МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЧЕРКАСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ім. Богдана Хмельницького

Авраменко В.С., Салапатов В.І.

ВСТУП ДО ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Том 2. Основи програмної інженерії

Навчальний посібник

Друге видання

Черкаси 2014

УДК 004(075.032)
ББК 018*32.973я723

Рецензенти:
Чл.-кор. АН України, д.т.н., проф. *Васильєв В.В.*
д.т.н., проф. *Литвинов В.В.*

Рекомендовано до друку Вченою радою
Черкаського національного університету імені Богдана Хмельницького
(протокол №2 від 15.11.2011 р.)

Авраменко В.С., Салапатов В.І.

Вступ до програмної інженерії. Том 2. Основи програмної інженерії. Навчальний посібник. Друге видання. – Черкаси: ЧНУ імені Богдана Хмельницького, 2014, – 370 с.
ISBN 978-966-353-261-5

Мета даного посібника – сформувати уявлення про професію програміста та закласти основи для отримання навичок в області обчислювальної техніки, комп'ютерних наук та програмування, розвинути професійний світогляд майбутніх програмістів та адміністраторів комп'ютерних систем та мереж, орієнтувати їх у бурхливому вирі нових ідей та технологій апаратного та програмного забезпечення, операційних систем і мов програмування.

У посібнику розглядаються вміст основних понять програмування, історія його розвитку, початкові відомості з інформатики, основні елементи мов програмування, принципи структурного програмування та основи програмування у середовищі Turbo Pascal.

Посібник розбитий на два томи. В першому томі надається еволюція розвитку комп'ютерної техніки та основи інформатики. У другому томі йдеться про основи програмування взагалі та конкретне застосування мови Pascal при розробці програм. Наводяться численні приклади перевірених програм.

Призначено для студентів ВНЗ, які навчаються за спеціальностями по напрямах підготовки "Комп'ютерні науки", "Програмна інженерія", "Системний аналіз", "Комп'ютерна інженерія", "Інформатика та обчислювальна техніка", для студентів інших споріднених спеціальностей, а також учнів старших класів шкіл, технікумів, коледжів з поглибленим вивченням інформатики.

У другому виданні посібника виправлені деякі помилки, доопрацьовані, доповнені та додані нові розділи.

УДК 004(075.032)
ББК 018*32.973я723

ISBN 978-966-353-261-5

ЗМІСТ

Передмова

Можна з упевненістю сказати, що сьогодні знання основ інформатики та обчислювальної техніки стали необхідними кожній людині як у переважній більшості областей професійної діяльності, так і в побуті, та по-справжньому є загальноосвітніми. Тому мета цього посібника – звернутися до минулого, зануритися в історію розвитку апаратного та програмного забезпечення і зрозуміти, як воно розвивалося і чому ще й досі не існує загальних технологій, які дозволили б усім розробникам створювати надійне програмне забезпечення з мінімальними витратами і за прийнятний проміжок часу, полегшити вивчення та використання основ інформатики та мов програмування.

Цей посібник за своїм змістом істотно відрізняється від підручників і посібників для початківців. Він виник на основі курсу лекцій для студентів першого курсу факультету обчислювальної техніки, інтелектуальних та управляючих систем Черкаського національного університету ім. Б. Хмельницького «Вступ до програмної інженерії» та «Основи програмування та алгоритмічні мови», які обрали програмування та адміністрування комп'ютерних систем та мереж своєю професією.

Студенти першого курсу після закінчення школи в цілому обізнані з побудовою комп'ютера та основами програмування, мають навички роботи з типовими пакетами програм. Вони із захопленням читають спеціальну літературу, сперечаються про переваги та недоліки новітніх мікропроцесорів та програм. Проте, в цілому, їх знання поверхневі та носять уривчастий характер, оскільки вони відірвані із загального контексту історії розвитку обчислювальної техніки та ідей інформатики і програмування.

Метою курсу «Вступ до програмної інженерії» і цього посібника зокрема є розвиток професійного світогляду майбутніх програмістів та адміністраторів комп'ютерних систем і мереж, орієнтація їх у бурхливому вирі нових ідей та технологій апаратного та програмного забезпечення, операційних систем та мов програмування.

Звідси витікає обраний нами історичний підхід до викладення матеріалу. Дивлячись на новинки, що з'являються щодня, легко розгубитися. Неосвідченій людині може здатися, що вся ця нескінченна різноманітність хаотична і абсолютно непередбачувана. Насправді це далеко не так. Найдивовижніше в інформатиці – не швидка мінливість, а, навпаки, дивовижна стійкість фундаментальних концепцій.

Електронним обчислювальним машинам не більше шестидесяти п'яти років. За цей час вони невпізнанно змінилися зовні, у мільйони разів збільшилася швидкість їх роботи та розмір пам'яті, проте основні принципи апаратного та програмного забезпечення залишилися багато у чому практично незмінними.

Темпи розвитку ЕОМ воістину фантастичні. Ще в 1984 р. американські газети писали:

«У 1953 р. ЕОМ з пам'яттю 64 Кбайт коштувала 1 млн дол, зараз вона коштує менше 1 тис. дол. Якби автомобілі розвивалися протягом останніх 20 років тими ж темпами, як комп'ютери, то сьогодні роллс-ройс коштував би 3,0 дол, проходив мільйон миль на галоні бензину, розвивав потужність лайнера «Queen Elisabeth» і 2 автомобілі поміщалися б на кінчику пера».

У 21 столітті темпи ще вище. Наприклад, якщо мікропроцесор Pentium IV «Northwood» (2002 р.) містив 42 млн транзисторів, то Pentium IV «Prescott» (2004 р.) – 125 млн.

Програмне забезпечення є «тінню» свого «старшого брата». Судіть самі: найпопулярніші мови програмування та алгоритми компіляції були розроблені на початку 60-х років минулого століття. Тоді ж були сформульовані принципи роботи операційних систем. Системи управління базами даних (СУБД) з’явилися на початку 70-х років і з тих пір майже не змінилися. Навіть наймодніша нині концепція об’єктно-орієнтованого програмування була запропонована, виявляється, ще в середині 60-х років минулого століття (мова Smalltalk). Наведені приклади свідчать про те, що принципово нові ідеї з’являються в інформатиці, як, втім, і в інших науках, відносно рідко, і в найновітнішому та розрекламованому пакеті програм, якщо добре придивитися, можна побачити добре забуте старе.

Правда, за останній час, завдяки новим технологіям, широке використання баз даних різними категоріями користувачів призвело, з одного боку, до створення інтерфейсів, що вимагають мінімум часу на освоєння коштів управління системою, а з іншого – до побудови потужних, гнучких СУБД, що мають у тому числі розвинені засоби захисту даних від випадкового або навмисного руйнування. З'явилися і засоби автоматизації розробки, що дозволяють створити базу даних будