файлів без типу в розділі **type** потрібно вказати таку конструкцію:

```
type
        ім'я = file;
var
        змінна_файл1, змінна_файл2, ... : ім'я;
```

де ***ім'я*** - назва описаного файлового типу;

змінна_файл1, змінна_файл2, ... - назви змінних, що позначають файли без типу.

Наприклад:

```
type
        binary = flle;
var
        B, B1, B2 : binary;
```

Як і для типів файлів описаних в розділах 11.2 та 11.3, для обробки Pascal-програмою файлів без типу спочатку потрібно процедурою **assign** співставити змінні, що відповідають цим файлам, з іменами відповідних файлів на диску. Після цього файли потрібно відкрити. Для цього використовуються процедури:

- **rewrite(*змінна_файл, розмір*)** - відкриває файл для запису в нього даних. При цьому маркер розміщується на початку файлу. Як правило, ця процедура використовується для створення файлу. Параметр ***розмір*** вказує розмір блоку даних, що повинні одночасно передаватись з пам'яті у файл. По замовчуванню встановлюється 128 байт;
- **reset(*змінна_файл, розмір*)** - відкриває файл для зчитування та запису в нього даних. При цьому маркер розміщується на початку файлу. Параметр ***розмір*** вказує розмір блоку даних, що повинні одночасно транспортуватись між пам'яттю та файлом. По замовчуванню встановлюється 128 байт.

Для зчитування з файлу без типу одного або декількох не стандартних записів в оперативну пам'ять, використовується процедура:

BlockRead(*змінна_файл, номер , кільк, [результ]*);

де ***змінна_файл*** - змінна, що позначає файл, з якого зчитуються дані. Даний параметр виступає, як параметр-змінна;

номер - змінна типу Word, що вказує номер байта, з якого починається зчитування даних з файлу. Даний параметр виступає, як параметр-змінна;

кільк - змінна типу Word, що вказує максимальну кількість байтів, що потрібно зчитати. Даний параметр виступає, як параметр-значення;

результ - змінна типу Word, що вказує фактичну кількість байтів, що зчиталась з файлу (виступає як вихідний параметр). Вказання цього параметра є необов'язковим. Параметр ***результ*** виступає, як параметр-змінна.

Цілий блок, що переданий в пам'ять займає розмір, який визначається за формулою ***кільк*розмір_запису***, де ***розмір_запису*** - розмір запису в момент коли файл був відкритий. Якщо розмір блоку перевищує 65 535 байт, то в програмі виникає помилка.

Для запису даних з оперативної пам'яті у файл без типу використовується процедура:

BlockWrite(*змінна_файл, номер , кільк, [результ]*);

де призначення всіх параметрів аналогічне відповідним параметрам процедури **BlockRead**.

Для обробки файлів без типу можна використовувати цілий ряд процедур та функцій, що використовуються і для файлів вказаного типу. Серед них процедури **rename(*змінна_файл,ім'я_файлу*)**, **erase(*змінна_файл*)**, **Truncate(*змінна_файл*)** та функції **eof(*змінна_файл*)**, **FileSize(*змінна_файл*)**, **FilePos(*змінна_файл*)**, **IOResult**.

Для прикладу обробки файлів без типу розглянемо процедуру, що здійснює копіювання вмісту одного двійкового файлу в інший.

```
procedure  CopyFile(FromF, ToF : file);
var
       NR, NW : Word;
       Buf  : array[1..2048] of Char;
       S1, S2  : string[12];
begin
       write('Введіть ім'я файлу оригінала  -  ');
       readln(S1);
       write('Введіть ім'я файлу копії  -  ');
       readln(S2);
       assign(FromF, S1);
       {$I-}  {відключення контролю помилок введення-виведення}
       reset(FromF, 1);  { розмір блоку  = 1 }
       if  IOResult <> 0 then
              begin
              write('Файл з таким іменем відсутній або неможливо відкрити');
              halt
              end;
       {$I+}  {включення контролю помилок  введення-виведення}
       assign(ToF, S2);
       rewrite(ToF, 1);  { Record size = 1 }
       repeat
              BlockRead(FromF, Buf, SizeOf(Buf), NR);
       {функція SizeOf встановлює кількість байт, що зайняті аргументом (Buf)}
              BlockWrite(ToF, Buf, NR, NW);
       until (NR = 0) or (NW <> NR);
       close(FromF);
       close(ToF);
end;
```

Розділ 12. Вказівниковий тип даних. Робота з динамічними змінними

В Turbo Pascal існує набір засобів, які призначені для роботи з адресами даних, що знаходяться в пам'яті комп'ютера. Ці засоби називають **вказівниковим типом даних**.

Вказівник - це змінна цілого типу, в якій зберігається адреса байта пам'яті, що містить відповідний елементу даних. Цим елементом може бути змінна, константа, адреса іншої змінної.

Переважно, коли користувач працює з деякою програмою, він не цікавиться розміщенням у пам'яті змінних та констант. Він просто може звертатись до них по їх іменах (тобто ідентифікаторах). При цьому Turbo Pascal сам визначає, де потрібно шукати ці змінні та константи. Для прикладу, програма містить в розділі опису змінних конструкцію:

```
var
      N : integer;
```

Даний запис вказує компілятору на необхідність зарезервувати ділянку в пам'яті, до якої ми будемо звертатись по імені N.

Адресу, за якою розміщена змінна N у пам'яті, можна визначити з допомогою спеціального оператора **@** або функції **addr(*змінна*)**. Тобто потрібно записати вираз:

```
P1 := @N;          або          P1 := Addr(N);.
```

В результаті виконання такої операції у змінну P1 буде записано адресу змінної N.

Пам'ять в ПК адресується шіснадцятковими числами в форматі **BA:BS**, де BA - **сегментна адреса** та BS - **зміщення**. **Сегмент** - це ділянка пам'яті довжиною 64K, яка починається з фізичної адреси, значення якої кратне 16. **Зміщення** - це число, яке визначає номер байту в сегменті, до якого необхідно звернутись.

Вказівки використовують спеціальну ділянку пам'яті, яку називають динамічною.

Динамічна ділянку пам'яті (Heap-пам'ять) - це оперативна пам'ять комп'ютера, що надається програмі поза пам'яттю, яку займає саме тіло програми, сегмент даних та стек.

Всю динамічну пам'ять можна розглядати як суцільний масив, що складається з неперервної послідовності байтів, який ще називають **купою** (**Heap-ділянку**). Heap-ділянка розміщається в пам'яті ПК відразу за ділянкою пам'яті, яку займає тіло програми.

Нижня межа Heap-ділянки визначається стандартною змінною-вказівником **HeapOrg**, яка містить абсолютну адресу початку динамічної пам'яті. Верхня межа Heap-ділянки визначається змінною-вказівником **HeapEnd**. Поточне значення вказівника, що розділяє зайняту та незайняту частини Heap-ділянки містить стандартна змінна-вказівник **HeapPtr**. При кожному новому виділені пам'яті система керування Heap-ділянкою переміщає вказівник HeapPtr вверх (у сторону збільшення адрес пам'яті). При звільненні розподілених у динамічній пам'яті змінних відбувається зворотна дія.

Розмір динамічної пам'яті можна регулювати з допомогою директиви компілятора:

```
{$M стек, динам}
```

де

стек - розмір, що відводиться для стеку пам'яті. В якості даного параметра повинно бути число цілого типу в межах від 1024 до 65520;

динам - розмір, що відводиться для динамічної ділянки пам'яті. В якості даного параметра повинно бути число цілого типу в межах від 0 до 65536, для реального режиму MS-DOS, та від 0 до 65520, для захищеного режиму Windows.

При роботі з динамічною пам'яттю, крім Heap-ділянки, виділяється ще одна ділянка пам'яті, що розміщена в самих верхніх адресах DOS. Адреса її нижньої межі зберігається в змінній-вказівнику **FreePtr**. Ця ділянка містить список записів, які реєструють наявність вільної пам'яті в Heap-ділянці. З її допомогою відбувається повне керування розподілом динамічної пам'яті. Кожен запис цього списку містить всю інформацію про розміщення відповідних динамічних змінних у пам'яті.

Інколи в процесі написання програми виникають ситуації, коли без використання вказівок обійтись не можливо. Розглянемо деякі з таких випадків:

1. Дана програма повинна працювати з великими об'ємами даних (загальний об'єм яких перевищує 64K). Справа в тому, що об'єм сегменту даних не може перевищувати 64K, оскільки при компіляції програми буде виведено повідомлення про помилку: **Out of memory** (Недостатньо пам'яті).
2. Дана програма під час компіляції використовує дані, для яких наперед не відомий об'єм пам'яті для їх зберігання. Для прикладу можна розглянути тип даних стрічка. Як відомо, для стрічок в Pascal відводиться жорстко визначена довжина (указана при описі стрічки). Але при введені реальних стрічок, їх довжина може бути значно меншою. Отже, не раціонально використовується зарезервована для стрічки пам'ять. Та якщо цю змінну розмістити в динамічну ділянку пам'яті, то для стрічки буде виділено стільки байт пам'яті, скільки буде вимагати реальна стрічка.
3. У програмі використовується буфер пам'яті для тимчасового зберігання даних. Якщо в процесі роботи програми необхідно тимчасово виділити пам'ять для зберігання певних даних і при цьому програміст не бажає зберігати ці дані на весь час роботи програми, то в цьому випадку не обійтись без вказівників та динамічних змінних. **Динамічні змінні** створюються (відводиться для них пам'ять) у процесі роботи програми і їх можна знищувати (очищати пам'ять) ще до завершення роботи програми. Над динамічними змінними не можна виконувати операцій введення та виведення. Доступ до них здійснюється через вказівки.
4. Дана програма працює з декількома типами даних. Однією із причин використання вказівок у програмі є необхідність використання посилання на складні типи даних (масиви, множини), які мають різну структуру.
5. Дана програма використовує динамічні структури даних, такі як лінійні списки, стеки, черги, двійкові дерева (див розділ 13).

Будь-який тип даних, пов'язаний з посиланнями, визначає множину значень, які є вказівниками на значення деякого іншого типу даних. В Turbo Pascal вказівники можуть зв'язуватись з деяким типом даних (**типізовані вказівники**) або не зв'язуватись (**нетипізовані вказівники**).

Розглянемо спочатку типізовані вказівники. Для оголошення типізованих вказівників як правило використовується символ ^, який розміщується безпосередньо перед відповідним типом даних, наприклад:

```
type
        pntr = ^ integer;
var
        P, K,  … : pntr;
```

або можна просто записати:

```
        var
                P, K,  … : ^integer;
```

В даному випадку описано вказівку на змінну цілого типу.

Нагадаємо, що в Turbo Pascal існує правило, згідно з яким ідентифікатор повинен бути описаний в програмі до того, як він уперше використовується. Але це правило не поширюється на вказівники, вони можуть посилатись на ще не оголошений тип даних. Це дозволяє організувати зберігання даних у вигляді списку. При цьому кожен елемент списку містити вказівки на наступний і попередній елементи (див. розділ 13), що дозволяє легко знаходити потрібні дані.

В Turbo Pascal 7.0 можна оголошувати вказівки і при цьому не зв'язувати їх з будь-яким конкретним типом даних. Для цієї мети використовується стандартний тип даних - **POINTER**, тобто **нетипізовані вказівки**.

Для опису нетипізованих вказівок потрібно записати конструкцію:

```
        var
                змінна1, змінна2, ... : pointer;
```

Оскільки нетипізовані вказівки не зв'язані з конкретним типом даних, то їх дуже зручно використовувати для динамічного розміщення даних, структура й тип яких змінюється в процесі виконання програми.

Для вказівників у Turbo Pascal доступні лише операції присвоєння та порівняння. Вказівнику можна присвоїти вміст іншого вказівника того ж типу, константу **NIL** (порожній вказівник) або адресу об'єкту, яка визначена функцією **addr** (або оператором **@**), а також адресу, визначену функцією **ptr**. Функція **Ptr(*сегм, зміщ*)** перетворює окремо задані значення адреси сегменту (***сегм*** - число типу Word) та зміщення (***зміщ*** - число типу Word) до типу **pointer**.

Наприклад, якщо оголошено наступні вказівники:

```
var
        Pntr1, Pntr2 : ^integer;
        Pntr3 : ^real;
        Pntr : pointer;
```

то присвоєння типу

```
Pntr1 := Nil;   Pntr1 := Pntr2;
Pntr2 := @X;    Pntr2 := Ptr(35, 00);
```

цілком допустимі, а присвоєння Pntr1 := Pntr3 заборонене, оскільки вказівки Pntr1 та Pntr3 вказують на різні типи даних. Але в тексті програми можуть зустрічатись присвоєння типу: Pntr := Pntr1 або Pntr2 := Pntr, оскільки Pntr є нетипізованим вказівником.

Для отримання доступу до значення, що зберігається в пам'яті за адресою, на яку посилається вказівник, необхідно після ідентифікатора вказівника помістити символ ^. Наприклад, запис Pntr1^ := 7 означає, що в комірку пам'яті, на яку вказує вказівник Pntr1 буде записано число 7.

Запис Pntr := ^^N означає, що вказівнику Pntr присвоюється адреса комірки пам'яті, де зберігається вказівник на комірку в, якій знаходиться змінна N.

В Turbo Pascal 7.0 існує два основних методи роботи з динамічною пам'яттю:

- з допомогою процедур **New** та **Dispose**;
- з допомогою процедур **GetMem** та **FreeMem**.

Процедура **New(*P*)**, де ***P*** - змінна типу **pointer**, створює нову динамічну змінну того типу на який посилається вказівник. При цьому встановлюється значення змінної ***P*** таким чином, щоб воно вказувало на цю нову динамічну змінну.

Наприклад:

```
var
        P1 : ^integer;
        P2 : ^string;
begin
        new(P1);
        new(P2);
          …
end.
```

Так, в даному прикладі після виконання процедури new(P1) вказівник P1 отримає значення, що рівне тому, яке досі мав вказівник **HeapPtr** (див. вище), а значення вказівника **HeapPtr** буде збільшене на 2, оскільки довжина даних типу integer, з якими зв'язаний вказівник P1, становить 2 байти. Процедура new(P2) викликає виділення блоку пам'яті довжиною 256 байтів і зміщення вказівника **HeapPtr** на цю ж величину.

Якщо вказівник посилається на тип даних, для якого потрібно більше пам'яті, ніж доступно в Heap-ділянці, то в цьому випадку виникає так звана помилка виконання програми.

Після того, як вказівник отримав значення, тобто почав указувати на відповідний фізичний байт пам'яті, по його адресі можна розмістити будь-яке значення з відповідним типом даних. Для цього, як уже відмічалось, необхідно відразу за вказівником помістити символ **^**.

```
var
        P1, P2 : ^integer;
     …
begin
        new(P1);
        new(P2);
        P1^ := 5;
        P2^ := 5*3;
          …
end.
```

Таким чином, значення будь-якого вказівника є адресою, а щоб указати, що мова йде не про адресу, а про значення змінної, що знаходиться по цій адресі, після вказівника необхідно ставити символ ^ (так як це показано у вище приведеному прикладі).

Динамічний розподіл даних можна використовувати в будь-якому місці програми, при цьому над їх значеннями дозволяється виконувати будь-які операції, що дозволені над звичайними змінними та константами, наприклад:

```
P1^ := 2*P2^+(P2^+7)*4;
```

Але необхідно пам'ятати, що вказівнику не можна присвоїти значення динамічної змінної, а значення динамічної змінної не можна присвоїти вказівнику. Отже, не можна вказувати наступні записи:

```
P1 := 3*P2^;
P1^ := P2;
```

Для того, щоб звільнити динамічну пам'ять від динамічної змінних можна скористатись процедурою **dispose(*P*)**, де ***P*** - змінна типу **pointer** (при цьому буде знищено динамічну змінну ***P*** та очищену ділянку пам'яті, в якій ця змінна знаходилась). Потрібно відмітити, що процедура **dispose** не змінює значення вказівника ***P***, а лише повертає в купу пам'яті (Heap-пам'ять) ту пам'ять, що раніше була зв'язана з цим вказівником. Але необхідно пам'ятати, що повторне використання процедури до вільного вказівника може викликати виникнення помилки під час виконання програми. Щоб запобігти такій помилці, звільнений вказівник необхідно помітити зарезервованим словом **Nil** (порожній). Наприклад:

```
...
dispose(P);
P := Nil;
...
```

Необхідно також відмітити, що початкове значення вказівника при його оголошені в розділі змінних може бути довільним. Тому використання вказівника, якому не було присвоєно відповідне початкове значення з допомогою процедури **new** або будь-яким іншим способом, може привести до непередбачуваних наслідків. Отже, після створення вказівників (перед їх використанням), яким не присвоєно конкретного значення, їм необхідно присвоїти значення **Nil** (порожнє).

В Turbo Pascal 7.0 існує можливість звільнення цілого блоку динамічно розподіленої пам'яті-купи. Цю операцію можна виконати з допомогою процедур **Mark** та **Relase**. Процедура **Mark(*P*)**, де ***P*** - змінна типу **pointer**, запам'ятовує поточну „вершину" Heap-ділянки (тобто значення вказівника **HeapPtr**) у вказівнику ***P*** для наступного звільнення блоків пам'яті, які розміщені вище даного вказівника. Процедура **Relase(*P*)**, де ***P*** - змінна типу **pointer**, звільнює пам'ять, що зайнята блоками, які знаходяться вище вказівника ***P***.

Процедура **GetMem(*P, Розмір*)** (де ***P*** - змінна типу **pointer**, а ***Розмір*** - змінна типу **word**) виділяє з Heap-ділянки блок пам'яті, об'ємом ***Розмір*** байтів, при цьому адреса початку даного блоку присвоюється вказівнику ***P***. Максимальний об'єм блоку, що може бути виділений таким чином не повинен перевищувати 65521 байт. Як і при використанні процедури **new**, розмір блоку пам'яті в процедурі **GetMem** вказується в байтах.

Процедура **FreeMem(*P, Розмір*)** (де ***P*** - змінна типу **pointer**, а ***Розмір*** - змінна типу **word**) знищує (очищає) блок пам'яті, об'ємом ***Розмір*** байтів, який адресований вказівником ***P***. Якщо в програмі використовується метод розподілу пам'яті з допомогою процедур **GetMem** та **FreeMem**, то виклики цих процедур повинні відповідати одна одній, а значення ***Розмір*** при звертанні до однієї і тієї ж змінної-вказівника повинні співпадати.

При розподілі динамічної пам'яті в Heap-ділянці можливе виникнення помилкових ситуацій. Існує декілька шляхів їх усунення.

Один із найпоширеніших шляхів - перед тим як використовувати стандартні процедури розподілу динамічної пам'яті, необхідно перевірити наявність необхідних об'ємів пам'яті в Heap-ділянці, наприклад, наступним чином:

```
function  check(var P : pointer; Size : word) : boolean;
begin
        Check : False;
        if Size > MaxAvail then Exit;
        Check : = True;
        GetMem(P, Size)
end;
```

Дана функція встановлює значення True, якщо об'єм пам'яті менший, ніж розмір найбільшої безперервної ділянки пам'яті в Heap-ділянці або дорівнює її (тобто виділення пам'яті пройшло успішно). Значення False встановлюється в тому випадку, коли розмір пам'яті більший, ніж розмір найбільшої безперервної ділянки пам'яті в Heap-ділянці. В даному прикладі використовувалась функція MaxAvial, яка встановлює розмір у байтах найбільшої безперервної ділянки пам'яті в Heap-ділянці.

Розглянемо основні процедури та функції, що використовуються в Turbo Pascal 7.0 для роботи з динамічною пам'яттю:

- функція **Add(*X*)** - установлює результат типу pointer, в якому міститься адреса значення змінної ***X*** в пам'яті (аналогічно ***@X***);
- функція **Cseg** (результат типу **Word**) - визначає поточне значення регістру Cs мікропроцесора. Коли програма розпочинає роботу в регістрі Cs міститься адреса початку сегмента коду програми;
- процедура **Dispose(*P*)** - знищує динамічно розподілену змінну, на яку вказує вказівник ***P*** (див. вище);
- функція **Dseg** (результат типу **Word**) - визначає поточне значення регістру Ds мікропроцесора. Коли програма розпочинає роботу в регістрі Ds міститься адреса початку сегмента даних програми;
- процедура **FreeMem(*P, Розмір*)** - звільняє блок пам'яті, розміром ***Розмір***, який адресований вказівником ***P*** (див. вище);
- процедура **GetMem(*P, Розмір*)** - виділяє з Heap-ділянки блок пам'яті вказаного розміру. При цьому адреса його початку присвоюється вказівнику ***P*** (див. вище);
- процедура **Mark(*P*)** - запам'ятовує поточне значення вказівника Heap-ділянки, тобто значення **HeapPtr**, у вказівнику P для наступного звільнення блоків пам'яті, що розміщені вище цього вказівника, з допомогою процедури **Relase** (див. вище);
- функція **MaxAvail** (результат типу **LongInt**) - встановлює розмір у байтах найбільшої безперервної ділянки пам'яті в Heap-ділянці (див. вище);
- функція **MemAvail** (результат типу **LongInt**) - встановлює розмір у байтах загального вільного простору в Heap-ділянці;
- процедура **New(*P*)** - резервує фрагмент Heap-ділянки для розміщення нової динамічної змінної (див. вище);

- функція **Ofs(*X*)** (результат типу **Word**) - установлює значення, яке дорівнює зміщенню адреси змінної, константи, процедури, функції і т.д. Аргумент ***X*** може мати довільний тип даних;
- функція **Ptr (*Сегмент, зміщення*)** - перетворює окремо задані значення сегменту та зміщення в значення типу **pointer** (див. вище);
- процедура **Relase(*P*)** - звільнює пам'ять, яка зайнята блоками, що розміщені вище вказівника ***P*** (див. вище);
- функція **Seg(*X*)** (результат типу **Word**) - встановлює значення, яке рівне сегменту адреси змінної, константи, процедури, функції і т.д. Аргумент ***X*** може мати довільний тип даних;
- функція **SizeOf(*X*)** (результат типу **Integer**) - установлює об'єм основної пам'яті, яку займає вказана змінна або тип даних, в байтах. Аргумент ***X*** може мати довільний тип даних.

Розділ 13. Тип даних - списки

13.1. Лінійні списки

Список - набір зв'язаних між собою компонентів, кожна з яких являє собою запис, що містить принаймні два поля: одне поле типу вказівник, що вказує на наступний запис, а друге для розміщення даних.

У загальному випадку запис може містити не один, а декілька вказівників і декілька полів даних, тобто лінійні й нелінійні списки. **Лінійні списки** - це списки, в яких кожен компонент зв'язаний з наступним одним вказівником. **Нелінійні списки** - це списки, в яких кожен елемент може мати вказівки на декілька інших компонентів. Поле даних може бути змінною, масивом, множиною або записом.

Лінійні списки бувають:

- **однонаправлені** - списки, в яких кожен компонент списку має вказівник лише на наступний елемент;
- **двонаправлені** - списки, в яких кожен компонент списку має вказівник на наступний та на попередній компонент;
- **циклічні** - списки, в яких останній компонент списку має вказівник на самий перший.

Для опису лінійних однонаправлених списків у розділі **type** потрібно вказати таку конструкцію

```
type
        Link = ^Node;
        Node = record
        Inf : тип даних;
        Next : Link;
        end;
var
        P, Top, … : Link;
```

а для опису двонаправлених списків потрібно вказати:

```
type
        LinkRev = ^Node;
        Node = record
        Inf : тип даних;
        Next, Prev : LinkRev;
        end;
var
        P, Top, … : LinkRev;
```

В поле **Inf** записується значення потрібного компоненту списку (***тип даних*** - тип поля даних цього компоненту). Як правило, змінною **Top** позначають перший елемент списку. Поле вказівник **Next** вказує на наступний компонент списку, а **Prev** - на попередній.

Розглянемо процедуру створення лінійного однонаправленого списку на прикладі списку цілих чисел:

```
procedure MkLink(var P : Link);
var
        Top : Link;
        X : integer;
begin
        New (Top);
        Top := Nil;  {вершині списку присвоюємо порожній вказівник}
        while  true  do
                begin
                New (P); {створюємо новий компонент списку}
                write ('X=');
                readln (X);
                if X=9999 then exit;  {умова поки вводити елементи списку}
                P^.Inf:=X;   {заносимо введене значення в список}
                {встановлюємо вказівник на попередній компонент}
                P^.Next := Top;
                Top := P;  {Переходимо до створення наступного компоненту}
        end
end;
```

Тепер опишемо процедуру створення лінійного двонаправленого списку на прикладі списку символів:

```
procedure MkLinkRev(var P : LinkRev);
var
        Top, Bot : LinkRev;
        X : char;
begin
        New (Top);
        Top := Nil;  {вершині списку присвоюємо порожній вказівник}
        New (P); {створюємо новий компонент списку}
        write ('X=');
        readln (X);
        P^.Inf:=X; {заносимо введене значення в список}
        while  true  do
                begin
                New (Bot); {створюємо новий компонент списку}
                write ('X=');
                readln (X);
                if X=9999 then exit;  {умова поки вводити елементи списку}
                Bot^.Inf:=X; {заносимо введене значення в список}
                {встановлюємо вказівник на попередній компонент}
                P^.Next := Top;
                {встановлюємо вказівник на наступний компонент}
                P^.Prev := Bot;
                Top := P; {Переходимо до створення наступного компоненту}
                P  := Bot; {Переходимо до створення наступного компоненту}
                end
end;
```

В якості прикладу розглянемо ще процедуру перегляду лінійного однонаправленого списку:

```
procedure PrintLink (var P : Link);
begin
        while  P<>Nil do  {поки список не закінчиться}
                begin
                        writeln (P^.inf);
                        P^.Next:=P;
                end
end;
```

При виконанні операцій над лінійними списками часто необхідно здійснити пошук компоненту з потрібним значенням (наприклад, символ із значенням змінної X). Для цього можна використати наступну процедуру:

```
procedure Find(X: char; var  Top, P: Link);
        begin
                P := Top;
                while (P <> NIL) and (X <> P^.Inf) do P := P^.Next;
        end;
```

Тепер розглянемо процедуру, яка здійснює вставку нової компоненти після знайденої попередньою процедурою:

```
procedure InsComp(var Top, P : Link);
var
        Aux : Link;
        S : Char;
begin
        Find(X, Top, P);
        New(Aux);
        write (Введіть значення нової компоненти  = ');
        readln (S);
        Aux^.Inf := S;
        Aux^.Next := P^.Next;
        P^.Next:=Aux
end;
```

Для того, щоб добавити новий компонент перед знайденим у вищеописаній процедурі потрібно внести такі зміни:

```
...
New (Aux);
write (Введіть значення нової компоненти  = ');
readln (S);
Aux^.Inf:=S;
Aux^:=P^
Aux^,Next := P^.Next;
Aux := P;
...
```

Для знищення знайденого компоненту потрібно скористатись процедурою:

```
procedure DelComp (var X : char;  var Top, P : Link);
var
        Ppre : Link;   {проміжна змінна – попередній компонент}
begin
        P := Top;
        {здійснимо пошук компоненти}
        while (P <> NIL) and (X <> P^.Inf) do
                begin
                        {попередньому компоненту присвоюємо поточний компонент}
                        Ppre := P;
                        {переходимо до наступного компоненту}
                        P := P^.Next
                end;
        Ppre^.Next := P^.Next
end;
```

Якщо потрібно знищити компоненту, яка слідує після знайденої процедурою Find, то потрібно ввести конструкцію:

```
P^.Next := P^.Next.Next;
```

13.2. Стеки та черги

Лінійні списки інколи організовують у структури, що називаються стеками (stack) та чергами (turn).

Стек (Stack) - це лінійний список, який має одну точку доступу, через яку проходить і запис і зчитування його компонентів. Тобто він працює за принципом

LIFO (Last-In, First-Out)

що можна перекласти, як ***„перший зайшов - останній вийшов“***.

Точку доступу до компонентів стеку називають **вершиною стеку**.

Для опису стеків потрібно в розділі **type** вказати наступну конструкцію:

```
type
    Stack = ^Node
    Node = record
    Inf   :  тип даних;
    Next : Stack
end;
var
    P, Top, … : Link;
```

де **Top** - вказівник вершини стеку.

Над стеками переважно можна виконувати чотири операції:

- початкове формування стека (запис першого компоненту);
- додавання компоненту в стек;
- перегляд елементів стеку;
- вибірка компоненти (видалення).

Процедура створення стеку цілих чисел буде мати вигляд:

```
procedure  CreateStack (var  Top: Stack; var X : integer);
       begin
              New(Top);
              Top^. Next := Nil;
              Top^. Inf := X;
       end;
```

Для того, щоб добавити нові компоненти можна скористатись процедурою:

```
procedure AddStack (var Top: Stack);
var Aux : Stack;
begin
       while  true do
       begin
              New (Aux);
              Aux^. Next := Top;
              Top := Aux;
              write ('X=');
              readln(X);
              if X>=9999 then exit;
              Top^.Inf := X
       end
end;
```

Перегляд компонентів стеку можна здійснити процедурою:

```
procedure PrintStack (var P: Stack);
begin
        while P <> Nil do
        begin
                writeln (P^.Inf);
                P^.Next := P
        end
end;
```

Для знищення компоненти із стека можна використати процедуру:

```
procedure  DelStack (var  Top : Stack; var  X : integer);
begin
        X := Top^.Inf;
        Top := Top^.Next
end;
```

Черга (Turn) - це лінійний список, в якому нові компоненти добавляються в один кінець списку, а зчитуються з іншого. Тобто він працює за принципом

FIFO (First-In, First-Out)

що можна перекласти, як ***„перший зайшов - перший вийшов“***.

Для формування черги і роботи з нею необхідно мати три змінних типу черга, перша з яких визначає початок черги (**Top**), друга - кінець черги (**Bottom**), третя - допоміжна (**P**).

Черги описуються в розділі **type**:

```
type
        Turn = ^Node
        Node = record
        Inf : тип даних;
        Next : Turn
end;
var
        P, Left, Right : Turn;
```

Над чергами виконуються такі ж операції, що й над стеками. Так для створення черги можна скористатись процедурою:

```
procedure  CreateTurn (var  Left, Right : Turn);
var
        X : integer;
begin
        New(Left);
        Left^.Next:=Nil;
        write ('Введіть X=');
        readln(X);
        Left^.Inf := X;
        Right := Left;
end;
```

Для добавлення до черги нових компонентів використовуємо процедуру:

```
procedure  AddTurn (var  P, Left, Right : Turn);
var
        X : integer;
begin
        while  true do
        begin
                New(p);
                write (’Введіть X=’);
                readln(X);
                if x>=9999 then exit;
                P^.Inf := X;
                P^.Next:=Left;
                Left := P;
        end;
end;
```

Перегляд компонентів черги можна здійснити процедурою:

```
procedure PrintTurn (var P, Left  : Turn);
begin
        P := Left;
        while P<> Nil do
        begin
                writeln (P^.inf);
                P^.Next := P
        end
end;
```

Для знищення активного компоненту з черги можна використати процедуру:

```
procedure  DelTurn(var Left : Turn; var X : integer);
begin
        X := Left^.Inf;
        Left := Left^.Next
end;
```

13.3. Нелінійні списки

Нелінійні списки - це списки, в яких кожен компонент зв’язаний з декількома сусідніми. До нелінійних списків відносяться: тексти, графи, двійкові дерева, N-кові дерева.

Текст - це нелінійний список, кожен компонент якого - змінна типу запис, що складається з трьох полів: поле даних, та два поля, що вказують на наступні компоненти (слова). Слова між собою з’єднані в рядки, кожне слово в рядку має вказівку на наступне. Рядки між собою зв’язані по певних словах, найчастіше це перше та останнє слово рядка.

Тексти описуються в розділі type

```
type
        Txt = ^Node
        Node = record
        Inf : тип даних;
        Word : txt;
        Row : txt;
end;
```

Інший тип нелінійних списків - **графи.**

Графи - це нелінійні списки, організовані по принципу: сума декартових координат вузлів і ребер рівні. Графи описуються в розділі type

```
type
        Gr = ^node;
        Node = rekord;
        Int : integer;
        L1, L2, L3 : Gr;
end;
```

Двійкові дерева - це списки, організовані по принципу, що кожен елемент має зв'язок з двома нижче стоячими елементами. Двійкові дерева описуються в розділі type. Для цього вводимо наступну конструкцію:

```
type
        tree = ^Node;
        Node = record;
        Int : integer;
        Left, Right : tree;
end;
```

Крім вищеописаних нелінійних списків є багато інших типів нелінійних списків: N-кове дерево, сітковий список і т.д.

Розділ 14. Модулі в Turbo Pascal

14.1. Типи модулів. Створення модуля користувача

Програми, написані Вами на Turbo Pascal і оформлені у вигляді готових процедур та функцій, можна використовувати в інших програмах. Основна концепція такого підходу полягає у об'єднанні своїх процедур та функцій у власні бібліотек, які пізніше можуть підключатись до нових програм, що Ви розробляєте. Такі готові бібліотеки підпрограм називають **модулями (Unit)**.

Модуль (Unit) - програмна одиниця, текст якої компілюється незалежно (автономно) і складається з набору процедур, функцій, типів даних, констант, які не входять у стандартний Pascal.

Концепцію модулів уперше розроблено програмістом Н. Віртом для мови програмування більш високого рівня Modula-2. Пізніше цей термін і спосіб побудови програм був реалізований в інших мовах.

Потрібно відмітити, що в Turbo Pascal входить цілий набір стандартних модулів, які поставляються разом із програмним пакетом цієї мови програмування. Такі модулі називаються **стандартними модулями**. Крім цього, як відмічалось вище, програміст може створювати власні модулі. Такі модулі називаються **модулями користувача**.

Кожен модуль (як стандартний так модуль користувача) у Turbo Pascal записується в окремому файлі з розширенням TPL чи TPU і поміщаються або в робочий каталог оболонки Turbo Pascal, або в спеціальний каталог для модулів (шлях до цього каталогу вказується в налаштуванні конфігурації оболонки Turbo Pascal).

Для підключення модуля до створюваної програми, його ім'я потрібно вказати в розділі **Uses** на початку програми. При цьому потрібно вказати таку конструкцію:

uses ***модуль1, модуль2, модуль3, ...;***.

До стандартних модулів Turbo Pascal 7.0 належать 10 модулів:

- **System** - містить набір стандартних констант, типів даних та підпрограм, що розглядались нами в попередніх розділах посібника. Він забезпечує виконання низькорівневих програм підтримки таких можливостей як файлове введення-виведення, обробка стрічок, операції з плаваючою комою і т.д. Усі програми в Turbo Pascal автоматично використовують цей модуль, тому його не потрібно підключати в розділі uses, створюваної програми. Даний модуль знаходиться у файлі turbo.tpl;
- **CRT** - містить підпрограми керування екранним режимом, звуком динаміка, текстовими вікнами (використання кольорових вікон), зчитування розширених кодів клавіатури. Він дозволяє писати програми, які направляють виведення на екран, безпосередньо в BIOS або відеопам'ять. Цей модуль можна використовувати лише в програмах, що будуть працювати на ПК фірми IBM або сумісних з ними. Даний модуль знаходиться у файлі turbo.tpl;
- **DOS** - підтримує більшість найбільш часто використовуваних функцій ОС MS-DOS і функцій обробки файлів. Даний модуль знаходиться у файлі turbo.tpl;

- **WinDOS** - подібний до попереднього модуля, але використовується для програм, що будуть працювати в ОС Windows. На відміну від модуля DOS модуль WinDOS використовує особливий тип стрічок - стрічки із завершаючим нулем. Даний модуль знаходиться у файлі windos.tpu;
- **Printer** - дозволяє перенаправляти стандартне виведення Turbo Pascal на принтер, використовуючи процедури **write** та **read**. Даний модуль знаходиться у файлі turbo.tpl;
- **Overlay** - містить процедури, функції та змінні, які використовує програма керування оверлеями в Turbo Pascal. Це дозволяє зменшити об'єм пам'яті, яка потрібна програмам, що виконуються в реальному режимі MS-DOS. Фактично, він дозволяє писати програми, яким потрібно більше пам'яті ніж реально доступно на машині, оскільки під час роботи програми в пам'яті буде працювати лише її частина. Даний модуль знаходиться у файлі turbo.tpl;
- **Strings** - дає можливість програмі використовувати стрічки із завершуючим нулем (null-terminated strings), що разом з розширеним синтаксисом дозволяє писати програми, що сумісні із прикладними програмами для Windows. Даний модуль знаходиться у файлі turbo.tpu;
- **Graph** - містить бібліотеку потужних і швидких підпрограм для роботи із графікою (дисплей у графічному режимі). Модуль містить апаратно незалежні драйвери, що підтримують найбільш поширені адаптери CGA, EGA, VGA, Hercules, AT&T 400, MCGA, 3270 PC та IBM-8514. Даний модуль знаходиться у файлі graph.tpu та graph.tpp;
- **Turbo3** - містить підпрограми для роботи з Pascal 3-ої та попередніх версій. Він містить дві змінні та декілька процедур та функцій, що вже не підтримуються цією мовою програмування. Даний модуль знаходиться у файлі turbo3.tpu;
- **Graph3** - містить набір процедур, що використовувались для роботи із графікою в Pascal 3.0. Даний модуль знаходиться у файлі graph3.tpu.

Крім цього в поставку Turbo Pascal входять також інші модулі:

- **WinCRT** - містить процедури, функції та константи для роботи з текстовими термінальними вікнами у Windows 3.1. Якщо використовувати даний модуль, то програмісту не буде потрібно писати Windows-специфічний програмний код. Даний модуль знаходиться у файлі wincrt.tpu;
- **WinAPI** - містить API функції та процедури для Windows 3.1. Вони використовуються в захищеному режимі роботи програм. Даний модуль знаходиться у файлі tpw.tpl;
- **WinPrn** - дозволяє перенаправляти стандартне виведення інформації на принтер у програмах написаних для Windows 3.1. Даний модуль знаходиться у файлі tpw.tpl;
- **WinProcs** - встановлює функції та заголовки процедур для програм у захищеному API режимі Windows 3.1. Кожна програма, що містить стандартні бібліотеки Windows може бути доступна через WinProcs. Разом з модулем WinTypes, WinProcs визначає реалізацію в Turbo Pascal Windows API. Даний модуль знаходиться у файлі tpw.tpl;

- **WinTypes** - встановлює в Turbo Pascal всі типи, що використовуються у Windows-програмах у режимі API, включаючи прості типи, структури даних (записи) та всі стандартні константи Windows (включаючи стилі, повідомлення і флаги). Даний модуль знаходиться у файлі tpw.tpl;
- **Win31** - забезпечує підтримку для додаткових програм API, що працюють у Windows 3.1. Програми, які використовують модуль Win31 не будуть працювати у Windows 3.0. Даний модуль знаходиться у файлі Win31.tpu.

Поряд із цими стандартними модулями фірма Borland включила в поставку пакету Borland Pascal цілий ряд додаткових модулів, таких як: **APP** (знаходиться у файлах app.tpu, app,tpp та app.tpw), **BWCC** (bwcc.tpw), **ColorSel** (colorsel.tpu, colorsel.tpp), **CustCntl** (custcntl), **Dialogs** (dialogs.tpu, dialogs.tpp), **Drivers** (drivers.tpu, drivers.tpp), **Editors** (editors.tpu, editors.tpp), **HitList** (hitlist.tpu, hitlist.tpp), **Memory** (memory.tpu, memory.tpp), **Menus** (menus.tpu, menus.tpp), **MsgBox** (msgbox.tpu, msgbox.tpp), **Objects** (objects.tpu, objects.tpp, objects.tpw), **Odialogs** (odialogs.tpw), **OMemory** (omemory.tpw), **Oprinter** (oprinter.tpw), **OstDdlgs** (ostddlgs.tpw), **OstWnds** (ostwnds.tpw), **OutLine** (outline.tpu, outline.tpp), **OWindows** (owindows.tpw), **StdDlg** (stddlg.tpu, stddlg.tpp), **TextView** (textview.tpu, textview.tpp), **Validate** (validate.tpu, validate.tpp, validate.tpw), **Views** (views.tpu, views.tpp) та інші. Модулі знаходяться в трьох варіантах (у файлах трьох типів), оскільки вони передбачають різні режими роботи програм: tpu - модулі для реального режиму MS-DOS, tpp - модулі для захищеного режиму MS-DOS та tpw - модулі для режиму Windows.

Розглянемо тепер порядок створення модулів користувача. Сам по собі модуль можна розділити на декілька розділів: заголовок, інтерфейсна частина, реалізаційна частина, ініціалізаційна частина. Отже розглянемо структуру модуля:

```
{заголовок модуля}
unit ім'я модуля;
{$директива компілятора+} {глобальна директива компілятора}
{інтерфейсна частина}
interface       {початок розділу оголошень}
[uses модуль1, модуль2, ....;] {якщо використовуються існуючі модулі}
[label мітка1, мітка2, ...;]      {якщо використовуються мітки}
[const
        ім'я1=значення1;
        ... ]      {якщо використовуються константи}
[type
        ім'я1=опис типу;
        ... ]      {якщо використовуються складні типи}
[var
        змінна1, змінна2,... : тип;
        ... ]      {якщо використовуються глобальні змінні}
[procedure ім'я1(список праметрів);
        procedure ім'я2(список праметрів);
        ... ]
```

```
[function  ім'я1(список праметрів) : тип результату;
        function  ім'я2(список праметрів) : тип результату;
        ... ]
{реалізаційна частина}
implementation
[uses  модуль1, модуль2, ....;]  {якщо використовуються існуючі модулі}
[label  мітка1, мітка2, ...;]      {якщо використовуються мітки}
[const
        ім'я1=значення1;
        ... ]      {якщо використовуються константи}
[type
        ім'я1=опис типу;
        ... ]      {якщо використовуються складні типи}
[var
        змінна1, змінна2,... : тип;
        ... ]      {якщо використовуються глобальні змінні}
[procedure  ім'я1(список праметрів);
                тіло процедури
procedure  ім'я2(список праметрів);
                тіло процедури
                ... ]
[function  ім'я1(список праметрів) : тип результату;
                тіло функції
function  ім'я2(список праметрів) : тип результату;
                тіло функції
                ... ]
{ініціалізаційна частина}
[begin]        {вказувати необов'язково}
end.
```

Розглянемо розділи модуля більш детально. В заголовку модуля потрібно вказати службове слово unit, яке показує, що даний текст програми є модулем, а тоді ім'я створюваного модуля. Потрібно відмітити, що ім'я модуля повинно співпадати з назвою файлу, в якому він записаний. Тут ж знаходяться директиви компілятора, що надають загальні узгодження та установки для цілого модуля.

Інтерфейсна частина описує всі константи типи даних, змінні, процедури та функції, які доступні в цьому модулі для використання зовнішніми програмами. Вона починається службовим словом **interface**. Інтерфейсна частина модуля несе інформацію, яка необхідна для використання процедур та функцій, що створюються в модулі. Крім цього в інтерфейсній частині можна зробити доступними для використання вже існуючі готові модулі, вказавши їх імена після оператора **uses**. Всі процедури та функції, створювані в даному модулі, повинні бути описані в інтерфейсній частині рядком-заголовком з указанням параметрів. Сам текст програми цих процедур та функцій записується в реалізаційній частині модуля.

Реалізаційна частина - це частина, в якій створюються самі процедури та функції. Вона починається зі службового слова **implementation**. Так само, як і для звичайної програми тут можна вказати глобальні (для модуля) змінні, типи даних і константи поряд із створеними процедурами та функціями.

Ініціалізаційна частина модуля представляє собою основний блок модуля. Указані в ній оператори виконуються першими, тобто вони виконуються перед операторами основного блоку головної програми, в яку включений даний модуль. Як правило ця частина модуля порожня і містить лише операторні дужки **begin end**. В цьому випадку оператор **begin** на початку ініціалізаційної частини модуля вказувати необов'язково.

Після створення тексту модуля його потрібно зберегти на диску, а тоді прокомпілювати. Для цього, спочатку, необхідно встановити в пункті меню **Compile** значення команди **Destination** рівним **Disk** (замість **Memory**). Тоді прокомпілювати текст модуля натискуванням комбінації клавіш Alt+F9. В результаті цих дій на диску з'явиться файл цього модуля з розширенням tpu.

Для прикладу, розглянемо створення модуля, що містить тригонометричні функції обчислення тангенса, арксинуса, арккосинуса та переведення кута з градусів у радіани та навпаки.

```
unit  trigono;
interface
function tan(X : real) : real;
function arcsin(X : real) : real;
function arccos(X : real) : real;
function grad(X : real) : real;
function radian(X : real) : real;
implementation
function tan(X : real) : real;
begin
       tan :=  sin(X)/cos(X)
end;
function arcsin(X : real) : real;
begin
       arcsin := arctan(X/sqrt(1 - sqr(X)))
end;
function arccos(X : real) : real;
begin
       arccos := arctan((1 - sqr(X))/X)
end;
function grad(X : real) : real;
begin
       grad :=  X*180/PI
end;
function radian(X : real) : real;
begin
       radian :=  X*PI/180
end;
begin
end.
```

14.2. Модуль CRT

Модуль CRT містить набір підпрограм, що надають можливість програмам, які працюють під DOS створювати кольорові текстові вікна, працювати з розширеними кодами клавіатури та звуковими сигналами. При використанні модуля CRT виведення інформації здійснюється безпосередньо через BIOS або для ще більшого пришвидшення операцій безпосередньо у відеопам'ять.

В модулі CRT міститься декілька констант і змінних, які призначені для керування текстовим екраном, клавіатурою, кольором і т.д. До них відносяться:

- **CheckBreack** - змінна логічного типу, яка дозволяє (значення TRUE) або забороняє (FALSE) використання комбінації клавіш Ctrl+Breack для виходу із програми;
- **CheckEOF** - змінна логічного типу, яка дозволяє (TRUE) або забороняє (FALSE) використання мітчика кінця файлу;
  CheckSnow - змінна логічного типу, яка дозволяє (TRUE) або забороняє (FALSE) перевірку екрану на „сніг";
- **DirectVideo** - змінна логічного типу, яка дозволяє (TRUE) або забороняє (FALSE) прямий доступ до відеопам'яті при використанні процедур **write** та **writeln**;
- **LastMode** - змінна цілого типу **word**, яка при кожному виклику процедури **TextMode** зберігає значення поточного відеорежиму;
- **TextAttr** - змінна цілого типу **byte**, яка містить значення атрибуту кольору тексту у вікні;
- **WindMin** -змінна цілого типу **word**, яка містить координати верхнього лівого кута поточного вікна;
- **WindMax** - змінна цілого типу **word**, яка містить координати нижнього правого кута поточного вікна;
- **BW40** - константа цілого типу, що має значення 0. Вона встановлює текстовий монохромний режим екрану 40x25 символів;
- **CO40** або **C40** - константа цілого типу, що має значення 1. Вона встановлює текстовий кольоровий режим екрану 40x25 символів;
- **BW80** - константа цілого типу, що має значення 2. Вона встановлює текстовий монохромний режим екрану 80x25 символів;
- **CO80** або **C80** - константа цілого типу, що має значення 3. Вона встановлює текстовий кольоровий режим екрану 80x25 символів;
- **Mono** - константа цілого типу, що має значення 7. Вона встановлює текстовий монохромний режим екрану 80x25 символів для адаптерів MDA та Hercules;
- **Font8x8** - константа цілого типу, що має значення 256. Вона встановлює текстовий кольоровий режим екрану 80x43 символи для адаптерів EGA та 80x50 символів для адаптерів VGA. Ця константа не являється самостійною, а записується разом з іншими константами режимів. Наприклад, „C80+Font8x8".

При роботі із текстовими вікнами спочатку бажано встановити режиму роботи екрану. Для цього використовується процедура **TextMode(*режим*)**, де ***режим*** - константа, що описує режим роботи екрану. Наприклад,: TextMode(C80+Font8x8).

Крім зміни текстового режиму екрану, процедура **TextMode** виконує ряд додаткових функцій:

- поточне вікно встановлює рівним цілому екрану;
- встановлює режим прямого запису у відеопам'ять. Змінній **DirectVideo** присвоюється значення TRUE;
- відключається режим контролю „снігу" (змінна **CheckSnow** стає рівною FALSE);
- установлюється режим нормальної яскравості символів;
- здійснюється очистка екрану;
- у змінній LastMode запам'ятовується попередній відеорежим.

Для створення текстового вікна можна скористатись процедурою **window(X1,Y1,X2,Y2)**, де **X1, Y1** - координати лівого верхнього кута вікна, а **X2, Y2** - правого нижнього. Потрібно відмітити, що координати визначаються від лівого верхнього кута екрану, починаючи з 0. Координати X збільшується з ліва на право, а Y - з верху в низ.

Після створення вікна потрібно задати колір його заднього фону, а колір тексту у вікні. Для встановлення кольору фону вікна використовується процедура **TextBackGround(*колір*)**, де ***колір*** - це змінна цілого типу **byte**, яка може приймати значення від 0 до 7. Вона встановлює колір фону вікна (див. таблицю 14.1). Замість числа допускається використання константи, що позначає відповідний колір. Для вказання кольору тексту у вікні використовується процедура **TextColor(*колір*)**, де ***колір*** - змінна цілого типу **byte**, яка може приймати значення від 0 до 15 (див. таблицю 14.1).

Таблиця 14.1

Значення кольорів для процедур TextColor та TextBackGround

Ім'я константи	Числове значення	Колір	Використовується процедурами
Black	0	чорний	TextColor, TextBackGround
Blue	1	синій	TextColor, TextBackGround
Green	2	Зелений	TextColor, TextBackGround
Cyan	3	блакитний	TextColor, TextBackGround
Red	4	червоний	TextColor, TextBackGround
Magenta	5	фіолетовий	TextColor, TextBackGround
Brown	6	Коричневий	TextColor, TextBackGround
LightGray	7	яскраво сірий	TextColor, TextBackGround
DarkGray	8	темно сірий	TextColor
LightBlue	9	яскраво синій	TextColor
LightGreen	10	яскраво зелений	TextColor
LightCyan	11	яскраво блакитний	TextColor
LightRed	12	яскраво червоний	TextColor
LightMagenta	13	яскраво фіолетовий	TextColor
Yellow	14	Жовтий	TextColor
White	15	Білий	TextColor
Blink	128	Миготіння	TextColor

Константа **Blink** задає миготіння тексту відображуваних символів і найчастіше використовується в процедурі **TextColor** у якості складового параметра. Наприклад,:

```
TextColor(Red+Blink);
```

Дія процедур **TextColor** та **TextBackGround** зводиться до запису в спеціальну змінну **TextAttr** модуля CRT відповідного значення. Ця змінна має тип **byte** і може приймати значення від 0 до 255.

Після встановлення кольору фону вікна та тексту, це вікно потрібно заповнити кольором. Це можна зробити з допомогою процедури без параметрів **ClrScr**. Ця процедура здійснює очистку вікна та заливання його фону відповідним кольором. При цьому курсор буде поміщено в лівий верхній кут вікна.

Для переміщення курсору в потрібну позицію використовується процедура **GotoXY(X1,Y1)**, де **X1,Y1** - змінні цілого типу word, які встановлюють потрібні координати курсору. Крім цього в модуль CRT входять дві функції без параметрів, які дозволяють визначити поточну координату курсору. Це функції **WhereX** - встановлює поточну координату курсору по осі X та **WhereY** - установлює поточну координату курсору по осі Y. Результатом роботи цих функцій є число цілого типу **word**.

Серед інших процедур та функцій для роботи з текстовими вікнами можна використати:

- **AssignCRT(*змінна*)** - процедура, яка перенаправляє стандартне введення-виведення на деякий логічний пристрій, що починає виконувати функції пристрою введення-виведення інформації. В якості ***змінної*** використовуються дані типу текстовий файл;
- **ClrEOL** - процедура, що знищує всі символи починаючи з позиції курсору до кінця рядка, без переміщення самого курсору;
- **DelLine** - процедура, що знищує рядок, в якому знаходиться курсор, і переміщує всі розміщені нижче рядки на один уверх;
- **InsLine** - процедура, що вставляє порожній рядок у положення курсору;
- **HighVideo** - процедура, що встановлює режим підвищеної яскравості символів;
- **LowVideo** - процедура, що відключає режим підвищеної яскравості символів;
- **NormVideo** - процедура, що встановлює режим стандартної яскравості символів.

Для роботи із клавіатурою в модулі CRT використовуються наступні функції:

- **ReadKey** - функція без параметрів, яка зчитує з буфера клавіатури значення останнього введеного символу. Результатом роботи функції є значення символьного типу. Потрібно відмітити, що буфер клавіатури організований у вигляді черги (за принципом „перший прийшов - перший вийшов“), тобто функція зчитує із буфера самий перший у черзі символ, одночасно знищує його з буфера. Якщо буфер порожній, то функція призупиняє виконання програми й очікує натискування клавіші;
- **KeyPressed** - функція без параметрів логічного типу, яка видає результат TRUE, якщо в буфері клавіатури знаходиться хоча б один символ (коли натиснуто клавішу) та FALSE в іншому випадку. Потрібно відмітити, що дані в буфер поступають при

натискуванні будь-якої клавіші, за виключенням NumLock, CapsLock, Shift, Ctrl, Alt. Дану функцію часто використовують для, того щоб зробити зупинку у виконанні програми до натискування довільної клавіші. Для цього слід указати наступну конструкцію:

Repeat until KeyPressed;.

Для роботи із внутрішнім динаміком комп`ютера та таймером у CRT використовуються наступні процедури:

- **Delay(*змінна*)** - процедура, що здійснює затримку у виконанні програми на час мілісекундах, який визначає ***змінна*** цілого типу **word**;
- **Sound(*змінна*)** - вмикає динамік і генерує звуковий сигнал з частотою, вказаною в герцах у ***змінній***. В якості змінної використовується цілий тип даних типу **word**. В результаті виконання даної процедури звуковий сигнал подається до тих пір поки в програмі не зустрінеться процедура **NoSound**;
- **NoSound** - зупиняє виведення звукового сигналу.

Потрібно відмітити, що при підключенні модуля CRT символи з кодами 0..31, що являються керуючими, відображаються на екрані у відповідності з таблицею ASCII як звичайні символи. Виключенням є лише деякі символи:

#7 - дзвінок Bell. Генерує звуковий сигнал;

#8 - BackSpace (BS). Переміщає курсор на одну позицію вліво. Якщо курсор на початку рядка, то ніяких подій не відбувається;

#9 - TAB, клавіша створення абзацного відступу;

#10 - переведення рядка LF. Переміщує курсор на один рядок униз. Якщо курсор уже знаходиться в нижньому рядку вікна, то вікно прокручується вверх на один рядок;

#13 - повернення каретки CR. Переміщає курсор на початок поточного рядка.

#27 - клавіша ESC.

Крім цього, деякі з клавіш мають розширені скен коди, що складаються з двох частин - основної (цифра 0) та розширеної. Коди таких клавіш представлені в таблиці 14.2.

Таблиця 14.2

Скен-коди розширених клавіш клавіатури

029	Ctrl	061	F3	069	NumLock	080	Down ↓
042	Лівий Shift	062	F4	070	Scroll Lock	081	Page Down
054	Правий Shift	063	F5	071	Home	082	Insert
055	PrintScr	064	F6	072	Up ↑	083	Delete
056	Alt	065	F7	073	Page Up	087	F11
058	CapsLock	066	F8	075	Left ←	088	F12
059	F1	067	F9	077	Right →		
060	F2	068	F10	079	End		

В якості прикладу використання модуля CRT розглянемо програму, яка визначає код натиснутої клавіші, враховуючи розширені коди клавіш.

```
uses crt;
var  Ch : char;
begin
       TextColor(4);  {задаємо колір символів}
       TextBackGround(7);  {задаємо колір вікна}
       Window(15,5,40,7);  {створюємо вікно}
       repeat
              ClrEOL; {очищаємо вікно}
              Ch := ReadKey;
              If  Ch=#0  {перевіряємо чи код клавіші розширений}
                     then
                     begin
                            Ch := ReadKey;  {прийом скен-коду}
                            case  ord(Ch) of
                                   59 : writeln('Клавіша  F1');
                                   60 : writeln('Клавіша  F2');
                                   ...       {і так далі}
                            end
                     end
                     else
                     begin
                            case ord(Ch) of
                                   8 : writeln('Клавіша  BackSpace');
                                   13 : writeln('Клавіша  Enter');
                                   27 : writeln('Клавіша  ESC');
                                   32 : writeln('Клавіша  пропуск');
                                   else  writeln('Клавіша ', Ch)
                            end
                     end;
       until  Ch=#27;   {виконувати цикл поки не натиснуто ESC}
end.
```

14.3. Модуль Graph

Для створення графічних зображень у мові Turbo Pascal використовується стандартний бібліотечний модуль Graph. Він являє собою бібліотеку підпрограм, які повністю забезпечують керування графічними режимами різних адаптерів дисплеїв. Бібліотека містить близько 80 графічних процедур і функцій, а також десятки стандартних констант і описів типів даних.

Модуль Grapg підключається до основної програми стандартним способом із допомогою зарезервованого слова uses.

Щоб завантажити програму, в якій використовуються процедури модуля Graph, потрібно помістити в робочий каталог (або спеціальний каталог описаний у конфігурації оболонки Turbo Pascal) відповідні графічні драйвери (файли з розширенням bgi).Якщо в графічному режимі використовуються шрифти, то в робочий каталог необхідно ще й помістити файли шрифтів (з розширенням chr). Серед графічних драйверів, що можна використовувати в Turbo Pascal можуть бути:

- CGA.BGI - для відеоадаптерів CGA та MCGA;
- EGAVGA.BGI - для відеоадаптерів EGA та VGA;
- HERC.BGI - для монохромних відеоадаптерів Hercules;
- ATT.BGI - для відеоадаптерів AT&T 6300;
- PC3270.BGI - для відеоадаптерів IBM 3270 PC;
- IBM8514.BGI - для відеоадаптерів IBM 8514.

При роботі з графікою в Turbo Pascal є можливість підключати не лише стандартні драйвери, але й користувацькі. Вони повинні лише задовольняти стандарту Turbo Pascal. В модулі Graph є процедура **RegisterBGIDriver**, яка забезпечує підтримку роботи з драйверами користувача.

Для встановлення відповідного графічного режиму в Pascal-програмі потрібно підключити графічний драйвер процедурою:

InitGraph(*GraphDriver, GraphMode, шлях*);

де

GraphDriver - константа (змінна) цілого типу (integer), яка встановлює тип відповідного графічного драйвера. При цьому можна вказувати, як назву константи, так і її числове значення (див. таблицю 14.3);

GraphMode - константа (змінна) цілого типу (integer), яка встановлює режим роботи графічного драйвера відповідного типу. При цьому можна вказувати, як назву константи, так і її числове значення (див. таблицю 14.3);

шлях - константа (змінна) стрічкового типу, яка встановлює шлях на диску до файлу потрібного графічного драйвера (нагадаю, що стрічкова константа береться в одинарні лапки). Якщо файл драйверу знаходиться в активному каталозі, то в якості шляху записуємо порожні лапки.

В кінці роботи створюваної програми графічний режим екрану обов'язково потрібно закрити процедурою

CloseGraph;

В якості прикладу встановимо графічний драйвер VGA із роздільною здатністю 640X480 точок при 16 кольорах (при умові, що він знаходиться в робочому каталозі програми):

```
GraphDriver := VGA;   {або GraphDriver := 9}
GraphMode := VGAHi; {або GraphMode := 2}
InitGraph(GraphDriver, GraphMode, '');.
```

Як видно із таблиці 14.3 однією з констант установлення типу драйверу є константа **Detect**. Якщо вказати дану константу, то система переходить в рижим автовизначення драйверу. В результаті цієї операції автоматично визначається тип драйверу і вмикається режим із максимальною роздільною здатністю для даного типу відеоадаптера. При цьому змінним ***GraphDriver*** і ***GraphMode*** присвоюється значення відповідних констант або код помилки (наприклад, у випадку відсутності файлу драйверу).

Таке значення параметра ***GraphDriver*** рекомендується встановлювати у випадках, коли програма повинна працювати на комп'ютерах з різними відеоадаптерами. Але

недоліком цього методу є те, що під час виконання програми всі драйвери повинні одночасно знаходитись в пам'яті або на диску. Для великих програм це може привести до значного зменшення швидкості виконання програми.

Таблиця 14.3

Значення констант типів драйверів та їх режимів роботи

Тип драйверу (GraphDriver)	**Файл драйверу**	**Режим (GraphMode)**	**Роздільна здатність екрану**	**Кількість кольорів**
Detect = 0	Автоматично	Змінна цілого типу	Автоматично	Автоматично
CGA = 1	cga..bgi	CGAC0 = 0	320x200	4
		CGAC1 = 1	320x200	4
		CGAC2 = 2	320x200	4
		CGAC3 = 3	320x200	4
		CGAHi = 4	640x200	2
MCGA = 2	cga..bgi	MCGAC0 = 0	320x200	4
		MCGAC1 = 1	320x200	4
		MCGAC2 = 2	320x200	4
		MCGAC3 = 3	320x200	4
		MCGAMed = 4	640x200	2
		CGAHi = 5	640x480	2
EGA = 3	egavga.bgi	EGALo = 0	640x200	16
		EGAHi = 1	640x350	16
EGA64 = 4	egavga.bgi	EGA64Lo = 0	640x200	16
		EGA64Hi = 1	640x350	4
EGAMono = 5	egevge.bgi	EGAMonoHi=3	640x350	2
IBM8514 = 6	Ibm8514.bgi	IBM8514Lo =0	640x480	256
		IBM8514 = 1	1024x768	256
HERCMono = 7	herc.bgi	HercMonoHi=0	720x348	2
ATT400 = 8	att.bgi	ATT400C0 = 0	320x200	4
		ATT400C1 = 1	320x200	4
		ATT400C2 = 2	320x200	4
		ATT400C3 = 3	320x200	4
		ATT400Med=4	640x200	2
		ATT400Hi = 5	640x400	2
VGA = 9	egavga.bgi	VGALo = 0	640x200	16
		VGAMed = 1	640x350	16
		VGAHi = 2	640x480	16
PC3270 = 10	pc3270.bgi	PC3270Hi=0	720x350	2
CurrentDriver=-128	Для функції GetGraphMode			

Для автоматичного визначення типу відеоадаптера можна також скористатись процедурою:

DetectGraph(*GraphDriver, GraphMode*);.

Ця процедура може викликатись, до ініціалізації графічного режиму (до **InitGraph**). При цьому для параметрів ***GraphDriver, GraphMode*** встановлюється значення максимального графічного режиму для даного відеоадаптера. Ці значення можна підставляти в процедуру **InitGraph**, що слідує після **DetectGraph**.

Існує можливість керувати режимами роботи графічного адаптера. Це реалізується з допомогою групи процедур та функцій, що вказуються після ініціалізації графіки.
Для визначення номеру поточного графічного режиму у встановленого драйвера можна визначити функцією **GetGraphMode**, яка видає результат цілого типу (Integer). Крім цього функція **GetMaxMode** (видає результат типу Word) встановлює номер режиму з максимальною роздільною здатністю для даного драйвера. Таким чином кожний графічний драйвер підтримує режими від **0** до **GetMaxMode**. Як правило цей ж результат можна отримати процедурою:

GetModeRange(*GraphDriver, LoMode, HiMode*);,

де змінні ***LoMode*** та ***HiMode*** встановлюють відповідно нижню та верхню межу режимів драйверу ***GraphDriver***.

Функція **GetDriverName** (результат типу String) видає ім'я поточного графічного режиму із списку констант указаних у таблиці 14.3.

Переключення між режимами роботи драйверу здійснюється процедурою:

SetGraphMode(*GraphMode*);,

яка перемикає систему у вказаний параметром ***GraphMode*** графічний режим і очищає екран дисплею.

Інколи, під час роботи програми виникає потреба тимчасово перейти з графічного режиму роботи в текстовий (що існував до ініціалізації графічного режиму). Для цього використовується процедура **RestoreCRTMode**. При виконанні цієї процедури графічний драйвер продовжує залишатись в пам'яті.

Зворотне переключення виконується функцією **GetGraphMode**, яка встановлює номер поточного графічного режиму. При цьому відбувається очистка екрану.

При виконанні будь-якої програми, в тому числі і в графічному режимі, можуть виникати помилки. В модулі Graph механізм виявлення помилок та генерації повідомлень про них реалізований з допомогою функцій **GraphResult** та **GraphErrorMsg**.

Функція **GraphResult** (результат типу Integer) установлює значення 0, якщо остання графічна операція виконалась без помилок, або число в діапазоні від -15 до -1, якщо виникла помилка (див. таблицю 14.4).

Для швидкої видачі повідомлення про тип помилки графічної системи використовується функція, яка перетворює результат виклику функції **GraphResult** у текстове повідомлення (результат типу String), що можна вивести процедурою **writeln**. Ця функція записується у форматі **GraphErrorMsg(*ErrorCode*)**, де ***ErrorCode*** - параметр, що є результатом роботи функції **GraphResult**.

Таблиця 14.4

Коди помилок графічного режиму

Константа	Код	Повідомлення про помилку	Пояснення
GrOk	0	No error	Немає помилок
GrNoInitGraph	-1	(BGI) Graphics not installed (use InitGraph)	Графічний режим не ініціалізований
GrNotDetected	-2	Graphics hardware not detected	Графічний драйвер не знайдено
GrFileNotFound	-3	Device driver file not detected	Немає BGI-файлу у вказаному каталозі
GrInvalidDriver	-4	Invalid device driver file	BGI-файл містить помилки
GrNoLoadMem	-5	Not enough memory to load driver	Для завантаження драйверу не вистачає оперативної пам'яті
GrNoScanMem	-6	Out of memory in scan file	Не вистачає оперативної пам'яті при роботі процедури **FillPoly**
GrNoFloodMem	-7	Out of memory in flood fill	Не вистачає оперативної пам'яті при роботі процедури **FloodFill**
GrNotFound	-8	Font file not found	У вказаному каталозі немає chr-файлу (файлу із шрифтом)
GrNoFontMem	-9	Not enough memory to load font	Не вистарчає оперативній пам'яті для завантаження шрифту
GrInvalidMode	-10	Invalid Graphics mode for selected driver	Неможливий режим для вибраного драйвера
GrError	-11	Graphics error	Помилка графіки
GrIOError	-12	Graphics I/O error	Помилка введення-виведення для графіки
GrInvalidFont	-13	Invalid font file	Помилки у файлі шрифту
grInvalidFontNum	-14	Invalid font number	Неіснуючий номер шрифту

Необхідно пам'ятати, що якщо функція **GraphResult**, викликається двічі після виконання однієї і тієї ж операції, то при повторному виклику її значення встановлюється 0. Тому рекомендується зберегти значення функції у певній змінній, якщо вона потрібна для подальшого використання.

Знаючи функції для визначення помилок при роботі графічної системи, можна визначити чи правильно ініціалізувався графічний режим. Для цього можна ввести наступну конструкцію операторів:

```
...
InitGraph(GraphDriver, GraphMode, '');
ErrorCode := GraphResult;  {зберегти результат ініціалізації}
if   ErrorCode <> grOk     {або   if  ErrorCode<>0}
then
begin
       writeln('Помилка графіки: ', GraphErrorMsg(ErrorCode));
       repeat  until KeyPsressed; {необхідно підключити модуль CRT}
       halt(1)
end;
...
```

В даному програмному блоці використано процедуру **halt(1)**, яка здійснює вихід із програми з одночасним закриттям графічного режиму. Тобто процедура **CloseGraph** тут використовується в неявному вигляді.

Вказання кольорів об'єктів та заповнень. Стилі заповнень

В графічному режимі Turbo Pascal координатна сітка веде свій відлік від лівого верхнього кута екрану. Ця точка має координати (0,0). Значення X збільшується в напрямку зліва на право, до значення, яке відповідає максимальній координаті (роздільна здатність по X мінус 1) установленого при ініціалізації графічного режиму. Координата Y змінюється зверху вниз, також до максимальної координати (роздільна здатність по Y мінус 1) установленого графічного режиму

Визначення значення максимальних координат екрану для встановленого графічного режиму здійснюється функціями **GetMaxX** (максимальна координата по X) та **GetMaxY** (максимальна координата по Y). В результаті роботи цих функцій отримуємо значення цілого типу **Integer**.

Щоб побудувати зображення, необхідно обов'язково вказати початкову позицію. В текстових режимах цю позицію визначає курсор. У графічних режимах видимого явного курсору немає, але є невидимий вказівник **CP (current pointer)**, що виконує ті ж функції.

У графічному режимі для переміщення CP використовуються процедури:

MoveTo(X,Y);, та **MoveRel(DX,DY);**,

де **X,Y** - дані цілого типу, що вказують, на яку координату перемістити CP, а **DX, DY** - на скільки точок перемістити курсор від активної координати. При цьому, якщо **DX, DY** додатні, то переміщення відбувається вправо (вниз), а якщо від'ємні - уліво (уверх).

Для визначення поточного положення графічного курсору використовуються функції **GetX** та **GetY**.

Перед побудовою об'єктів необхідно задати параметри їх кольору. Для різних типів адаптерів кількість кольорів, що одночасно відображається на екрані може бути різною. Але для всіх BGI-драйверів вона обмежена діапазоном цілочисельних значень від 0 до 15 (ці значення аналогічні кольорам, що представлені в таблиці 14.1 для кольору тексту модуля CRT).

Щоб вказати колір ліній створюваних об'єктів потрібно скористатись процедурою **SetColor(*колір*)**, де ***колір*** - число від 0 до 15. Для вказання кольору фону об'єктів (усього екрану) можна скористатись процедурою **SetBKColor(*колір*)**, де ***колір*** - також число від 0 до 15.

Для того щоб вибрати стиль ліній з яких будуть будуватись об'єкти на графічному екрані потрібно задатись процедурою **SetLineStyle(*тип,стиль,товщина*)**, де ***тип*** - тип лінії, ***стиль*** - її стиль та ***товщина*** - товщина лінії яка може приймати два значення 1 - нормальна товщина та 3 - товста. Усі ці параметри приймаються типу **Word**. ***Тип*** лінії може приймати значення: SolidLn=0 - суцільна, DottedLn=1 - пунктирна, CenterLn=2 - штрих-пунктирна, DashedLn=3 - штрихова та UserBitLn=4 - задана користувачем. Якщо ***тип*** лінії не приймається рівним 4 (UserBitLn), то значення параметру ***стиль*** ігнорується.

Значення типу рівне 4 встановлюється у випадку, якщо користувача не задовольняє ні один із стандартних типів ліній. Тоді для визначення типу лінії потрібно користуватись такими правилами:

- Відрізок, з якого будуємо шаблон представляє собою сукупність фрагментів, кожен із яких має довжину 16 пікселів. Якщо довжина відрізку не ділиться на 16, то останній відрізок відкидається;
- Можна задавати шаблон-комбінацію 16-ти засвічених (відповідає 1), та погашених (відповідає 0) пікселів. Суцільна лінія наприклад буде задаватись 1111111111111111 - усього 16 розрядів.

Використовуючи ці правила двійкове число перетворюємо в 10-кове (використовуючи правила переведення в іншу систему числення). Отримане число буде значенням стилю лінії створеної користувачем (параметр ***стиль***).

Щоб отримати інформацію про поточний стиль лінії, можна використати процедуру **GetLineSettings(*LineType*)**, де ***LineType*** - змінна типу **LineSettingsType** що визначає стиль лінії. Тип можна **LineSettingsType** описати наступним чином:

```
type
       LineSettingsType = record
       LineStyle : word;  {тип лінії}
       Pattern : word;  {стиль лінії}
       Thickeness : word;   {товщина}
       end;
```

Для того щоб отримати значення поточних кольорів ліній та фону використовується функції відповідно **GetColor** та **GetBkColor**. Якщо необхідно замалювати весь екран кольором фону, то потрібно скористатись процедурою очистки екрану **ClearDevice**.

Для того щоб зафарбувати кольором фону лише замкнутий об'єкт (контур) використовується процедура **FloodFill(*X,Y,рамка*)**, де ***X,Y*** - координати точки в середині замкнутого контуру (бажано по ближче до середини), а ***рамка*** - задає колір цього замкнутого контуру (типу **Word**). Якщо координати точки знаходяться поза контуром, то заповнюється простір поза цим контуром. У випадку, коли вказаний контур не є замкнутим, то заповнюється весь екран.

В графічному режимі Turbo Pascal є можливість заповнювати контури не лише суцільним фоном, але й указувати інші стилі заповнення. Для цього щоб вказати стиль заповнення контуру потрібно задати процедуру **SetFillStyle(*стиль,колір*)**, де ***стиль*** - параметр типу **Word**, що вказує стиль заповнення та ***колір*** - параметр типу ***Word,*** який визначає колір заповнення. В Turbo Pascal можна використовувати 13 стилів заповнення (див таблицю 14.5).

Для того щоб визначити поточний стиль заповнення використовується процедура **GetFillSettings(*FillType*)**, де FillType змінна типу **FillSettingsType**, яка встановлює поточний тип заповнення. Отже, для заповнення контурів у модулі Graph передбачено специфічний тип даних **FillSettingsType**, який можна описати наступним чином:

```
type
       FillSettingsType = record
       Pattern : word;    {шаблон заповнення}
       Color : word;    {колір заповнення}
       end;
```

Таблиця 14.5

Стилі заповнення процедурою SetFillStyle

Константа	Значення	Опис візерунку
EmptyFill	0	Суцільним кольором фону
SolidFill	1	Суцільним кольором ліній контуру
LineFill	2	Штрихувими лініями — — —
LtSlashFill	3	Похилими лініями / / / звичайної товщини
SlashFill	4	Похилими лініями / / / подвійної товщини
BkSlashFill	5	Похилими лініями \ \ \ подвійної товщини
LtBkSlashFill	6	Похилими лініями \ \ \ звичайної товщини
HatchFill	7	Заповнення в клітинку
XhatchFill	8	Заповнення в похилу рідку клітинку
Густу	9	Заповнення в похилу густу клітинку
WideDotFill	10	Заповнення рідкими крапками
CloseDotFill	11	Заповнення густими крапками
UserFill	12	Заповнення, що встановлюється користувачем

Якщо ні один із стандартних типів заповнення не задовольняє програміста, то можна створити свій стиль. Для цього у вищеописаній процедурі вказується 12-ий стиль, попередньо створивши його процедурою **SetFillPattern(*Pattern,Color*)**, де ***Pattern*** - новий шаблон заповнення та ***Color*** - його колір. Шаблон користувача займає 8 байтів (64 біти). Для його створення потрібно намалювати матрицю прямокутних комірок 8X8 (наприклад, на міліметровому папері). Кожна комірка буде відповідати пікселю на екрані дисплею. Після чого створити візерунок шаблону і записати побітово 8 двійкових чисел (кожне число - рядок матриці), при цьому кожна зафарбована комірка буде 1, а не зафарбована - 0. На наступному етапі всі двійкові числа необхідно перетворити в десяткові (за правилами переведення систем числення), а тоді записати масив із 8-ми цілих чисел типу **byte**. Таким чином отримаємо шаблон стилю заливки.

Отже шаблон стилю заливки (***Pattern***) має тип даних **FillPatternType**, який можна описати:

```
type
        FillPatternType = array[1..8] of byte;.
```

Для отримання детальної інформації про поточний шаблон стилю заповнення можна скористатись процедурою **GetFillPattern(*PattMatrix*)**, де ***PattMatrix*** - змінна типу **FillPatternType,** яка видає масив чисел, що показують візерунок шаблону заповнення.

Побудова простих графічних об'єктів

Щоб побудувати точку на графічному екрані потрібно використати процедуру **PUTPixel(*X,Y,колір*)**, де ***X,Y*** - дані типу integer, які вказують координати цієї точки, а ***колір*** - її колір (тип **Word**) у діапазоні від 0 до 15. Для того, щоб визначити поточний колір певної точки на графічному екрані можна скористатись функцією ***GetPixel(X,Y)***, яка видає число типу **Word**, що вказує колір цієї точки.

Для побудови прямих ліній на екрані в модулі Graph використовується процедура **Line(*X1,Y1,X2,Y2*)**, де ***X1***, ***Y1*** - координата початкової, а X2, Y2 - кінцевої точки відрізку прямої.

Крім цього, для побудови відрізків прямих у Graph можна також скористатись процедурами:

- **LineTo(*X,Y*)** - будує пряму лінію від точки поточного положення вказівника до точки з координатами ***X***,***Y***;
- **LineRel(DX,DY)** - будує пряму лінію від точки поточного положення вказівника до точки яка зміщена на величину ***DX*** по горизонталі та ***DY*** по вертикалі.

Розглянемо тепер інші процедури для побудови простих геометричних фігур:

- **Rectangle(*X1,Y1,X2,Y2*)** - побудова прямокутника, в якому ***X1***,***Y1*** (типу Integer) координати лівого верхнього кута, а ***X2***,***Y2*** (типу Integer) - правого нижнього;
- **Bar(*X1,Y1,X2,Y2*)** - побудова зафарбованого прямокутника згідно поточного фону зафарбування, який задається процедурою **SetFillStyle** та **SetBkColor**. В даному випадку ***X1***,***Y1*** (типу Integer) координати лівого верхнього кута прямокутника, а ***X2***,***Y2*** (типу Integer) - правого нижнього;
- **Bar3D(*X1,Y1,X2,Y2,D3,Top*)** - побудова паралелепіпеда, де ***X1***,***Y1*** (типу Integer) - координати лівого верхнього кута його основи, ***X2***,***Y2*** (типу Integer) - правого нижнього, ***D3*** (типу Word) - глибина (висота) паралелепіпеда, а ***Top*** (типу Boolean) - задає режим відтворення верхньої площини (**True** - відображати, **False** - не відображати);
- **DrawPoly(*N,PolyPoints*)** - побудова ламаної лінії (багатокутника) з довільною кількістю точок заломлення. В даному випадку ***N*** (типу Word) кількість точок ламаної, а ***PolyPoint*** - змінна, що представляє собою запис, який складається з двох полів - координат точки. Цей тип у модулі Graph описаний, як **PointType**:

```
type
      PointType = record
      X, Y : integer;   {координати точки}
      end;
```

- **FillPoly(*N,PolyPoints*)** - побудова зафарбованого багатокутника з довільною кількістю точок заломлення. Параметри цієї процедури повністю аналогічні параметрам процедури **DrawPoly**. Потрібно відмітити, що координати першої та останньої точок фігури повинні співпадати (багатокутник повинен бути замкнутий). Фон зафарбування багатокутника задається процедурою **SetFillStyle** та **SetBkColor**;
- **Circle(*X,Y,R*)** - побудова кола, в якому ***X***,***Y*** (типу Integer) координати його центра, а ***R*** (типу Word) - радіус кола;
- **Arc(*X,Y,StAngle,EndAngle,R*)** - побудова дуги кола, в якій ***X***,***Y*** (типу Integer) координати центра кривизни дуги, ***StAngle,EndAngle*** (типу Word) - початковий та кінцевий кути дуги (визначаються від горизонтальної осі в градусах) та ***R*** (типу Word) - радіус кривизни дуги;
- **GetArcCords(*ArcCords*)** - видає інформацію про параметри даної дуги. При цьому змінна ***ArcCords*** - це змінна типу **ArcCordsType**, який у модулі Graph описується як:

```
type
       ArcCordsType = record
       X, Y  : integer;     {центр дуги}
       XStar,YStar  : integer;    {початок дуги}
       XEnd, YEnd : integer;    {кінець дуги}
       end;
```

- **PieSlice(*X,Y,StAngle,EndAngle,R*)** - побудова зафарбованого сектора кола. Усі параметри даної процедури аналогічні параметрам процедури **Arc**. Фон зафарбування задається процедурою **SetFillStyle** та **SetBkColor**;
- **Ellipse(*X,Y,StAngle,EndAngle,XR,YR*)** - побудова еліптичних дуг та еліпсів, При цьому змінні ***X,Y*** (типу Integer) - це координати центра кривизни дуги, ***StAngle,EndAngle*** (типу Word) - початковий та кінцевий кути еліптичної дуги (визначаються від горизонтальної осі в градусах) та ***XR, YR*** (типу Word) - горизонтальна та вертикальна осі еліпса;
- **Sector(*X,Y,StAngle,EndAngle,R*)** - побудова зафарбованого сектора еліптичної дуги. Усі параметри даної процедури аналогічні параметрам процедури **Ellipse**. Фон зафарбування задається процедурою **SetFillStyle** та **SetBkColor**;
- **FillEllipse(*X,Y,XR,YR*)** - побудова зафарбованого еліпса. Фон зафарбування задається процедурою **SetFillStyle** та **SetBkColor**. При цьому змінні ***X,Y*** (типу Integer) - це координати центра еліпса, а ***XR, YR*** (типу Word) - горизонтальна та вертикальна осі еліпса.

Робота з текстом у модулі Graph

Відображення тексту в графічному режимі має ряд відмінностей від текстового режиму вікна. Основна відмінність полягає в тому, що в графічному режимі всі дії виконуються лише зі стрічковими константами і змінними. Уся числова інформація повинна попередньо перетворюватись у символьну. Інші відмінність полягає у можливості використання в графічному режимі різних шрифтів.

Перед виведенням текстової інформації, у модулі Graph необхідно попередньо задати шрифт та стиль виведення інформації.

В комплект поставки пакету Turbo Pascal входить набір штрихових шрифтів. Файли цих шрифтів мають розширення *.chr. В штрихових шрифтах при побудові символу використовується не матричний (як у стандартних шрифтах для текстового режиму), а векторний спосіб. Це дозволяє змінювати розміри шрифтів без втрати їх якості.

Даний стандартний набір шрифтів включає в себе чотири шрифти (хоча їх кількість можна розширити). Крім цього при роботі з Turbo Pascal доступний системний матричний шрифт 8x8 для графічних режимів (завжди доступні символи ASCII таблиці з кодами від 0 до 127 і символи з кодами від 128 до 255, якщо їх матриці завантажені в пам'ять ПК, тобто підключені відповідні драйвери національних алфавітів). Цей шрифт встановлюється в модулі Graph по замовчуванню, якщо в програмі не задано команду, яка встановлює тип шрифту (йому відповідає константа **DefaultFont**).

Для встановлення відповідного шрифту та його стилю використовується процедура:

SetTextStyle(*Font,Direct,Size*);,

де ***Font*** - це змінна типу Word, яка задає тип шрифту. Для її вказання в модулі Graph використовуються стандартні константи, які представлені в таблиці 14.6,

Direct - змінна типу Word, яка вказує орієнтацію та напрямок виведення символів. Значення змінної ***Direct*** рівне 0 відповідає стандартному для текстового режиму виведенню стрічки зліва направо. При значенні ***Direct*** рівному числу 1 кожен символ буде повернутий на 90° у напрямку проти годинникової стрілки (символи „лежачі на боці“) і відображення буде здійснюватись знизу вверх. Якщо вказати Directi рівний 2, то орієнтація символів буде такою ж, як при ***Direct***=1, але виведення стрічки буде здійснюватись в горизонтальному напрямку зліва направо;

Size - встановлює розмір відображуваних символів. Цей параметр може набувати значення від 0 до 10. Причому 0 відповідає розміру шрифту, який стандартний для даного шрифту. Щоб взнати розміри стрічки символів по вертикалі та горизонталі можна скористатись відповідно функціями **TextHeight(*стрічка*)** та **TextWidth(*стрічка*)**, які визначають висоту та ширину стрічки символів у пікселях.

Існує ще один спосіб встановлення висоти та та ширини стрічки символів. Для цього слід скористатись процедурою **SetUserCharSize(*MultX,DivX,MultY,DivY*)**. Першими двома параметрами задається розмір по горизонталі, а наступні - по вертикалі. Якщо прийняти за 1 значення ширини символу стандартного шрифту, то відношення ***MultY***/***DivY*** - буде висотою. Шрифт та напрямок виведення стрічки можна задати з допомогою процедури **SetTextStyle** - це можна зробити до або після виклику **SetUserCharSize**.

Таблиця 14.6

Константи стандартних шрифтів модуля Graph

Константа	Числове значення	Опис	Файл
DefaultFont	0	Матричний шрифт 8x8 (по замовч.)	--
TriplexFont	1	Напівжирний шрифт	Trip.chr
SmallFont	2	Тонкий шрифт	litt.chr
SanSerifFont	3	Рублений шрифт	sans.chr
GothicFont	4	Готичний шрифт	goth.chr

Кожен раз, коли в програмі викликається процедура **SetTextStyle**, файл відповідного шрифту (див. таблицю 14.6) зчитується з диску і завантажується в пам’ять. При цьому потрібно враховувати наступне:

- якщо в програмі використовуються штрихові шрифти, необхідно, щоб файл відповідного шрифту знаходився в тому ж каталозі, що й BGI-файли. У іншому випадку система не зможе їх знайти й буде використовувати матричний шрифт 8X8;
- при переключенні між декількома шрифтами виконання програми буде зупинене на час, який необхідний на зчитування відповідного шрифту з диска.

Визначити результат зчитування шрифту з диску можна з допомогою функції **GraphResult** (див. таблицю 14.7).

Таблиця 14.7

Результати роботи функції GraphResult

Результат	Значення
0	Успішне завершення операції
-8	Відповідний файл CHR не знайдено
-9	Не достатньо пам'яті для завантаження відповідного шрифту
-11	Помилка графіки
-12	Помилка введення-виведення графіки
-13	Невірний уміст файлу шрифту
-14	Невірний номер шрифту

При виведенні текстових стрічок на екран, можна задавати розміщення наступної стрічки (що буде виводитись) відносно поточного положення вказівника курсору. Для цього слід скористатись процедурою **SetTextJustify(*Horiz*,*Vert*)**, де змінні вказують позицію відображуваних символів відповідно у горизонтальному та вертикальному напрямках. Константи, які використовуються в якості значень цих змінних представлені в таблицях 14.8 та 14.9.

Таблиця 14.8

Значення констант для вирівнювання тексту в горизонтальному напрямку

Horiz	Значення	Призначення
LeftText	0	Встановлює поточну позицію вказівника лівою границею стрічки
CenterText	1	Центрує стрічку відносно вказівника
RightText	3	Встановлює поточну позицію вказівника правою границею стрічки

Таблиця 14.9

Значення констант для вирівнювання тексту у вертикальному напрямку

Horiz	Значення	Призначення
BottomText	0	Розміщує символи нижче вказівника
CenterText	1	Центрує стрічку відносно вказівника
TopText	3	Розміщує символи вижче вказівника

Для виведення самої текстової стрічки в Turbo Pascal використовуються наступні процедури:

- **OutText(*стрічка*)** - виводить на екран текстову стрічку, починаючи з поточної позиції вказівника CP;
- **OutTextXY(*X,Y,стрічка*)** - виводить на екран текстову стрічку, починаючи з позиції з координатами ***X***,***Y***.

Нагадаємо також, що числова інформація в модулі Graph не виводиться. Для виведення такої інформації, її спочатку потрібно перетворити у стрічку з допомогою процедури **Str**.

Потрібно відмітити, що, якщо програма використовує декілька шрифтів одночасно, то їх необхідно попередньо розмістити в пам'ять і провести реєстрацію кожного з них послідовним використанням функції **RegisterBGIFont(FontPtr)**. Дана функція реєструє шрифт і встановлює його код або код помилки при неправильній реєстрації. Реєстрація повинна відбуватись до ініціалізації графічного режиму, в результаті чого шрифт стає доступним у будь-який момент роботи програми.

Щоб здійснити реєстрацію шрифту необхідно:

- в динамічній пам'яті вказати область, розмір якої буде дорівнювати розміру файла шрифта;
- зчитати файл шрифта і помістити в динамічну пам'ять;
- зареєструвати вказівник на шрифт у пам'яті за допомогою функції **RegisterBGIFont**.

Нижче показано приклад реєстрації файлу готичного шрифту:

```
uses graph;
var
        Driver,Mode : integer;
        Font : File;
        P : pointer;
begin
        assign(Font,'goth.chr');
        reset(Font,1);
        {виділення пам'яті}
        GetMem(P,FileSize(Font));
        {зчитування в пам'ять}
        BlockRead(Font,P^,FileSize(Font));
        Close(Font);
        {реєстрація}
        if RegisterBGIFont(P)<0 then Halt(1);
        Driver := Detect;
        InitGraph(Driver,Mode,'');
        If GraphResult<>0 then Halt(1);
        {підключення шрифту}
        SetTextStyle(GothicFont,0,6);
        ...
```

Turbo Pascal містить спеціальні засоби підключення шрифтів, розроблених користувачем, а також їх включення у EXE-файл. Розробка нових шрифтів ускладнена ще й тим, що формат CHR, на сьогодні ще недостатньо описаний. Це ускладнює процес створення графічного інтерфейсу шрифтів на українській мові, оскільки штрихові шрифти не містять у собі символів кирилиці. Для того щоб підключити розроблений користувачем, або формовий шрифт у форматі CHR, потрібно скористатись функцією **InstallUserFont(*ім'я*)**, де ***ім'я*** - назва файлу шрифту (із розширенням CHR). Дана функція (результат типу integer) встановлює ціле число, яке надається функції **SetTextStyle** в якості ідентифікатора шрифту.

Робота із фрагментами малюнка. Використання відеосторінок

Розглянемо тепер набір процедур модуля Graph, які дозволяють зберігати та відтворювати окремі фрагменти зображення на екрані. Нижче приведено дві процедури та одна функція, які використовуються для запам'ятовування в буфері пам'яті та наступного

відтворення з нього фрагментів графічного зображення, що мають форму прямокутника. Це дозволяє при створенні графічного зображення використовувати вже готові одинакові фрагменти для побудови завершеного зображення.

При роботі з буфером завжди важливо знати розмір буфера пам'яті, що необхідна для запам'ятовування вигляду об'єкту (це значення може відрізнятись в залежності від відеоадаптерів). Для цієї мети використовується функція **ImageSize(*X1,Y1,X2,Y2*)**, де ***X1,Y1,X2,Y2*** (типу integer) - координати лівого верхнього та правого нижнього кутів прямокутної області. Значення, що видає дана функція (типу Word) представляє собою розмір пам'яті в байтах, що необхідна для запам'ятовування цієї області.

Оскільки, для зберігання фрагменту зображення найкраще використовувати динамічну пам'ять, то результат роботи даної функції краще за все використовувати в якості вхідної інформації для процедури **GetMem** (див. розділ 12), яка виділяє вказаний об'єм пам'яті в динамічній області.

Збереження зображення фрагменту об'єкта в пам'яті здійснюється процедурою **GetImage(*X1,Y1,X2,Y2,BitMap*)**, де ***X1,Y1,X2,Y2*** (типу integer) - координати прямокутної області, ***BitMap*** - нетипізований параметр, який повинен бути більший або рівний 6 плюс розмір пам'яті, що відведена для зони екрану. Перші два слова параметра ***BitMap*** визначають ширину та висоту частини екрану, третє слово зарезервоване. Інша частина параметру ***BitMap*** використовується для збереження самого двійкового представлення ділянки екрану. Для визначення розміру пам'яті, що необхідний для параметра ***BitMap***, необхідно скористатись функцією **ImageSize**, яка представлена вище. Потрібно відмітити, що розмір зображення, що зберігається не повинен перевищувати 64K.

Процедура **PutImage(*X,Y,BitMap,Mode*)** виводить збережену в буфері ***BitMap*** прямокутну область зображення, верхній лівий кут якої відповідає координатам ***X***,***Y***. На відміну від процедури **GetImage** у даній процедурі достатньо задати координати лівого верхнього кута зображення. Це пояснюється тим, що в структурі ***BitMap*** перші два слова (чотири байти) містять ширину та висоту в пікселях збереженого зображення.

Для вказання режиму виведення зображення служить параметр ***Mode***, який задає оператор, що буде використовуватись при виведенні двійкового коду відповідної ділянки екрану. Далі показано перелік допустимих значень констант цього параметру (у фігурних дужках представлені відповідні оператори Assembler): **CopyPut = 0 {mov}**, **XORPut = 1 {xor}**, **ORPut = 2 {or}**, **ANDPut = 3 {and}**, **NOTPut = 4 {not}**.

Необхідно відмітити, що виведення фрагментів у режимі **XORPut** забезпечує його видимість на будь-якому фоні. Повторне виведення того ж фрагменту означає, що нове зображення буде знищене, а відновлене попереднє.

При використанні режиму **ANDPut** пікселі зображення, що буде виводитись, які мають чорний колір, будуть чорними також на екрані, а пікселі, що мають білий колір, не змінюють вмісту екрану. Цей режим корисний для „вирізання" із фону фрагменту довільної форми чорного кольору. В цьому режимі можна виводити на екран „непрозорі" об'єкти, в тому числі і при побудові рухомих зображень.

В наступному прикладі показано використання операції збереження і відображення області екрану з різними режимами введення.

```
{Робота із фрагментами зображення}
uses  graph;
var
       Mode : integer;
       Size  : word;  {розмір ділянки}
       X1,Y1 : word;  {координати початкової точки}
       P : pointer; {вказівник на ділянку}
begin
       InitGraph(Detect,Mode,'');{автоматичне визначення відеодрайверу}
       if  GraphResult<>0  then  halt(1);
       {побудова об'єкта}
       SetFillStyle(SolidFill,LightRed);
       PieSlice((GetMaxX div 2),(GetMaxY div 2),0,360,100);
       readln;
       {визначення розміру ділянки}
       X1 := GetMaxX div 2 - 100;
       Y1 := GetMaxY div 2 - 100;
       Size := ImageSize(X1,Y1,X1+200,Y1+200);
       {виділення пам'яті для ділянки}
       GetMem(P,Size);
       {збереження вказаної ділянки в пам'яті}
       GetImage(X1,Y1,X1+200,Y1+200,P^);
       ClearDevice;
       SetBkColor(Blue);
       {відображення ділянки в режимі копіювання}
       PutImage(X1,Y1,P^,CopyPut);
       readln;
       {відображення ділянки в режимі XOR}
       {зображення відділяється з екрана}
       PutImage(X1,Y1,P^,XORPut);
       readln;
       CloseGraph;
end.
```

Потрібно зауважити, що пам'ять відеоадаптерів (відеобуфер) представляється у вигляді відеосторінок (подалі будемо використовувати термін „сторінка"), і по замовчуванню в графічному режимі дії здійснюються з нульовою сторінкою, то може виникнути ситуація, що результат роботи програми не буде відображуватись на екрані. В такому випадку, для того, щоб зображення з'явилось на екрані, потрібно дати команду зчитати видимою цю відеосторінку.

Пам'ять відеоадаптерів EGA, VGA і Hercules містить більше, ніж одну відеосторінку (найчастіше відеосторінки нумеруються з 0). Так, наприклад, режим EGALo та VGALo містять по 4 відеосторінки, а режими EGAHi та VGAMed - по 2 (VGAHi - лише одну).

Для роботи з відеосторінками в Turbo Pascal використовуються дві процедури: **SetActivePage** та **SetVisualPage**. Їх часто використовують при створенні анімаційних програм. Процедура **SetVisualPage(*сторінка*)** (***сторінка*** - змінна типу word) встановлює в якості видимої вказану сторінку, тоді як процедура **SetActivePage(*сторінка*)** (***сторінка*** - змінна типу word) встановлює вказану сторінку в якості активної - це означає, що всі графічні операції будуть виконуватись над цією відеосторінкою.

Використовуючи ці дві процедури можна легко створювати рухоме анімаційне зображення. Для цього робимо один із кадрів рухомого зображень (одну зі сторінок) активним, в той час як на іншій відеосторінці створюємо наступний кадр. Таким чином

можна створити в програмі плавне переміщення об'єктів.

В наступному прикладі складне зображення, що складається з хаотично виникаючих відрізків різних кольорів формується на активній, але невидимій сторінці, а тоді з допомогою процедури **SetVisualPage** миттєво відображається на екрані.

```
uses  graph,crt;
var
      Mode : Integer;
begin
      Mode := VgaMed; {Режим із двома відеосторінками}
      InitGraph(VGA,Mode,'');
      if  GraphResult<>0 then halt(1);
      randomize;
      {активна поточна видима сторінка}
      SetActivePage(0);
      OutText('Нажміть довільну клавішу');
      {Зображення формується на невидимій сторінці}
      SetActivePage(1);
      SetVisualPage(0);
      SetLineStyle(0,0,2);
      repeat
            {побудова випадково вибраним кольором довільних ліній}
            SetColor(Random(GetMaxColor));
            LineTo(Random(GetMaxX),Random(GetMaxY));
      until  KeyPressed;  {поки не натиснуто довільну клавішу}
      {відображення зображення}
      SetVisualPage(1);
      readln;
      CloseGraph
end.
```

В модулі Graph передбачені дві процедури керування динамічною пам'яттю: **GraphFreeMem** та **GraphGetMem**. Перша з них звільняє пам'ять, розподілену для відеодрайверів, а друга - розподіляє пам'ять для цих драйверів. Синтаксис цих процедур має вигляд:

GraphGetMem(*P* : Pointer, *Size* : Word);
GraphFreeMem(*P* : Pointer, *Size* : Word);.

Потрібно зауважити, що в модулі Graph, є два стандартних вказівники (типу **Pointer**), які містять адреси цих стандартних процедур. Це **GraphGetMemPtr** - вказівник на програму розподілу пам'яті та **GraphFreeMemPtr** - вказівник на програму звільнення пам'яті.

Виклик процедур керування відеопам'яттю здійснюється, коли:

- при звертанні до графічного буферу загального призначення, розмір якого можна встановити з допомогою процедури **SetGraphBufSize** (по замовчуванню 4 кілобайти);
- при завантаженні драйверів пристроїв під час ініціалізації графічного режиму (файли *.bgi);
- при завантаженні файлів штрихових шрифтів під час виконання процедури **SetTextStyle** (файли *.chr).

Графічний буфер завжди розміщується в динамічній пам'яті (купі).

14.4. Модуль Strings

Turbo Pascal містить тип символьних стрічок, які називаються стрічками із завершуючим нулем. Такий тип даних введено для сумісності програм написаних на Turbo Pascal, з типами даних для Windows, а також для встановлення відповідності з іншими мовами програмування (C, Assembler і ін.).

Функції для роботи з такими стрічками об'єднанні в модулі **Strings**. Вони дозволяють обробляти такі стрічки та перетворювати їх у стандартний для Pascal тип string і навпаки.

Нагадаємо, що в Turbo Pascal стрічки стандартного типу String складаються із байту, що містить вказівку на кількість символів у стрічці та саму послідовність символів. Максимальна довжина такої стрічки - 255 символів.

Стрічки із завершуючим нулем не містять байту, що вказує на довжину стрічки. На відміну від звичайних стрічок, вони містять послідовність не нульових символів, за якими слідує символ NULL (#0). Ніяких обмежень на стрічки із завершуючими нулями не накладається, але у 16-ти розрядних архітектурах DOS та Windows вони не можуть перевищувати 65535 символів.

Ці стрічки описуються стандартним типом Pchar. Фактично, вони вказують на символ **Pchar=^Char**, але розширений синтаксис (див. далі) дозволяє ставити у відповідність стрічкам із завершуючим нулем символьний масив типу

Pchar=array[0..X] of char,

де X - додатне число типу Integer, яке визначає кількість символів у стрічці, не враховуючи завершуючого нуля з кодом #0. Такі масиви називаються **масивами з нульовим індексом**. На відміну від типу String, символ з індексом 0 є першим символом стрічки, а останній, з індексом X, - завершуючим символом із кодом 0.

При роботі над стрічками із завершуючим нулем доцільно використовувати розширений синтаксис, який задається компілятору директивою **{$X+}** (хоча її можна не вказувати на початку програми, оскільки вона приймається по замовчуванню). Розширений синтаксис дозволяє використовувати особливі правила для стандартного типу Pchar і масивів із нульовим індексом. Наприклад, з такими масивами можна буде використовувати процедури **read**, **readln** і **str**, а процедури **write**, **writeln**, **val**, **assign**, **rename** можуть використовуватись як з масивами, так і із символьними вказівниками.

Крім цього, при розширеному синтаксисі можна викликати функції аналогічно виклику процедур. Це може бути корисно, коли результат, що встановлюється цими функціями, не має суттєвого значення.

При використанні розширеного синтаксису можна, також, змінним типу Pchar присвоювати значення текстових констант. Наприклад, якщо маємо змінну S типу Pchar, то можна записати S := 'Стрічкова константа'. В результаті такого присвоєння значення вказівника буде відповідати ділянці пам'яті, що містить копію стрічкової константи, але вже із завершуючим нулем.

Допускається виклик процедур і функцій з формальними параметрами типу Pchar, в яких фактичні параметри є стрічковими константами.

Потрібно також відмітити, що типізована константа типу Pchar може ініціалізуватись

стрічковою константою. Це справджується також для складних типів даних як масиви Pchar і записи, а також для об'єктів, що містять поля типу Pchar.

Стрічкові константи завжди обчислюються як стрічка типу String, навіть якщо вони ініціалізують типізовану константу типу Pchar. Отже, довжина стрічкової константи обмежена і не може перевищувати 255 символів.

Як вже зазначалось, при використанні розширеного синтаксису символьний масив із нульовим індексом також сумісний з типом PChar. Це означає, що при використанні типу Pchar може використовуватись відповідний символьний масив. Якщо символьний масив використовується замість значення типу Pchar, компілятор перетворює символьний масив у вказівник-константу, значення якого відповідає адресі першого елементу масиву.

Можна ініціалізувати типізовану константу, що має тип символьного масиву з нульовим індексом, з допомогою стрічкової константи, довжина якої менша, ніж розмір масиву. Символи, що залишились, заповнюються значенням **NULL(#0)**. Масив буде містити стрічку із завершуючим нулем.

В Turbo Pascal не має вбудованих функцій та процедур для роботи із стрічками із завершуючими нулями. Такі функції та процедури зібрані в модулі Strings. Розглянемо основні елементи цього модуля.

Для одночасного використання звичайних стрічок та стрічок із завершуючими нулями в модулі Strings передбачені функції перетворення типів. Так, для перетворення стрічки типу PChar у стрічку типу String, використовується функція ***S*:=StrPas(*PS*)**, де ***PS*** - це змінна типу PChar, а ***S*** - змінна типу String.

Зворотне перетворення стрічки типу String у стрічку типу PChar здійснюється функцією **StrPCopy(*Dest,Source*)**. Дана функція копіює стрічку ***Source*** (типу String) у стрічку ***Dest*** (типу PChar) і встановлює вказівник на ***Dest***. При цьому перевірка довжини отримуваної стрічки не відбувається. Перед виконанням функції необхідно переконатись, що в буфері стрічки ***Dest*** достатньо місця (хоча б довжина стрічки ***Source*** плюс 1). Функція **StrPCopy** дає результат типу PChar, який не є значимим для ходу програми, тому дану функцію можна записувати, так як це прийнято для процедур.

До функцій перетворення можна віднести також функції перетворення великих літер стрічки типу PChar у малі і навпаки. Так, для перетворення всіх символів стрічки ***S1*** (тип PChar) у малі літери (стрічка ***S*** типу PChar) можна скористатись функцією ***S*:=StrLower(*S1*)**. Для перетворення всіх символів стрічки у великі літери використовується функція ***S*:=StrUpper(*S1*)**. Обидві функції по завершені роботи встановлюють вказівник на ***S1***.

Для визначення довжини стрічки ***S*** типу PChar використовується функція ***N*:=StrLen(*S*)**. Дана функція встановлює результат ***N*** типу Word. Потрібно відмітити, що завершуючий нульовий символ у довжину стрічки не входить.

Копіювання стрічок типу PChar виконується функцією **StrCopy(*Dest,Source*)**, яка копіює стрічку ***Source*** (типу PChar) у стрічку ***Dest*** (типу PChar) і встановлює вказівник на ***Dest***. Перевірка довжини отримуваної стрічки не проводиться. Необхідно переконатись, що в буфері стрічки ***Dest*** достатньо місця для символів (хоча б **StrLen(*Source*)+1**). Функція

StrCopy видає результат типу PChar, який не впливає на хід програми, тому дану функцію можна використовувати як процедуру.

Ще одна функція для копіювання стрічок типу PChar - це функція **StrECopy(*Dest,Source*)**. Дана функція виконує ті ж дії, що й функція **StrCopy** але встановлює вказівник не на ***Dest***, а на останній нульовий елемент стрічки. Завдяки цій можливості, дану функцію можна використовувати для склеювання стрічок. Наприклад,

```
StrECopy(StrECopy(StrECopy(S,S1),S2),S3);.
```

Об'єднання двох стрічок виконується також функцією **StrCat(*Dest,Source*)**. Дана функція приєднує копію стрічки ***Source*** (типу PChar) до стрічки ***Dest*** (типу PChar) і встановлює вказівник на ***Dest***. При цьому перевірка довжини отримуваної стрічки не проводиться. Необхідно переконатись, що в буфері стрічки ***Dest*** достатньо місця для символів (хоча б **StrLen(*Source*)+StrLen(*Dest*)+1**).

Для перевірки довжини стрічки можна скористатись функція **StrLCat(*Dest,Source,N*)**. Ця функція робить те ж саме, що й **StrCat**, але при цьому гарантується, що стрічка ***Dest*** буде містити не більше ***N*** (типу Word) символів. Для визначення параметра ***N*** можна скористатись функцією **SizeOf(*Dest*)**, яка визначає розмір вільної пам'яті для символів стрічки ***Dest***.

Для копіювання стрічок використовується також функція **StrLCopy(*Dest*,Source,N)**, яка копіює із стрічки ***Source*** (типу PChar) у стрічку ***Dest*** (типу PChar) ***N*** символів, де ***N*** - змінна типу Word. При цьому вказівник встановлюється на ***Dest***. Якщо довжина копійованої стрічки менша числа ***N***, то копіюється фактична кількість символів цієї стрічки. Функція **StrLCopy** видає результат типу PChar, який не впливає на хід програми, тому, як і попередні функції, її можна використовувати як процедуру. Для встановлення параметра ***N***, в даній функції, також можна скористатись функцією **SizeOf** (див. вище).

Функція **StrMove(*Dest*,Source,N)** копіює із стрічки ***Source*** (типу PChar) у стрічку ***Dest*** (типу PChar) ***N*** (типу Word) символів, навіть якщо це число більше довжини стрічки ***Source***, і при цьому встановлює вказівник на ***Dest***. При цьому стрічки можуть перекриватись в пам'яті.

Для порівняння двох стрічок використовуються функції двох типів: функції, що розрізняють великі та малі літери, та функції, що не розрізняють ці літери. До функцій, що розрізняють великі та малі літери, відносяться функції:

N **:= StrComp(*S1,S2*);** та ***N*** **:= StrLComp(*S1,S2,M*);**.

Якщо перша функція порівнює повністю дві стрічки ***S1*** та ***S2*** типу PChar, то друга лише перших ***M*** (типу Word) символів. Результат роботи цих функцій - це змінна ***N*** типу Integer, яка показує скільки перших (від лівого краю стрічок) символів, що розміщені поряд, у стрічках одинакові.

До функцій, що не розрізняють великих та малих літер відносяться дві функції:

N **:= StrIComp(*S1,S2*);** та ***N*** **:= StrLIComp(*S1,S2,M*);**.

Дані функції діють аналогічно відповідним вищеописаним, але не розрізняють великих та малих латинських літер.

Пошук першого входження стрічки ***PS*** (типу PChar) у стрічку ***S*** (типу PChar) виконується функцією ***P*** **:= StrPos(*S,PS*)**, яка встановлює вказівник ***P*** (типу PChar) у

стрічці ***S*** на перше входження стрічки ***PS***. Якщо такого входження немає, то вказівник встановлюється на **Nill**.

Для пошуку входження символів у стрічку використовуються дві функції:

P **:= StrScan(*S*,*C*);** та ***P*** **:= StrRScan(*S*,*C*);**.

Функція **StrScan** шукає перше входження символу ***C*** (типу Char) у стрічку ***S*** (типу PChar), а функція **StrRScan** останнє. При цьому завершуючий символ Null (#0) також вважається частиною стрічки. Результат роботи цих функцій (змінна ***P***) аналогічний результату роботи функції **StrPos**.

Ще одна функція ***P*****:=StrEnd(S)** модуля **Strings** встановлює вказівник ***P*** (змінна типу PChar) на кінець стрічки ***S*** (типу PChar).

Для того, щоб стрічку типу PChar розмістити в динамічній пам'яті, використовується функція **StrNew(*S*)**, яка створює копію стрічки ***S*** (типу PChar) і поміщає її у динамічну пам'ять із значенням вказівника ***P*** (типу PChar). Якщо ***S*** дорівнює **Nill** або вказує на нульову стрічку, то **StrNew** встановлює **Nill** і не виділяє пам'яті.

Знищити стрічку ***S***, типу PChar, із динамічної пам'яті можна з допомогою функції **StrDispose(*S*)**. Якщо ***S*** дорівнює **Nill**, то **StrDispose** не виконує ніяких дій.

Потрібно зауважити, що функція **StrDispose**, хоча й оголошена як функція, але не встановлює ніякого результату і фактично є процедурою. Тому її необхідно використовувати з директивою **{$X+}**, яка дозволяє розширений синтаксис.

14.5. Модуль DOS

Модуль DOS містить ряд підпрограм для роботи з файлами і доступу до засобів операційної системи.

Процедури і функції модуля Dos можна умовно розбити на наступні групи:

- функції керування операційним середовищем;
- процедури керування процесами;
- процедури обслуговування переривань;
- процедури роботи з датою та часом;
- процедури та функції перевірки стану диску;
- процедури та функції роботи з файлами;
- процедури та функції різного призначення.

Розглянемо ці типи підпрограм більш детально.

Функції керування операційним середовищем

Програма, написана на мові Turbo Pascal 7.0, має можливість отримати від MS-DOS інформацію про оточення (**environment**) ОС. Дані про оточення зберігаються у спеціальній ділянці операційної системи і представляють собою набір текстових рядків, в яких записані змінні оточення. Значення певних змінних встановлюється ОС по замовчуванню, хоча значення багатьох із них встановлюється користувачем або певною

прикладною програмою. Кожне значення, записане в ділянці оточення, представляє собою рядок типу:

Ім'я = Значення,

де **Ім'я** - це ім'я відповідної змінної, а **значення** - текстовий рядок, що встановлює значення цієї змінної.

Змінні оточення призначені для зберігання певної системної інформації, яка необхідна для різних прикладних програм. Наприклад, змінна **COMSPEC** при завантаженні MS-DOS встановлює шлях до файлу командного процесора (COMMAND.COM). По замовчуванню цей шлях має вид C:\COMMAND.COM, але якщо командний процесор розміщений в іншому каталозі, то необхідно точно вказати шлях до нього. Інша змінна - PATH - зберігає список каталогів, в яких буде відбуватись пошук вказаного в командному рядку виконуючого файла. Якщо ця змінна в якості значення зберігає порожню стрічку, то будь-який файл, ім'я якого вводиться в командному рядку, почне виконуватись тільки в тому випадку, якщо він знаходиться в активному каталозі або якщо точно заданий шлях до нього. Значення змінної PATH не встановлюється автоматично, тому її необхідно встановити в файлі AUTOEXEC.BAT.

Потрібно зауважити, що довжина текстової стрічки, яка встановлює змінну оточення, не може перевищувати 128 байт.

Для того, щоб встановити змінну оточення, у MS-DOS використовується команда SET, наприклад:

```
SET COMSPEC=C:\DOS\COMMAND.COM
```

Якщо вказати команду SET без аргументу, то на екран буде виданий список всіх змінних із їх поточними значенням.

В модулі DOS встановлено три функції для роботи із змінними оточення - це **EnvCount**, **EnvStr** та **GetEnv**.

Функція **EnvCount** не має параметрів і видає загальну кількість змінних оточення (результат типу integer), які в даний момент встановлені в ОС. Функція **EnvStr(*Index*)** виводить стрічку (результат типу String), яка містить ім'я і значення змінної, яка відповідає вказаному номеру (змінна ***Index*** типу integer). Нумерація змінних починається з 1. Якщо вказаний індекс менший 1 чи більше значення **EnvCount**, то функція **EnvStr** виводить порожню стрічку.

Функція **GetEnv(*EnvVar*)** дозволяє за іменем змінної оточення EnvVar (типу String) отримати її значення (результат типу String). Наприклад, якщо вказати оператор writeln(GetEnv ('PATH')), то на екрані з'явиться повідомлення типу: C:\;C:\DOS;\C:\WINDOWS;D:\ARHIV;.

Нижче наведений приклад використання функцій керування операційним середовищем.

```
{використання функцій EnvCount і EnvStr}
uses DOS;
var
        I : integer;
begin
```

```
        for  I:=1 to EnvCount do
        writeln(I,'-та змінна - ', EnvStr (I));
        readln
end.
```

Результат роботи даної програми може виглядати приблизно так:

```
COMSPEC = C:\COMMAND.COM
PATH = C:\NU;C:\;C:\WINDOWS;
SYMANTEC = C:\SYMANTEC
NU = C:\NU
```

Як видно із прикладу, в системі встановлено чотири змінних, з яких дві останні використовуються програмами з пакету Norton Utilites.

Процедури керування процесами

Використовуючи спеціальні засоби Turbo Pascal, можна викликати із однієї програми іншу, яку називають програмою-нащадком. Важливо пам'ятати, що для того, щоб програма-нащадок успішно завантажилась в пам'ять і почала працювати, потрібно забезпечити їй потрібну кількість оперативної пам'яті. Так як програма, яка виконується в даний момент, по замовчуванню захоплює всю вільну динамічну пам'ять системи, то для завантаження програми-нащадка вже не залишається місця в пам'яті. Для того, щоб виділити для програми-нащадка потрібну кількість пам'яті, необхідно на початку програми вказати (з допомогою директиви компілятора **{$M}**) мінімальний розмір пам'яті, що виділяється програмі.

Наприклад, директива **{SM 16384,0,655360}** встановлюється для програм по замовчуванню. Тут вказано, що максимальний розмір динамічної ділянки пам'яті рівний всій допустимій пам'яті, тому виклик програми-нащадка неможливий. Але, якщо вказати директиву **{SM 1024,0,0}**, то динамічна пам'ять зовсім не виділяється. В такому випадку вже не можна викликати програму-нащадка. Виклик проходить за допомогою процедури **Exec(*Path,CmdLine*)**, де параметр ***Path*** (типу String) - це повний шлях до виконуючого файлу і його ім'я, а ***CmdLine*** (типу String) - параметр, в якому можна передати викликаній програмі параметри командного рядка. Результат роботи даної процедури передається системній змінній **DosError**, яка описана в модулі DOS. Якщо ця змінна рівна 0, то це значить, що виклик пройшов успішно. Ненульове значення свідчить про помилку (див. таблицю 14.10).

Коли викликається програма-нащадок, бажано, щоб ця програма виконувалась в середовищі DOS, відповідаючи тому оточенню, яке забезпечує операційна система. Однак програма, написана на Turbo Pascal, зазвичай не виконує цих вимог. При завантаженні вона завжди наперед визначає деякі переривання для особистих цілей. Тому, перед викликом програми-нащадка, потрібно встановити вихідні адреси обробників переривань (MS-DOS обробників), а, після закінчення роботи викликаної програми, встановити адреси обробників Turbo Pascal. Ці дії виконує процедура **SwapVectors**, яка задається без параметрів.

Таблиця 14.10

Значення змінної DosError

Код помилки	Значення
0	Нормальне завершення
2	Файл не знайдено
3	Шлях не знайдено
4	Дуже багато відкритих файлів
5	Доступ закритий
6	Пошкоджена інформація в полях файлу чи системних областях
8	Недостатньо пам'яті
10	Несумісні параметри оточення
11	Нерозпізнаний формат диску
18	Немає більше файлів (в роботі з процедурою FindNext)

Використання процедур **Exec** і **SwapVectors** можна продемонструвати на наступному прикладі:

```
{$M 1024,0,0} {звільнення пам'яті для нащадка}
uses DOS;
var
        ProgName,CmdLine : STRING;
begin
        write(`Вводимо шлях та ім''я виконуючого файла - `);
        readln(ProgName);
        write(`Вводимо командну стрічку - `);
        readln(CmdLine);
        SwapVectors;     {Перевстановленя векторів переривань}
        Exes(ProgName, CmdLine);
        SwapVectors;    {Відновлення векторів переривань}
        if DosError <> 0
                then  {Перевірка на помилку завантаження}
                        writeln(`Помилка Dos `, DosError)
                else
                        writeln(`Виконання пройшло успішно`);
        readln
end.
```

В даному прикладі програма буде займати невеликий об'єм оперативної пам'яті, що дозволить успішно завантажувати і виконувати зовнішні програми. Так як наперед не відомо, які програми будуть завантажуватись процедурою Exec, то в програмі використовується **SwapVectors**.

Використовуючи механізм виклику самостійних програм, можна створити складні програмні комплекси, в яких програма-диспетчер, займаючи невеликий об'єм пам'яті, здійснює виклик необхідних (в тому числі системних) програм. Наприклад, широко відома системна оболонка NC - це звичайна програма, яка використовує механізм виклику інших програм. В такому випадку може виникнути необхідність передачі інформації з дочірньої програми в викликаючу програму або навпаки. На жаль, можливості MS-DOS по здійсненню таких обмінів невеликі. Наприклад, можна організувати передачу інформації через командний рядок DOS.

Потрібно зауважити, що якщо викликаюча програма написана на мові Turbo Pascal, то доступ до її командного рядка можна здійснити з допомогою системних функцій **ParamStr** і **ParamCount**.

Вказаний вище метод передачі параметрів підходить і у випадку коли потрібно передати відносно невеликий об'єм інформації. Адже розміри командного рядка не можуть перевищувати 127 байт. Але можна організувати обмін даними і в пам'яті, що більш ефективно. Для цього необхідно в командному рядку передати команді-нащадку сегментну адресу блоку пам'яті, де розміщені необхідні параметри. На жаль, в Turbo Pascal немає наявних процедур для проведення даної операції, тому, якщо виникає потреба, приходиться використовувати процедуру MS-DOS, про яку буде описано пізніше.

Стандартним способом передачі інформації в дочірню програму є процедура **HALT**, яка має один параметр і через нього передає код завершення, котрий, в свою чергу, може бути використаним в дочірній програмі за допомогою функції **DosExitCode**, яка виводить результат -число word, в молодшому байті якого міститься код повернення, а в старшому - ознака того, як завершилась програма. Дана функція може приймати результати: 0 - нормальне завершення програми, 1 - достроково завершена натискуванням комбінації клавіш Ctrl+Break (Ctrl+C), 2 - достроково завершена через помилку пристрою, 3 - завершена процедурою **Keep**.

Необхідно нагадати про ще одну можливість організації передачі інформації між різними програмами, яка, в основному, використовується для зв'язку з резидентною програмою (резидентні програми будуть розглянуті далі). Для обміну інформації використовується область міжзадачних зв'язків, яка є частиною ділянки даних в пам'яті BIOS і розташована в діапазоні адрес $40:F0-$40:FF. Ця ділянка не використовується системою і призначена спеціально для обміну між програмами користувача. Звичайно, в 16 байтах неможливо помістити великий об'єм інформації, тому зручніше передавати не параметри, а їх адреси.

Процедури обслуговування переривань

Незважаючи на те, що Turbo Pascal має великий набір засобів для використання можливостей MS-DOS, в багатьох випадках виникає необхідність прямого звертання до підпрограм ОС щоб використати деякі специфічні функції операційної системи (наприклад, для організації обміну інформацією в оперативній пам'яті). Ці підпрограми відомі користувачам ПК під назвою **переривання (Interrupt)**.

Для роботи з перериваннями в Turbo Pascal використовуються дві процедури: **Intr** та **MsDos**. Процедура **Intr(*IntNo*, *Regs*)** виконує вказане програмою переривання. Параметр ***IntNo*** (типу byte), що використовується в даній процедурі, вказує номер програмного переривання, а ***Regs*** - змінна типу **REGISTERS**. Даний тип є типом запису, що визначений в модулі DOS, як:

```
type
       REGISTERS = record
       case integer of
             0 : (AX, BX, CX, DX, DX, BP, SI, SI, DS, ES, Flags : WORD);
             1 : (AL, AH, BL, BH, CL, CH, DL, DH : BYTE);
end;
```

Поля даного запису відповідають реєстрам процесора. Коли виконується процедура **Intr**, значення полів **AX**, **BX**, **CX**, **DX**, **DX**, **BP**, **SI**, **SI**, **DS**, **ES** завантажується у відповідні реєстри процесора, а тоді викликається переривання з номером ***IntNo***. Після того, як переривання оброблене, склад реєстрів процесора завантажується в запис ***Regs***, а склад реєстру флагів копіюється в полі ***Flags***. Таким чином, можна дістати доступ до результатів роботи процедури переривання.

Для застосування процедури **Intr** необхідні знання деяких тонкощів роботи операційної системи та добрі знання семантики, параметрів і результатів роботи викликаного переривання.

Процедура **MsDOS(*Regs*)**, де ***Regs*** - змінна типу REGISTERS, виконує виклик переривання з номером $21, яке називається перериванням DOS і об'єднує в себе велику кількість різноманітних функцій. Отже, результат звертання до процедури **MsDos** буде той же, що і при звертанні до процедури **Intr** з номером переривання $21.

Досить часто трапляється ситуація, коли в програмі необхідно визначити власні алгоритми реакції на переривання операційної системи. При цьому потрібно відмінити стандартні реакції, або зробити так, щоб виконувалась і стандартна реакція, і своя власна. Для цього Turbo Pascal дозволяє створювати процедури спеціального типу - обробники переривань. Заголовок таких процедур, повинен мати стандартний вигляд:

procedure ім'я(Flags,CS,IP,AX,BX,CX,DX,SI, DI, DS, ES, BP);.

При активізації процедури обробки переривань в стеці автоматично зберігається вміст всіх реєстрів, а процедурі передаються копії вмісту тих реєстрів, які вказані в якості формальних параметрів. Тому в процедурі їх можна змінювати і використовувати. Порядок слідування параметрів повинен точно співпадати з вище наведеним, але необхідно вказувати тільки необхідні з них.

Для встановлення нової адреси обробника переривань використовується процедура **SetIntVec(*IntNo,Vector*)**, де ***IntNo*** (змінна типу byte) - номер переривання, яке необхідно перевизначити; ***Vector*** (типу pointer) - адреса нової процедури обробки переривань. Для того, щоб змінити реакцію системи на переривання і визначити користувацький обробник, **SetIntVec** просто змінює запис в системній ділянці DOS, яка називається ділянкою векторів переривань, та при цьому старий вектор, тобто адреса старої процедури обробки переривань, не зберігається. Він може бути необхідним, щоб через деякий час повернути стару програму переривань, або щоб при виклику переривання спочатку спрацював новий обробник, який потім передає керування оригінальному (старому) обробнику. Цей принцип використовує більшість резидентних програм. Таким чином, роботоздатність системи не порушується, коли декілька програм перехоплюють одне переривання. Резидентна програма спочатку спрацьовує сама, а потім передає керування іншій програмі і так далі, а в кінці керувння передається операційній системі. Тому дуже важливо зберегти адресу старого обробника переривань. Для цього можна використовувати процедуру **GetIntVec(*IntNo,Vector*)**, яка присвоює параметру-змінній ***Vector*** адресу поточного обробника переривань, номер якого заданий в параметрі ***IntNo***.

Резидентні програми

Існує, ще одна процедура, що відноситься до процедур керування процесами (див. вище). Це процедура **Keep(*ExitCode*:word)**. Виклик цієї процедури приводить до завершення роботи програми, але при цьому залишає її в пам'яті. Такі програми носять назви **резидентних в пам'яті** (**Terminate and Stay Resident, TSR**), або просто **резидентних програм**. На цьому принципі побудовані драйвери пристроїв і різні сервісні програми.

Зробити резидентною можна як програму типу „.com", так і програму типу „.exe", але у зв'язку з відмінностями у внутрішній структурі програми типу „.com" займають в пам'яті менше місця, тому резидентні програми найчастіше реалізують в цьому форматі. Після того, як програма залишилась резидентною в пам'яті, вона передає керування командному процесору, а сама ніби „завмирає". Для того щоб активізуватись в потрібний момент, програма повинна перехопити певне переривання, наприклад, переривання клавіатури. Після натискання потрібної комбінації клавіш програма перехоплює керування і виконує свої функції.

Процедури роботи з датою і часом

Модуль DOS дає програмісту можливість доступу до системного часу та дати, а також дозволяє змінювати дату та час створення файлу. Для доступу до системної дати використовують процедури **GetDate** та **SetDate**.

Процедура **GetDate(*Year, Month,Day,Day_of_week*)** виводить поточну дату, що встановлена в системі. Змінні ***Year, Month,Day,Day_of_week*** - це вихідні параметри, які є змінними типу word, і які виводять, відповідно, поточний рік, мсяць, день та день тижня.

Процедура **SetDate (*Year*,Month,Day)** встановлює поточну дату в операційній системі. Змінні ***Year*,Month,Day** - це змінні типу word, які задають значення поточного року, місяця та дня.

Параметр ***Year*** може приймати значення від 1980 до 2099, ***Month*** - значення від 1 до 12, ***Day*** - від 1 до 31, ***Day_of_week*** - від 0 до 6, де 0 встановлює неділю.

Для доступу до системного годинника використовуються процедури **SetTime** і **GetTime**.

Процедура **GetTime(*Hour,Minute,Second,Hung*)** виводить поточний час, що встановлений в системі. Змінні ***Hour,Minute,Second,Hung*** - це змінні типу word, які виводять відповідно годину, хвилину, секунду та соту долю секунди. Значення параметра ***Hund***, яке виводить дана процедура, досить наближене, оскільки в операційній системі перемикання лічильника таймера проходить 18,2 рази в секунду, отже, встановити точно соті долі секунди не можливо.

Процедура **SetTime(*Hour,Minute*,Second,Hung)** задає нове значення поточного часу в операційній системі.

Параметр ***Hour*** може приймати значення від 0 до 23, ***Minute*** - значення від 0 до 59, ***Second*** - від 0 до 59, ***Hund*** - від 0 до 99.

Для роботи з датою створення файла використовується процедура **GetTime(*F,Time*)**, яка виводить час та дату створення файлу. Змінна ***F*** повинна бути

файловою змінною і вказувати на файл, для якого визначаємо дані параметри. Час та дата передається змінній ***Time*** (типу LongInt) в запакованому форматі у вигляді двійкового слова (32 біти). Для її розпакування використовується процедура **UnpackTime(*Time,DT*)**, де ***DT*** - змінна типу **DateTime**, який описаний в модулі DOS наступним чином:

```
type
       DateTime = record
       Year, Month, Day, Hour, Min, Sek: word;
end;
```

Процедури **GetFTime** та **UnpackTime** використовуються для зчитування дати і часу створення файлу. Для того щоб змінити цю величину, використовуються дві інші процедури: **PackTime** та **SetFTime**.

Процедура **PackTime(*T,Time*)** перетворює запис ***T*** типу **DateTime** в двійкове слово ***Time*** (типу LongInt), яке пізніше використовується процедурою **SetFTime**.

Процедура **SetFTime(*F,Time*)** встановлює дату створення файла ***F*** (файловий тип даних). Про успішність операції можна дізнатися із змінної **DosError**, описаної в модулі Dos. У випадку успішного завершення **DosError** буде містити 0. У випадку невдачі код помилки може бути тільки 6 (зруйнований заголовок файла). Також потрібно пам'ятати, що файл, для якого необхідно змінити дату створення, повинен бути відкритий для читання. Якщо файл відкритий процедурою Rewrite, то після зміни дати створення файлу (коли в кінці роботи файл буде закритий процедурою Close), вона автоматично оновлюється, тобто встановлюється поточна системна дата.

Операії зчитування і зміни дати створення файла рідко використовується при написанні програм. Необхідність в них може виникнути, наприклад, при створенні програм, що контролюють файли на зараження вірусами, або при створенні резервних копій файлів, які були створені останнім часом.

Процедури та функції перевірки стану диска

Модуль Dos містить функції та процедури для роботи з диском. З їх допомогою можна здійснити контроль за об'ємом вільного місця на диску, контроль правильності запису на диску, а також визначати загальний об'єм пам'яті диску.

Функція **X := DiskFree(*Drive*)** встановлює кількість байтів вільного місця на вказаному диску (де ***Drive*** - змінна типу byte, яка задає номер диску (0 - поточний диск, 1 - дисковод „A", 2 - дисковод „B", 3 - диск „C", 4 - диск „D" і т.д.), а ***X*** (типу LongInt) - кількість байтів вільного місця на цьому диску.

Якщо номер диску заданий некоректно, то функція повертає значення 1. таким чином, завжди є можливість проконтролювати виклик функції.

Функція **X := DiskSize(*Drive*)** встановлює загальний об'єм вказаного диску, де ***Drive*** (типу byte) задає номер диску(аналогічно попередній функції), а ***X*** (типу LongInt) - загальний об'єм цього диску в байтах.

Нижче приведений простий приклад з використанням цих двох функцій.

```
{Використання функцій DiskFree і DiskSize}
uses DOS;
begin
       writeln (`Об'єм пам'яті вашого диску`,
       (DiskSize(0) DIV 1024), `Kb`);
       writeln (`На диску вільно`, (DiskFree(0) DIV 1024), `Kb`);
end.
```

Процедура **GetVerify(*Ver*)** використовується для визначення поточного значення режиму перевірки стану DOS, що має ім'я **VERIFY (верифікація інформації)**. Цей режим служить для визначення методу запису даних на диск. Якщо **VERIFY** знаходиться у ввімкненому стані, рівне **ON**, то після того, як інформація записана на диск, вона знову читається і звіряється з оригіналом. Якщо ж режим знаходиться у вимкненому стані, тобто **OFF**, то інформація записується на диск без перевірки. Змінна ***Ver*** (типу Boolean) може приймати два значення **TRUE** значення режиму **ON** або **FALSE** значення режиму **OFF**.

Для того, щоб змінити значення режиму VERIFY використовується процедура **SetVerify(*Ver*)**, яка встановлює (***Ver*=TRUE**) або відміняє (***Ver*=FALSE**) режим верифікації інформації.

Процедури і функції для роботи з файлами

При роботі з файлами часто виникає необхідність виконувати операції пошуку файлів з одинаковим іменем, пошуку по шаблону, виділення потрібного файла із знайденої групи файлів. Для виконання цих операцій в модулі DOS можна використати процедури **FSplit**, **FExpand**, **FSearch**, **FindFirst**, **FindNext**.

Таблиця 14.11

Константи атрибутів

Назва	Значення
ReadOnly	$01
Hidden	$02
SysFile	$04
VolymeID	$08
Directory	$10
Archive	$20
AnyFile	$3F

Процедура **FindFirst(*Path,Attr,S*)** здіснює пошук файлу із заданим іменем і шляхом до нього (змінна ***Path*** типу string) та набором атрибутів (змінна ***Attr*** типу Word, значення якої предстaлено в таблиці 14.11). Коли в параметрі ***Path*** не вказаний шлях, а лише ім'я файлу або шаблон, то пошук буде проведений у поточному каталозі.

Результатом роботи цієї процедури буде змінна ***S*** типу **SearchRec**, який описаний в модулі DOS, як запис:

```
type
       SearchRec = record
       Fill : array[1...21] of byte;
       Attr : byte;
       Time : LongInt:;
       Size : LongInt;
       Name : string[12];
       end;
```

В даному записі поле **Fill** містить службову інформацію MS-DOS і не повинно модифікуватися. Поле **Attr** містить атрибути файла, які визначаються константами атрибутів (таблиця 14.11). Поле **Time** містить дату і час створення файла в запакованому вигляді (для розпакування необхідно використовувати процедуру **UnpackTime**). Наступне поле **Size** містить розмір файла в байтах. В останньому полі **Name** містить ім'я знайденого файла.

Процедура **FindNext(*S*)** шукає наступний файл з атрибутами, заданими при попередньому виклику процедури **FindFirst** (коли ім'я було задано шаблоном). Результатом роботи даної процедури буде також змінна типу **SearchRec**. Коли **FindNext** не знаходить більше файлів, то в системній змінній **DosError** встановлюється значення 18.

Нижче вказано приклад, що демонструє використання процедур пошуку файлів.

```
{Використання процедур FindFirst і FindNext}
uses DOS;
var
        DT: DateTime;
        S  : SearchRec;
function FirstZero(D : word) : string;
var
        T : string;
        begin
                Str(D:0,T);
                if Length(T) = 1
                then  T:= '0' + T;
                FirstZero:= T
        end;
begin
        WriteLn;
        FindFirst('*.exe',AnyFile,S);
        while DosError = 0 do
        begin
                UnpackTime(S,Time,DT);
                Write(S.Name);
                Write(' ',FirstZero(DT.Hour), ':', FirstZero(DT.Min),
                        ':', FirstZero(DT.Sec));
                Writeln(' ',FirstZero(DT.Day),':',FirstZero(VDT.Month),
                        ':', DT.Year);
                FindNext(S);
        end;
        readln;
end.
```

Наведена програма подібна на команду MS-DOS dir. Вданому випадку вона виводить на екрані імена всіх файлів з розширенням ".exe". Ці значення, при допомозі процедури **UnpackTime,** викликаються з поля Time змінною S типу **SearchRec**.

Як вже відмічалось процедури **FindFirst** і **FindNext** при пошуку файла або групи файлів переглядають тільки вказпний каталог. Якщо ж виникає необхідність пошуку в різних каталогах, то використовується функція **FSearch(*Path,DirList*)**, де ***Path*** (типу String) - шлях і ім'я файла, який необхідно знайти, а ***DirList*** (типу **PathStr**) - це список каталогів, в яких буде проводитись пошук. Тип **PathStr** описаний в модулі DOS, як стрічковий тип String[79]. Каталоги у списку повинні бути розділені крапкою з комою аналогічно тому, як це робиться у команді MS-DOS PATH.

Пошук завжди починається з активного каталога поточного диску. У випадку успішного пошуку функція видає стрічку (типу **PathStr**), яка містить повне ім'я файлу (шлях і ім'я). У випадку невдачі виводиться порожня стрічка.

Для роботи з іменами файлів використовується також процедура **FSplit(*Path,Dir,Name,Ext*)**, яка розбиває повний шлях до файлу ***Path*** (типу **PathStr**) на три компоненти: ***Dir*** (типу **DirStr**) - шлях до файлу, ***Name*** (типу **NameStr**) - ім'я файлу та ***Ext*** (типу **ExtStr**) - розширення. Типи **DirStr**, **NameStr** та **ExtStr** описані в модулі DOS наступним чином:

```
type
        DirStr = STRING[67];
        NameStr = STRING[8];
        ExtStr = STRING[4];
```

Функція **Expand(Path)** виконує дії, протилежні до поцедури **FSplit**. Вона доповнює ім'я файла ***Path*** (типу **PathStr**) до повного імені. Результат роботи функції, також, змінна типу **PathStr**.

Для зміни атрибутів файла призначені дві процедури - **GetFAttr** та **SetFAttr**, включені у модуль DOS. Процедура **GetFAttr(*F,Attr*)** виводить атрибути файла, зв'язаного із змінною ***F*** (файлового типу), присвоюючи їх змінній ***Attr***. Константи атрибутів представлені в таблиці 14.11.

Коли необхідно встановити атрибут на файл, то можна скористатись процедурою **SetFAttr(*F,Attr*)**. При чому, якщо потрібно встановити одночасно декілька атрибутів, то їх записують через знак "+", наприклад, SetFAttr(F,Hidden+ReadOnly).

У випадку помилки при роботі процедур **SetFAttr** або **GetFAttr** в змінну **DosError** вноситься код помилки. Значення коду помилки може бути 3 (неправильний шлях) або 5 (немає доступу).

Для роботи з файлами призначені два типи, визначених в модулі DOS - це **FileRec** і **TextRec**. Тип **FileRec** використовується самим Turbo Pascal і визначає внутрішній формат типізованих і нетипізованих файлів. Він визначається наступним чином:

```
type
        FileRec = record
        Handle : word;
        Mode : word;
        RecSize : word;
        Private : array[1..26] of byte;
        UserDate : array[1..16] of byte;
        Name : array[0..79] of char;
        end;
```

де **Handle** - дескриптор файлу - унікальне число, присвоєне файлу операційною системою; **Mode** - стан файла (закритий - **fmClosed** або **$D7B0**, відкритий для запису - **fmInput** або $**D7B1**, відкритий для читання - **fmOutput** або **$D7B2**), відкритий для читання і запису -**fmLnOut** або $**D7B3**), **RecSize** - довжина запису в байтах, **Private** - зарезервована ділянка, **UserData** - ділянка у яку вноситься користувацька інформація, **Name** - повне ім'я файла, який закінчується символом #0.

Тип **TextRec** також використовується в Turbo Pascal. Він призначений для визначення внутрішнього формату текстових файлів . Він показаний наступним чином:

```
type
        TextBuf = array[1..127] of char;
        TextRec = record
        Handle : word;
        Mode : word;
        BufSize : word;
        Private : word;
        BufPos : word;
        BufEnd : word;
        BufPtr : ^TextBuf;
        OpenFunc : pointer;
        InOutFunc : pointer;
        FlushFunc : pointer;
        CloseFunc : pointer;
        UserDate : array[1..16] of byte;
        Name : array [1..79] of char;
        Buffer : TextBuf;
        end;
```

де **BufSize** - розмір буфера текстового файла в байтах; **BufPos** - позиція поточного символа в буфері текстового файла; **BufEnd** - загальне число символів, записане у буфері; **BufPtr** - вказує на буфер текстового файла; **CloseFunc** - вказівник на підпрограми керування текстовим файлом; **Buffer** - буфер текстового файла. Призначення інших полів аналогічне відповідним поля для типу **FileRec**.

Інші процедури та функції

У модулі Dos є декілька процедур і функцій, які не відносяться ні до одного із розглянутих нами розділів. До них відноситься функція **DosVersion** та процедури **GetCBreak** і **SetCBreak**.

Функція **DosVersion** виводить номер версії операційної системи. Результат роботи цієї функції - це змінна типу word, в старшому байті якої міститься цілий розділ номеру версії, (наприклад 6), а в молодшому байті - дробовий розділ номера (наприклад 22), в результаті чого отримуємо 6.22).

Процедуири **GetCBreak** і **SetCBreak** призначені для роботи із змінною MS-DOS **BREAK**. Ця змінна відповідає за реакцію системи на натискування комбінації клавіш Ctrl+Break. Якщо **BREAK** рівна **ON**, то переривання відбувається при будь-яких системних викликах. Процедура **GetCBreak(*BR*)** визначає значення системної змінної MS-DOS **BREAK** і присвоює його змінній ***BR***, логічного типу. Прцедура **SetCBreak(*BR*)** встановлює системну змінну MS-DOS BREAK в стан, що задається змінною ***BR***.

14.6. Модуль WinDOS

Модуль WinDOS, як і модуль DOS, містить ряд підпрограм для роботи із файлами й доступу до засобів ОС. Крім цього модуль **WinDOS** передбачає роботу зі стрічками із завершуючим нулем. Цей модуль рекомендують використовувати, коли в програмі використовуються стрічки із завершуючим нулем або необхідно надалі використовувати дану програму у середовищі Windows.

Процедури та функції модуля WinDOS можна умовно розбити на наступні групи:

- функції керування операційним середовищем;
- процедури обслуговування переривань;
- процедури роботи з датою та часом;
- процедури та функції перевірки стану диска;
- процедури та функції для роботи з каталогами;
- процедури та функції для роботи з файлами;
- процедури та функції різного призначення.

Функції керування операційним середовищем

До функцій керування операційним середовищем у WinDOS відноситься три функції: **GetArgCount**, **GetArgStr**, **GetEnvVar**.

Функція **GetEnvVar(*VarName*)** аналогічна функції **GetEnv** модуля DOS, тобто по імені змінної оточення встановлює її значення. Відмінність цієї функції, в порівнянні з **GetEnv**, полягає в тому, що аргумент ***VarName*** стрічкового типу замінено типом PChar. Результат роботи цієї функції, також, змінна типу PChar. Дана функція встановлює вказівник на відповідну змінну середовища.

Наступна функція **N:=GetArgCount** (не містить аргументів) встановлює кількість параметрів (число ***N*** типу integer), що передаються програмі в командній стрічці.

Третя функція керування операційним середовищем ***P*:=GetArgStr(*Dest*,*Index*,*N*)** виводить текстову стрічку ***P*** (типу PChar), яка містить значення параметра з номером ***Index*** (типу integer) із командної стрічки ***Dest*** представленої типом PChar. Аргумент ***N*** (типу Word) встановлює максимальну довжину командної стрічки ***Dest***. Якщо ***Index*** рівний 0, то змінній ***P*** присвоюється ім'я програми, а коли ***Index*** від'ємний або більший ***N***, то записується порожня стрічка.

Наступний приклад демонструє можливість роботи з параметрами командної стрічки:

```
uses  WinDOS, Strings;
var
       I : integer;
       S : array[0..79] of char;
begin
       writeln('Кількість параметрів командної стрічки ', GetArgCount+1);
       {виведення списку параметрів}
       for  I := to GetArgCount  do
       writeln(GetArgStr(S,I,79);
       readln
end.
```

Процедури обслуговування переривань

Потрібно відмітити, що процедури керування процесами **Exec**, **Keep** і **SwapVectors**, які встановлені в модулі DOS, у модулі WinDOS не визначені. Процедури **GetIntVector** та **SetIntVector** мають синтаксис запису, який повністю відповідає синтаксису однойменних процедур модуля DOS. Нагадаємо, що перша з них зберігає адресу старого вектора переривань, а друга - встановлює адресу нового вектора переривань.

Синтаксис запису двох інших процедур для обслуговування переривань **Intr** та **MsDos** відрізняється лише тим, що тип **REGISTERS** у модулі WinDOS замінено на **TREGISTERS**, тобто вони мають такий формат:

Intr(IntNo : byte, var Regs : TREGISTERS);, **MsDos(var Regs:TREGISTERS);**

Нагадаємо, що перша з них виконує задане програмне переривання, а інша - викликає відповідну функцію DOS.

Тип даних **TREGISTERS** описується наступним чином:

```
type
       TREGISTERS = record
       case   integer  of
              0 : (AX,BX,CX,DX,BP,SI,SI,DS,ES,Flags : word);
              1 : (AL,AH,BL,BH,CL,CH,DL,DH : byte);
       end;
```

Значення всіх полів запису TREGISTERS, як і в REGISTERS відповідає відповідним реєстрам та флагам процесора.

Для доступу до окремих бітів реєстру флагів процесора використовуються ті ж константи масок, що й у модулі DOS.

Процедури роботи з датою та часом

Для роботи з датою та часом у модулі WinDOS використовуються наступні процедури: **GetDate**, **GetFTime**, **GetTime**, **PackTime**, **SetDate**, **SetFTime**, **SetTime**, **UnpackTime**.

Формат і призначення даних процедур ті ж самі (див. вище), що й у модулі DOS. Єдина відмінність полягає в тому, що першим параметром процедури **PackTime** повинна бути змінна типу **TDateTime**, яка визначається аналогічно **DateTime** модуля DOS:

```
type
       TdataTime = record
       Year, Month, Day, Hour, Min, Sec : word;
       end;
```

Процедури та функції для перевірки стану диска

В модулі WinDOS всі процедури та функції для перевірки стану диску мають повністю аналогічний формат та призначення, що й відповідні процедури модуля DOS. Нагадаємо, що до цих процедур та функцій належать: **DiskFree**, **DiskSize**, **GetVerify** та **SetVerify**.

Процедури та функції для роботи з каталогами

В модулі WinDOS використовуються процедури та функції для роботи з каталогами, що не використовуються в модулі DOS. Але всі вони відповідають стандартним процедурам та функціям для роботи з каталогами, що описані в модулі **System**.

Розглянемо перелік процедур для роботи з каталогами, що описані в модулі WinDOS:

- процедура **CreateDir(*Dir*)** - створює каталог ***Dir***. Змінна ***Dir*** (типу PChar) представляє собою шлях та ім'я створюваного каталогу, записані за правилами DOS. При виникненні помилки в написанні шляху або імені, код помилки заноситься в змінну **DosError**;
- процедура **RemoveDir(*Dir*)** - знищує каталог ***Dir*** (типу PChar). Дана змінна представляє собою шлях та ім'я відповідного каталогу. При виникненні помилки в написанні шляху або імені код помилки заноситься в змінну **DosError**;
- процедура **SetCurDir(*шлях*)** - встановлює активним каталог, що вказаний по даному шляху (змінна ***шлях*** типу PChar). При виникненні помилки, її код також записується у змінну **DosError**;
- функція **SetCurDir(*Dir,Drive*)** - видає шлях (результат типу PChar) до поточного каталогу диску. Змінна ***Dir*** (типу PChar) - вказівка на стрічку, в якій записано специфікацію поточного каталогу, а ***Drive*** (типу byte) - умовний номер пристрою: 0 - активний диск, 1 - дисковод „A", 2 - дисковод „B" і т.д. При виникненні помилки, її код також записується у змінну **DosError**. Якщо вказано номер неіснуючого диска, то в стрічку, що вказана ***Dir,*** буде записано „X:\".

Процедури та функції для роботи з файлами

Для роботи з файлами в модулі WinDOS встановлені ті ж процедури та функції, що й у модулі DOS, але деякі з них мають інші формати.

Для процедур **GetFAttr(*файл, Attr*)** (отримання атрибутів файлів) та **SetFAttr(*файл, Attr*)** (встановлення атрибутів файлів) параметр ***Attr*** може приймати ті ж значення, але назви констант змінено: **faReadOnly=$01** (в DOS - ReadOnly), **faHidden=$02** (Hidden), **faSysFile=$04** (SysFile), **faVolumeID=$08** (VolumeID), **faDirectory=$10** (Directory), **faArchive=$20** (Archive) та **faAnyFile=$3F** (AnyFile). Параметр ***файл*** - це змінна файлового типу, яка вказує на файл, над яким проводимо операції по обробці атрибутів.

Не змінилась також робота процедур **FindFirst** (пошук першого файлу із заданим іменем та атрибутами) та **FindNext** (пошук наступного файлу із заданим іменем та атрибутами), але замість змінних типу String тут використовуються змінні типу PChar (якщо параметр задається константою, то це не має значення). Крім цього в даних процедурах тип **SearchRec** замінено на **TSearchRec**, який повністю аналогічний попередньому. Тип **TSearchRec** описується наступним чином:

```
type
       TSearchRec = record
       Fill : array[1..21] of byte;
       Attr : byte;
       Time : LongInt;
       Size : LongInt;
       Name : string[0..12] of char;
       end;
```

Замість процедури **FSearch** у модулі WinDOS використовується функція **FileSearch(*Dest,Name,List*)**, формат якої значно змінено. Дана функція здійснює пошук

файлу ***Name*** (типу PChar) у списку каталогів ***List*** (типу PChar). Результат пошуку заноситься у змінну ***Dest*** (типу PChar) - вказівник на стрічку з іменем знайденого файлу. При цьому результатом роботи даної функції буде значення типу PChar, де вказано ім'я файлу з добавленням шляху до нього, якщо файл знайдено, і порожньої стрічки в противному випадку.

Аналогічні зміни відбулись з функцією **FExpand** (назва у модулі DOS). Її назва в модулі WinDOS - **FileExpand(*Dest,Name*)**. Дана функція добавляє до імені файлу ***Name*** (типу PChar) значення повного шляху на диску. Змінній ***Dest*** (типу PChar) присвоюється значення вказівника на стрічку для розміщення повного імені файлу. Результат роботи функції (типу PChar) - вказівник на повний шлях та ім'я даного файлу.

Найбільших змін зазнала процедура **FSplit** (назва у модулі DOS), яка перетворилась у функцію **FileSplit(*Path,Dir,Name,Ext*)**. Дана функція розбиває повне ім'я файлу (із шляхом до нього) на складові: ***Path*** (типу PChar) - вказівник на стрічку, що містить повне ім'я файлу (разом із шляхом); ***Dir*** (типу PChar) - вказівник на стрічку, що містить лише шлях до файлу; ***Name*** (типу PChar) - вказівник на стрічку, що містить лише ім'я файлу; ***Ext*** (типу PChar) - вказівник на стрічку що містить лише розширення файлу.

Результатом роботи функції **FileSplit** є число типу word з ознаками наявності відповідних частин імені файлу і групових операцій (* або ?): $0001 - наявність розширення, $0002 -наявність імені файлу, $0004 - наявність шляху та $0008 - наявність групових операцій.

Процедури та функції різного призначення

До цієї групи процедур та функцій належать функції **GetCBreack** та **SetCBreack**, а також процедура **DosVersion**, які повністю ідентичні відповідним підпрограмам модуля DOS.

14.7. Модуль Overlay

Модуль **Overlay** представляє собою засіб створеня програм розділених на окремі модулі. Як вже відмічалось, розмір модуля в Turbo Pascal не може перевищувати 64K, але кількість модулів не обмежена. Це дозволяє створювати програми, які вимагають великих ресурсів оперативної пам'яті. Але в деяких випадках розмір необхідної оперативної пам'яті може перевищувати її реальні можливості. В цих випадках видається повідомлення **Not enough memory** (не достатньо пам'яті). Якщо програма завантажується із оболонки Turbo Pascal, то частину пам'яті (близько 230K) займає лише середовище Pascal. В такому випадку зменшити дефіцит пам'яті можна використовуючи автономний компілятор TPC.EXE, який запускається з середовища ОС. Якщо й цього виявиться недостатньо, то для вирішення цієї проблеми потрібно використати **оверлеї** (**Overlay**).

Це простий і досить ефективний спосіб, який дозволяє створювати програми практично необмеженого об`єму. Це пояснюється тим, щоб при виконанні програми, побудованої на оверлеях, в пам'яті знаходяться тільки ті із оверлейних процедур та функцій, які необхідні в даний момент.

Оверлейні модулі завантажуються в так званий оверлейний буфер (ділянка пам'яті певного розміру) і використовують його по черзі. Дії по завантаженню оверлеїв в пам'ять та вивантаженню їх на диск виконуються адміністратором оверлеїв. Оверлейний буфер знаходиться в пам'яті між сегментом стека та динамічно розподіленою ділянкою пам'яті. По замовчуванню для оверлейного буфера відводиться мінімально можливий об'єм пам'яті, але під час роботи програми його розмір може бути легко збільшений шляхом виділення додаткової пам'яті із динамічної ділянки пам'яті. При відсутності необхідного об'єму пам'яті виводиться повідомлення про помилку **Program too big to fit in memory** (програма надто велика, щоб розміститись в пам'ять) або **Not enogh memory to run program** (не достатньо пам'яті для завантаження програми).

Однією із найважливіших переваг адміністратора оверлеїв є можливість використання додаткової (**Expanded**) пам'яті, що відповідає стандарту **EMS (Lotus/Intel/ Microsoft Expanded Memory Specification)**. Якщо оверлейний файл розміщений в EMS, то всі наступні дії по завантаженню оверлея здійснюється шляхом швидкої передачі інформації із однієї ділянки пам'яті в іншу.

Потрібно відмітити, що оверлеї корисні лише в програмх DOS, що працюють в реальному режимі, оскільки для програм Windows пам'яттю керує сама ОС, а для програм DOS, що працюють в захищеному режимі - адміністратор виконання (RTM.EXE). Ці засоби включають в себе повний механізм обслуговування, що виконують оверлеї, тому, потреби в використанні оверлеїв в цих режимах роботи програм, не має.

Turbo Pascal керує оверлеями на рівні модулів, які є найменшою частиною програми, що утворює оверлей. При компіляції програми, яка містить оверлеї, поряд із виконуваним файлом (що має розширення EXE) створюється файл з розширенням **OVR**. Файл з розширенням EXE містить статичні (неоверлейні) частини програми, а файл з розширенням OVR - всі оверлейні модулі.

Адміністратор оверлеїв Turbo Pascal реалізований з допомогою стандартного модуля **Overlay**. Адміністратор оверлеїв використовує удосконалені методи керування буферами пам'яті, що забезпечує оптимальне виконання програми в ділянці пам'яті, що є в наявності. Наприклад, підсистема керування оверлеями залишає в оверлейному буфері стільки оверлеїв, скільки можливо. Це дозволяє зменшити кількість зчитувань оверлеїв з диску.

Крім цього, коли в адміністратора оверлеїв виникає необхідність вивантажити один оверлей, щоб звільнити місце для іншого, то він намагається спочатку виватажити ті оверлеї, які в даний момент не активні.

За виключенням деяких правил по оформленню оверлейного модуля, він нічим не відрізняється від звичайного (неоверлейного).

На початку всіх оверлейних модулів повинна бути директива компілятора **{$O+}**, що вказує на те, що цей модуль повинен компілюватись як оверлейний. Але, потрібно відмітити, що вказання даної директиви компілятора не означає, що цей модуль можна використовувати лише як оверлейний. Дана директива лише забезпечує цю можливість. Якщо модуль планується використовувати як в оверлейних, так і в неоверлейних програмах, то його компіляція з директивою **{$O+}** забезпечує використання однієї версії модуля для обидвох випадків.

Іншою умовою для виконання програм, що містять оверлеї, є наявність в процедурах та функціях, що прямо або побічно викликають оверлейні підпрограми, директиви компілятора **{$F+}**. В цій програмі вона вказує на віддалений тип виклику.

На початку оверлейного модуля обов'язково повинні знаходитись директиви **{$O+}** та **{$F+}**, а в основній програмі, в перших рядках (або перед кожною процедурою, що викликає оверлей) потрібно вказати директиву **{$F+}** .

Отже початок програми, що використовує оверлеї повинен мати наступний вигляд:

```
{програма повинна мати віддалений тип виклику}
{$F+}
{порядок підключення оверлейних модулів повинен бути наступним}
uses мод1, мод2, ..., Overlay, ов_мод1, ов_мод2, ...;
{вказуємо компілятору які саме модулі оверлейні}
{$O ов_мод1}
{$O ов_мод2}
... {і так далі}
```

В даному випадку ***мод1, мод2, ...*** позначено неоверлейні модулі, а ***ов_мод1, ов_мод2, ...*** - оверлейні.

Якщо спробувати використати в якості оверлейного модуль, який при компіляції не містив директиви **{$O+}**, то компілятор видасть повідомлення про помилку. Із стандартних модулів Turbo Pascal в якості оверлейного можна використовувати лише модуль DOS. Не можуть бути оверлейними також модулі, які містять обробку переривань.

Розглянемо тепер основні процедури та функції, що використовуються в модулі **Overlay**.

Для того, щоб ініціалізувати адміністратор оверлеїв та відкрити оверлейний файл (з розширенням OVR) необхідно скористатись процедурою **OvrInit(*файл*)**, де ***файл*** (типу String) - стрічка, що вказує ім'я відповідного OVR-файлу.

Дана процедура є обов'язковою для кожної програми, яка використовує оверлеї. Вона повинна виконуватись перед першим звертанням до будь-якої оверлейної підпрограми.

Для контролю за успішністю операції ініціалізації оверлейного файлу можна використати спеціальну змінну із модуля Overlay - **OvrResult** , яка зберігає код завершення поцедур та функцій модуля. Допустимі значення даної змінної та список відповідних їм констант представлені в табл. 14.12.

Таблиця 14.12

Значення змінної OvrResult

Значення	**Константа**	**Опис**
0	OvrOK	Нормальне завершення
-1	OvrError	Помилка керування оверлеями
-2	OvrNotFound	Файл .OVR не знайдено
-3	OvrNoMemory	Не достатньо пам'яті для оверлейного буфера
-4	OvrIOError	Помилка при звертанні до OVR-файла
-5	OvrNoEMSDriver	Не встановлено драйвер EMS
-6	OvrNoEMSMemory	Недостатній розмір EMS-пам'яті

Крім основної пам'яті оверлеї можна розміщувати в додатковій (Expanded) пам'яті. Для цього використовується процедура **OvrInitEMS** (без параметрів). Дана процедура перевіряє можливість використання Expanded пам'яті і, якщо це можливо, то оверлейний

файл повністю розміщується в цій пам'яті. Потрібно відмітити, що дана процедура не замінює процедуру **OvrInit**. При використанні Expanded пам'яті потрібно задавати обидві ці процедури (в порядку заданому в даному посібнику).

Для керування оверлейним буфером в модулі Overlay застосовується функція **OvrGetBuf** та процедури **OvrSetBuf** і **OvrClearBuf**. Використання цих процедур необхідне в тому випадку, коли системні засоби модуля, що керують цим буфером, з певних причин не задовільняють програміста. В цьому випадку виникає необхідність самостійно керувати конфігурацією і вмістом буфера.

Функція **OvrGetBuf,** на відміну від спеціальної зміної **OvrHeapSize** модуля System, видає поточний розмір оверлейного буфера в байтах (результат типу LongInt).

Початковий розмір оверлейного буфера вибирається мінімально можливим, тобто таким, щоб помістити оверлей найбільшого розміру. Для деякий прикладних програм цього може виявитись достатньо, але можливі випадки, коли одна із функцій програми реалізується з допомогою двох чи більше модулів (кожен з яких є оверлейним). Якщо сумарний розмір таких модулів більший, як розмір найбільшого оверлея, то часте звертання модулів один до одного приведе до дуже частого обміну інформацією між диском (де розміщені ці модулі) і пам'ттю, що сповільнить роботу програми.

Очевидно, що вирішення цієї проблеми полягає у збільшені розміру оверлейного буфера, щоб в заданий момент часу було достатньо пам'яті для розміщення в ній всіх оверлеїв, що часто звертаються один до одного.

Процедура **OvrSetBuf(*Size*)** встановлює новий розмір буфера і повинна викликатися після процедур **OvrInit** та **OvrInitEMS**. Параметри ***Size*** (типу LongInt) визначає потрібний розмір буфера в байтах. Цей розмір не повиненн бути меншим розміру буфера, встановленого на початку. Крім цього, цей розмір не повинен перевищувати розмір доступної пам'яті, який визначається різницею значень розміру максимального неперервного блоку динамічної пам'яті (визначається змінною **MaxAvail**, описаною в модулі System) та значення вже згадуваної змінної **OvrHeapSize**, яке містить початковий розмір буфера. Якщо значення параметру ***Size*** перевищує текучий розмір буфера, то об'єм пам'яті, що не вистачає, виділяється з динамічної ділянки пам'яті, а якщо менше, то залишок поміщається як вільний і повертається в цю ділянку.

Після виклику процедури **OvrSetBuf** необхідно перевірити значення змінної **OvrResult**. Якщо її значення рівне **OvrError**, то це означає, що в динамічній ділянці пам'яті вже були розміщені динамічні змінні (з допомогою процедур **New** та **GetMem**) або значення ***Size*** занадто мале для створення буфера (можливо також, що не була встановлена процедура ініціалізації). Найчастіше помилка при використанні процедури **OvrSetBuf** виникає через те, що в програмах TurboPascal по замовчуванню використовується деректива компілятора.**{$M16384,0,655360}**. В цьому випадку виникає помилка **OvrNoMemory**, яка вказує на те, що для збільшення розміру буфера недостатньо пам'яті. В такому випадку необхідно вказати на початку програми директиву **{$M16384,655536,655360}**, яка задає потрібний мінімальний розмір динамічної ділянки.

Найчастіше виділення додаткової пам'яті для буфера виконується процедурою

OvrSetBuf(*OverGetBuF+ExtraSize*);

При використанні модуля Graph необхідно пам'ятати про те, що при ініціалізації графічного режиму виділяється пам'ять для графічних драйверів та шрифтів. Це означає, що при виклику процедури **OvrSetBuf** вміст динамічної ділянки буде визначено як зайнятий. Отже виклик цієї процедури повинен бути виконаний до виклику процедури ініціалізації **InitGraph**, та до виклику процедур реєстрації драйверів та шрифтів.

Процедура **OvrClearBuf** не має параметрів і виконує очистку оверлейного буфера, тобто вивантажує всі оверлейні модулі, які знаходяться в ньому. Ця процедура виконується в тих випадках, коли необхідно на визначеному етапі програми звільнити динамічну пам'ять для розміщення в ній певних змінних. Для визначення адресів початку і кінця буфера можна використовувати спеціальні змінні **OvrHeapOrg** і **OvrHeapEnd**, що описані в модулі System.

Оверлейний буфер Turbo Pascal найкраще зобразити у вигляді циклічно замкнутого буфера, в якому є показник початку і показник кінця. Оверлеї завжди завантажуються на початок буфера. При цьому найбільш „старі" оверлеї зміщуються в його кінець. Коли буфер заповнюється (тобто між його початком і кінцем не достатньо вільного простору), то вивантажується оверлей у кінці буфера, якщо він в даний момент не використовується, та виділяється місце для нових оверлеїв.

Цей режим використовується адміністратором овелеїв по замовчуванню. Але Turbo Pascal має можливість оптимізувати алгоритм керування оверлеями.

Нехай оверлей А містить деякі часто використовуванні підпрограми. Хоч деякі з цих підпрограм використовуються постійно, існує ймовірність, що оверлей А буде вивантажений з буфера і завантажений в нього знову. Проблема полягає в тому, що підсистема керування оверлеєм нічого незнає про частоту викликів підпрограм в оверлеї А. Вона знає лише, що при звертанні до підпрограми оверлея А його немає у пам'яті, тоді потрібно завантажити цей оверлей. Одне з можливих вирішень полягає в тому, щоб перехоплювати кожне звертання до підпрограм оверлея А і тоді при кожному виклику переміщати оверлей А на початок оверлейного буфера, щоб був виражений його новий стан - як останього використаного оверлея. Таке перехоплення викликів, нажаль, буде занадто непродуктивним на рахунок швидкості виконання і в деяких випадках може навіть сповільнити роботу прикладної програми.

В Turbo Pascal знайдено компромісне вирішення цієї проблеми, яке не приводить до непродуктивних витрат ресурсів і забезпечує високу степінь успіху індетифікації останніх використаних оверлеїв, які не потрібно вивантажувати.

Коли оверлей наближається до кінця оверлейного буфера, починається його випробування. Якщо в ході випробування виконується виклик підпрограми даного оверлея, то він не буде вивантажений при досягненні кінця оверлейного буфера. Замість цього він просто переміщається на початок буфера, і починається новий цикл його переміщення по циклічному оверлейному буфері. Якщо ж в процесі випробування звертання до оверлея не буде, то оверлей при досягненні кінця буфера вивантажеться.

Використання механізму випробувань (проба/відмова) призводить до того, що часто використовувані оверлеї будуть зберігатися у оверлейному буфері за рахунок того,

що буде перехоплюватися майже кожен виклик, коли оверлей наближатиметься до кінця оверлейного буфера.

Для керування механізмом випробувань в модулі Overlay використовується функція **OvrGetRetry** і процедура **OvrSetRetry**. Функція **OvrGetRetry** не має параметрів і виводить поточний встановлений розмір ділянки випробувань. Вона видає результат типу LongInt. На початку роботи програми розмір ділянки випробувань завжди рівний нулю.

Процедура **OvrSetRetry(*Size*)** встановлює розмір ***Size*** (типу LongInt) ділянки випробувань в оверлейному буфері. Дана процедура повинна викликатися після процедур **OvrInit** і **OvrInitEMS**.

По замовчуванню механізм випробувань не використовується. Якщо по певних причинах (див вище) він необхідний, то рекомендується призначити розмір ділянки випробувань, приблизно рівним одній третій буфера. Для чого слід задати процедуру **OvrSetRetry(OvrGetBuf DIV 3)**.

Іншим способом визначення розміру ділянки випробувань є використання спеціальних змінних **OvrTrapCount** та **OvrLoadCount** модуля Overlay, в яких зберігається інформація про те, як часто викликається оверлейний файл. В змінній **OvrTrapCount** зберігається кількість звертань до процедур оверлейного модуля, які виникли в момент, коли модуль був відсутнім в буфері або знаходився в ділянці випробувань. Застосовуючи **OvrLoadCount** можна визначити кількість завантажень в пам'ять даного модуля. Початкові значення цих змінних рівні нулю. Для встановлення оптимального розміру ділянки випробувань необхідно, щоб відношення **OvrLoadCount** до OvrTrapCount було мінімальним.

Для перевизначення функції читання оверлеїв призначені тип і змінні, описанні в модулі Overlay наступним чином:

```
type
        OvrReadFunc = FUNCTION (OvrSeg : word) : integer;
var
        OvrReadBuf: OvrReadFunc;
        OvrFIleMode: byte;
```

Змінна **OvrReadBuf** дозволяє перехоплювати операції завантаження оверлеїв. Коли адміністратору необхідно зчитати овелей, він викликає функцію, адреса якої записаний в **OvrReadBuf**. Якщо функція виводить нульове значення, то адміністратор оверлеїв „вважає", що операція була успішною. Якщо функція встановлює не нульовий результат, то компілятор генерує помилку виконання **Ovrlay file read error** (помилка читання оверлейного файла). Параметр **OvrSeg** визначає, який саме оверлей потрібно завантажити, але не має необхідності встановлювати параметри самостійно, так як він неявно передається у функкцію типу **OvrReadFunc** процедурою **OvrInit**.

У використанні змінної **OvrFileMode** переважно не виникає необхідності. Вона визначає режим доступу до оверлейного файлу, який по змовчуванню встановлений як Read only (тільки для читання). Коли режим доступу необхідно змінити, то робити це потрібно до інсталяції оверлеїв.

При виклику оверлеїв із зовнішніх програм на мові Ассемблера для забезпечення коректної роботи підсистеми керування оверлеями потрібно дотримуватись відповідних правил програмування.

Якщо в програмі на мові Ассемблера реалізується звертання до якої-небудь оверлейної процедури чи функції, то в програмі на Ассемблері повинен використовуватись віддалений тип виклику і з допомогою реєстру **BP** повинні бути встановлені межі стека.

Оверлейні програми на мові Ассемблера не повинні створювати змінних в сегменті коду, оскільки при звільненні оверлея будь-які зміни, внесені в оверлейний сегмент коду, втрачаються. Так само не слід думати, що вказівники на розміщені в оверлейному сегменті коду об'єкти залишуться при виклику інших оверлеїв, оскільки підсистема керування оверлеями може спокійно переміщати і звільняти оверлейні сегменти коду.

Потрібно відмітити, що якщо адмністратор оверлеїв ініціалізується процедурою **OvrInit(ParamStr (0))**, то це означає, що оверлейний файл розміщається в EXE-файлі. В такому випадку, при роботі в інтегрованому середовищі Turbo Pascal основну програму можна компілювати тільки з допомогою комбінації клавіш Alt+F9, але лише на диск (а не в пам'ять!). Ці команди створять виконуючий код, але не запустять програму. Необхідно пам'ятати, що будь-яка спроба виконати основну програму до того, як реалізована операція копіювання оверлейного файлу в EXE-файл в командій стрічці DOS приведе до виникнення помилки ініціалізації оверлею. Тому послідовність дійсно повинна бути наступною:

- відкомпілювати основну програму з допомогою комбінації клавіш Alt+F9 (якщо решта модулів вже скомпільовані);
- вийти з інтегрованого середовища Turbo Pascal;
- виконати операцію копіювання оверлейного файлу в EXE-файл, використавши команду MS-DOS **copy /b ім'я1.exe+ім'я2.ovr**, де **ім'я1** - ім'я виконуючого EXE-файлу програми, а **ім'я2** - ім'я відповідного оверлейного файлу;
- завантажити EXE-файл, щоб провірити правильність його роботи.

14.8. Модуль Printer

Модуль **Printer** призначений для встановлення зв'язку програми з друкуючим пристроєм (найчастіше принтером), що підключений до паралельного порту LPT1. Даний модуль має всього одну змінну **Lst** файлового типу **Text**, яка встановлює зв'язок з системним портом LPT1. Її можна використовувати в якості файлової змінної в процедурах **Write** в **WriteLn**, які будуть виводити дані не у файл, а безпосередньо на друк.

Для цього спочатку потрібно асоціювати змінну **Lst** із службовим словом LPT1, що сприймається операційною системою як перший паралельний порт. Щоб здійснити цю операцію потрібно задати процедуру **Assign(Lst,'LPT1')**. Після цього асоційований умовний файл потрібно відкрити для запису процедурою **Rewrite(Lst)**.

Ці дві процедури виконують автоматичний зв'язок програми з принтером, тому не потрібно після виведення інформації закривати файл Lst.

Використовування модуля Printer може виявитися корисним, якщо програма будується з різних модулів, багато з яких виводять дані на друк. Змінна Lst є загальною для всіх модулів, що є досить зручною властивістю.

Якщо прінтер підключений до іншого порту, то можна написати власний варіант модуля Printer, замінивши в процедурі Assign ім'я 'LPT1' нового порту, наприклад на COM1.

Додаток. Повідомлення про помилки

Повідомлення про помилки виникають в процесі компіляції, або виконання. Якщо помилка виникає, при компіляції всередині Turbo Pascal 7.0, то активізується вікно редагування і курсор поміщається на місце помилки у вихідній програмі, а також виводиться код і назва помилки. Нижче наведено коди і відповідні їм повідомлення про помилки, а також можливі шляхи усунення даних помилок.

1. **Out of memory (Недостатньо пам'яті).** Потрібно вивантажити резидентні програми, здійснити компіляцію на диск (замість пам'яті), або розбити програму на окремі модулі.
2. **Identifier expected (Потрібно індентифікатор).** В цьому місці повинен находитися індентифікатор.
3. **Unknown indentifier (Невідомий ідентифікатор).** Такий ідентифікатор не описаний.
4. **Duplicate identifier (Повторення ідентифікатора)**.
5. **Syntax error (Синтаксична помилка).** Перевірити правильність написання команди, функції чи ін..
6. **Error in real constant (Помилка в константі дійсного типу).**
7. **Error in integer constant (Помилка в константі цілого типу)**.
8. **String constant exceeds line (Стрічкова константа перевищує розміри стрічки)**.
9. **Too many nested files (Забагато вкладених файлів).** Компілятор дозволяє не більше 15 вкладених вихідних файлів і не більше 4 файлів, що включаються.
10. **Unexpected end of file (Неочікуваний кінець файла).** Неоднакова кількість операторів BEGIN і END, або у не закриті лапки коментаря.
11. **Line to long (Стрічка занадто довга).** Максимальна довжина стрічки 127 символів.
12. **Type indentifier expected (Очікування типу індефікатор).** Невказаний тип індифікатора.
13. **Too many opens files (Дуже багато відкритих файлів).** Встановити в файлі CONFIG. SYS значення параметра FILES=30, або більше.
14. **Invalid file name (Недоступне ім'я файла).** Ім'я файлу не відповідає стандарту DOS, або неправильний шлях.
15. **File not found (Файл не знайдений).** Файл відсутній, або неправильний шлях.
16. **Disk full (Диск заповнений).** Знищити з диску непотрібні файли чи зберегти текст програми на іншому диску.
17. **Invalid compiler directive (Недоступна директива компілятора).** Невірна буква в директиві компілятора, або один із параметрів дериктиви компілятора невірний.
18. **Too many files (Занадто багато файлів).** В компіляції програми для програмного модуля беруть участь дуже багато фалів, порібно об'єднати кілька файлів.
19. **Undefined type in pointer definition (Невизначений тип в описі вказівника)**.
20. **Variable indentifier expected (Потрібно індентифікатор змінної)**.
21. **Error in type (Помилка в визначенні типу).** Визначення типу не може починатися з цього символу.
22. **Structure too large (Занадто довга структура).** Максимальний розмір структурного типу - 65520 байт.
23. **Set base type of range (Кількість елементів в множині перевищує допустиме значення).** Базовий тип множини це інтервальний чи перечислювальний тип даних не більше чим 256 значень.
24. **File components may not be files or objects (Компоненти файла не можуть бути файлами чи об'єктами)**.
25. **Invalid string length (Неправильна довжина стрічки).** Довжина стрічки повинна бути в межах від 1 до 255.
26. **Type mismatch (Невідповідність типів).** Неспівпадання типів змінної і виразу в операторі присвоєння, неспівпадання типів фактичного і формального параметрів у зверненні до процедури і функції, тип виразу несумісний з типом індексу при індексації масиву, або несумісні типи оперантів в виразі.
27. **Invalid subrange base type (Неправильний базовий ип для інтервала)**.
28. **Lower bound greater than bound (Нижня границя перевищує верхню).** При опису інтервального типу даних нижню границю встановленою більшою від верхньої.
29. **Ordinal type expected (Потрібно перерахований тип).** Інші типи не використовувати
30. **Conctant expected (Потрібна константа)**.
31. **Integer or real constant exhected (Потрібна константа цілого чи дійсного типу)**.
32. **Integer constant expected (Потрібна контанта цілого типу)**.
33. **Pointer type indentifier expected (Потрібно ідентифікатор типу вказівник)**.
34. **Invalid function result type (Недопустимий тип результату функції)**.
35. **Label identifier expected (Потрібно ідентифікатор мітки). З**силка на мітку, яка не описана в розділі LABEL.
36. **BEGIN expected (Потрібно оператор BEGIN)**.
37. **END expected (Потрібно оператор END)**.
38. **Integer expression expected (Потрібно вираз цілого типу)**.
39. **Ordinal expression expected (Потрібно вираз перечисленого типу)**.
40. **Boolean expression expected (Потрібно вираз логічного типу)**.
41. **Operand types do not match operator (Несумісництво типів оперантів).** Наприклад число поділити на стрічку.
42. **Error in expression (Помилка в виразі).**
43. **Illegal assignment (Заборонене присвоєння).** Файлам і нетипізованим змінним неможна присвоювати значення, або ідентифікатору функції вийшов за межі середини розділу операторів даної функції.

44. **Field identifier expected (Потрібно ідентифікатор поля).** Даний ідентифікатор не відповідає полю змінної типу RECORD чи OBJECT.
45. **Object file too large (Об'єктний файл занадто великий).** Розмір OBJ-файлу повинен бути меншим 64 Кб.
46. **Undefined external (Не визначена зовнішня процедура).**
47. **Invalid object file record (Недопустимий запис в об'єктному файлі).** Запис в об'єктному файлі недопустимий для нього.
48. **Code segment too large (Сегмент коду занадто великий).** Максимальний розмір повинен бути меншим 65520 байт. Потрібно поділити програму чи програмний модуль на кілька частин.
49. **Data segment too large (Сегмент даний занадто великий).** Максимальний розмір сегменту даних програми - 65520 байт. Більші структури даних описують з допомогою вказівників і виділяють для них пам'ять динамічно, з допомогою процедури NEW.
50. **DO expected (Потрібно оператор DO).**
51. **Invalid PUBLIC definition (Недопустимий опис - PUBLIC).** Цей ідентифікатор отримав тип **PUBLIC** з відповідної дериктиви Assembler, а не відповідає опису EXTERNAL в програмі чи програмному модулі, або дві чи більше дериктиви **PUBLIC** на Assembler визначають той самий ідентифікатор.
52. **Invalid EXTRN definition (Неправильне визначення EXTRN).** Із Assembler здійснене посилання з допомогою директиви **EXTRN** на ідентифікатор, який не був описаний в тексті Pascal-програми.
53. **Too many EXTRN definition (Занадто багато визначень тиру EXTRN).** Не можна використовувати файли OBJ, які містять більш ніж 256 визначень **EXTRN.**
54. **OF expected (Потрібно оператор OF)**.
55. **INTERFACE expected (Потрібно оператор INTERFACE)**.
56. **Invalid relocatable reference (Недозволене переміщення посилання).** В obj-файлі не можна використовувати переміщення посилань.
57. **THEN expected (Потрібно операторTHEN).**
58. **TO or DOWNTO expected (Потрібно зарезервоване слово TO або DOWNTO)**
59. **Undefined forward (Невизначено випереджаючий опис).** Процедура чи функція, описана в інтерфейсній частині програмного модуля, а їх визначення відсутнє в частині реалізації, або вони описані за допомогою випереджуючого опису, а їх вміст не знайдено.
61. **Invalid typecast (Недопустиме перетворення типів).** Розмірність змінної і тип результату відрізняється один від одного при зведенні типу змінних.
62. **Division by zero (Ділення на нуль).** Дана операція намагається виконати ділення на нуль.
63. **Invalid file type (Недопустимий тип файлів).** Файловий тип не обслуговується процедурою обробки файлів. Наприклад, процедура **Seek** використовується для текстового файла.
64. **Cannot Read or Write variables of this type (Нема можливості зчитувати чи записувати змінні даного типу).** Процедури **Read і ReadLn** можуть зчитувати змінні тільки символьного,цілого, дійсного і стрічкового типів, а **Write і WriteLn** виводити змінні символьного, цілого, дійсного,булевого і стрічкового типу.
65. **Pointer variable expected (Потрібна змінна типу вказівник).**
66. **String variable expected** (**Потрібна стрічкова змінна**)
67. **String expression expected (Потрібно вираз типу стрічка**).
68. **Circular unit reference (Циклічна залежність модулів).**
69. **Unit name mismatch** (**Невідповідність імен прграмних модулів**). Ім'я програмного модуля, знайдене в файлі .TPU, не відповідає імені, вказаному в операторі USES.
70. **Unit version mismatch** (**Невідповідність версій програмних модулів**). Один або декілька програмних модулів, використовуваних даною програмою, були поміняні після їх компілювання.
71. **Internal stack overflow (Переповнення внутрішнього стеку).** Потрібно перенести функції і процедури в окремий модуль.
72. **Unit file format error (Помилка формату у файлі програного модуля)**.
73. **Implementation expected (Потрібно оператор IMPLEMENTATION)**. В модулі відсутній розділ реалізації.
74. **Constant and case types do not match (Неспівпадання типів константи і оператора CASE).**
75. **Record variable expected (Потрібна змінна типу запис)**. Вказана змінна повинна мати тип - заис.
76. **Constant out of range (Константа за межами діапазону)**. Виникає при спробі вказати масив з константами, що виходять за межі масиву, або при спробі присвоїти змінній значення константи, що виходить за діапазон змінної, або при спробі передати константу поза діапазоном в якості праметра поцедурі чи функції.
77. **File variable expected (Потрібна файлова змінна)**. Вказана змінна повинна мати тип - файл.
78. **Pointer expression expected (Потрібно вираз типу вказівник)**. Вказаний вираз повинен мати тип - вказівник.
79. **Integer or real expression expected (Потрібно вираз цілого або дійсного типу).** Вказаний вираз повинен мати цілий або дійсний типи.
80. **Label not within current block (Мітка за межами поточного програмного блоку).** Оператор **GOTO** не може здійснити перехід на мітку, що знаходиться за межами поточного програмного блоку.
81. **Label already definet (Мітка вже описана**).
82. **Undefined label in processing statement part (Невизначена мітка в попередній частині оператора)**.

83. **Invalid @ argument (Неправильний аргумент оператора @).** Правильними аргументами є імена змінних, процедур або функцій.
84. **Unit expected (Потрібно оператор UNIT).**
85. **„;“ expected (Потрібно символ „;“).**
86. **„:“ expected (Потрібно символ „:“).**
87. **„,“ expected (Потрібно символ „,“).**
88. **„(“ expected (Потрібно символ „(“).**
89. **„)“ expected (Потрібно символ „)“).**
90. **„=“ expected (Потрібно символ „=“).**
91. **:=“ expected (Потрібно символ „:=“).**
92. **„[“ or „(.“ expected (Потрібно символ „[“ або „(.“).**
93. **„]“ or „).“expected (Потрібно символ „]“ або „).“).**
94. **„.“ expected (Потрібно символ „.“).**
95. **„..“ expected (Потрібно символ „..“).**
96. **Too many variables (Надто багато змінних).** Об’єм всіх глобальних або локальних змінних не повинен перевищувати 64 K.
97. **Invalid FOR control variable (Неправильна змінна лічильника циклу FOR).** Змінна лічильника FOR повинна бути перераховного типу.
98. **Integer variable expected (Потрібна змінна цілого типу).** Вказана змінна повинна мати цілий тип.
99. **Files types are not allowed here (Файлий і процедурний типи тут не дозволені).**
100. **String length mismatch (Неспівпадання довжини стрічки).** Довжина стрічкової константи не відповідає допустимій кількості елементів для стрічок.
101. **Invalid ordering of fields (Неправильний порядок полів).** Поля в константі **RECORD** повинні записуватись в порядку їхнього опису.
102. **String constant expected (Потрібна константа стрічкового типу).**
103. **Integer or real variable expected (Потрібна змінна цілого або дійсного типу).**
104. **Ordinal variable expected (Потрібна змінна перераховного типу).**
105. **INLINE error (Помилка в операторі INLINE).** Оператор **INLINE** не можна використовувати з посиланнями що переміщуються.
106. **Character expression expected (Потрібно вираз символьного типу).**
107. **Too many relocation items (Надто багато виконуваних пунктів).** Розмір виконуючого модуля „.EXE“-файлу повинен бути менший 64K (Потрібно використовувати Overlay).
108. **Overflow in arithmetic operation (Переповнення при виконанні математичних операцій).** Значення результату виконаної математичної операції перевищила тип LongInt.
109. **No enclosing FOR, WHILE or REPEAT statement (Не знайдені оператори циклу).** Процедури **Break** і **Continue** використовуються лише в циклах.
112. **CASE constant out of range (Константа в операторі CASE поза допустимим діапазоном). З**начення цілочисельних констант оператора **CASE** повино знаходитись в межах від - 32768 до 32767.
113. **Error in statement (Помилка в операторі).** Символ над курсором не можу бути першим в операторі.
114. **Cannot call an interrupt procedure (Неможливо викликати процедуру обробки переривань).**
116. **Must be in 8087 mode to compile this (Для компіляції необхідний режим 8087).**
117. **Target address not found (Вказаний адрес не знайдений). К**оманда Search/Find Error компілятора не дозволяє знайти оператор, що відповідає вказаній адресі.
118. **Include files are not allowed here (В цьому місці програми підключення файлу неможливе).**
119. **No inherited methods are accessible here (Недопустиме використання спадкових методів).** Зарезервоване слово INHERITED використовують тільки всередині об’єктного типу.
120. **NIL expected (Потрібно оператор NIL).**
121. **Invalid qualifier (Неправильний класифікатор).** Виникає при спробі індексувати змінну, яка не є масивом, або при спробі вказати поля в змінній, яка не являється записом.
122. **Invalid variable reference (Недопустиме посилання на змінну). С**интаксично посилання правильне, але не вказує адресу пам’яті.
123. **Too many symbols (Надто багато символів).** Програма чи програмний модуль не повинна містити більше 64 Кб символів.
124. **Statement part too large (Надто великий розділ операторів). Р**озмір розділу операторів не повинен перевищувати 24 Кб. Частину розділу операторів потрібно перемістити в одну чи декілька процедур.
126. **Files must be var parameters (Файли повинні мати змінні в якості параметрів).**
127. **Too many conditional symbols (Надто багато символів в виразі умови).**
128. **Misplaced conditional directive (Пропущена умовна деректива)**Компілятор знайшов дерективу **{$ELSE}** чи **{ENDIF}** без відповідних деректив {**IFDEF}, {IFNDEF}** чи {**IFOPT}.**

129. **ENDIF directive missing (Пропущена деректива ENDIF).** Вихідний файл закінчився всередині конструкції умовної компіляції. У вихідному файлі повинна бути рівна кількість деректив **{$Ifxxx}** і **{$ENDIF}**.
130. **Error in initial conditional defines(Помилка у визначенні початкових умовних виразів).** Вихідні умовні ідентифікатори, вказані у опції **Options/Compiler/Conditional Defines** є недозволеними.
131. **Header does not match previous definition (Заголовок не відповідає попередньому визначенню).** Заголовок процедури чи функції, вказаний в інтерфейсній частині, не відповідає заголовку виконуючої частини процедури чи функції, або заголовок процедури чи функції, вказаний з допомогою попереднього описання **FORWARD,** не відповідає заголовку знайденої однойменної процедури чи функції.
132. **Critical disk error (Критична помилка диску).** Фізична помилка на дискеті чи жорсткому диску.
133. **Cannot evaluate this expression (Не можливо обчислити цей вираз).** В виразі-константі чи іншому використовується недозволені засоби.
134. **Expression incorrectly terminated (Вираз неправильно завершено).**
135. **Invalid format specifier (Неправильний визначник формату).**
136. **Invalid indirect reference (Неправильний непрямий вказівник).** Оператор намагається здійснити недопустиму непряме посилання.
137. **Structured variable are not alloved here (В даному місці не допускається використання структурної змінної).** Наприклад, причиною помилки може бути спроба перемножити два записи.
138. **Cannot evaluate without System unit (Неможлива компіляція без системного модуля).** Немає файлу TURBO.TPL, який містить системний модуль, або він пошкоджений.
139. **Cannot access this symbol (Немає доступу до цього символу).** В режимі компіляції недоступні певні символи.
140. **Invalid floating-poing operation (Неправильна операція з дійсними числами). О**тримано переповнення або ділення на нуль.
141. **Cannot compile overlays to memory (Не можна компілювати програми з оверлеями в пам'яті).** Встановити компілювання на диск.
142. **Pointer or procedural variable expected (Потрібна змінна типу процедура або функція).**
143. **Invalid procedure or tunction reference (Неправильний вказівник на процедуру або функцію).** Спроба викликати процедуру у виразі.
144. **Cannot overlay this unit (Модуль не може використовувати оверлеї).**
145. **Too many nested scopes (Надто багато вкладень).** Програма може використовувати до 512 вкладень, а кожен модуль до 128.
146. **File access denied (Заборонено доступ до файлу).** Файл не може бути відкритий або створений. Найчастіше компілятор намагається зробити запис у файл з атрибутом Read Only.
147. **Obyect type expected (Потрібно об´єктивний тип).**
148. **Local object types not allowed (Локальні об'єктні типи не дозволені).** Об'єктні типи можуть бути визначені тільки в глобальному блоці програми і модуля.
149. **VIRTUAL expected (Очікується ключеве слово VIRTUAL).**
150. **Method identifier expected (Очікується ідентифікатор методу).**
151. **Virtual constructor are not allowed (Конструктор не можна оголошувати віртуальним).** Він повинен бути статичним.
152. **Constructor idintifier expected (Очікується ідентифікатор конструктора).**
153. **Destructor idintifier expected (Очікується ідентифікатор деструктора).**
154. **Fail only allowed within constructors (Виклик FAIL допускається тільки всередині конструктора)**
155. **Invalid combination of opcode and operands (Не доступна комбінація коду операції і операндів).** Код асемблерної команди не сприймає дане поєднання операндів.
156. **Memory reference expected (Очікування зсилки на область пам'яті).** Операнд асемблерної інструкції не є потрібним вказівником на область пам'яті
157. **Cannot add or substract relocatable symbols (Неможливе додавання або віднімання символів, що переміщаються).**
158. **Invalid register combination (Недопустима комбінація регістрів).**
159. **286/287 Instructions not alloved (Інструкції процесорів 286/287 не дозволені).** Потрібно використати директиву компілятора {$G+}.
160. **Invalid symbol reference (Неприваильне посилання на симол).**
161. **Code generation error (Помилка генерації коду).**
162. **ASM expected (Потрібно ключове слово ASM).**
163. **Duplicate dynamic method index (Дублювання індексу динамічного методу).**
164. **Duplicate resource identifier (Дублювання ідентифікатора ресурсу).**
165. **Duplicate or invalid export clause (Дублювання або наявність недозволеного експортного індексу).**
166. **Procedure or function intentifier expected (Потрібно ідентифікатор процедури або функції).**
167. **Cannot export this symbol (Неможна експортувати цей символ).**
168. **Duplicate export name (Подвійне експортне ім'я).**
169. **Executable file header too large (Заголовок виконуваного файлу надто великий.)**

Рекомендована література

1. Аладьев В.З. Тупло В.Г. Turbo Pascal для всех. -К.: Техника, 1993. -176с.
2. Бартків А.Б. та ін.Турбо Паскаль: Алгоритми і програми. -К.: Вища школа, 1992. -248с.
3. Джонс Ж., Харроу К. Решение задач в Турбо Паскаль. -М.:Финансы и статистика, 1991. -720с.
4. Довгаль С.И., Сбитнев А.И. Интерфейс современной программной системы Турбо Паскаль 7.0. -К.: Информсистема-сервис, 1994. -416с.
5. Йенсен К., Вирт Н. Паскаль. Руководство пользователя и описание языка. -М.: Финансы и статистика, 1989. -256с.
6. Паскаль для персональных компьютеров: Справочное пособие. -М.: Высшая школа, 1991. -365с.
7. Перминов О.Н. Программирование на языке Паскаль. Справочник. -М.: Радио и связь., 1989. -129с.
8. Фаронов В.В. Турбо Паскаль 7.0. Практика программирования. Учебное пособие. -М.: Нолидж, 1997. -432с.
9. Ян Белецкий. Турбо Паскаль с графикой для персональных компьютеров. -М.: Машиностроение, 1991. -320с.

Папір офсетний. Друк офсетний.
Умови.-друк. арк.
Облік.-видавн. арк. Зам. № ???

СМП "Астон" м.Тернопіль, вул.Гайова, 8
тел. (0352) 22-71-36, 22-25-60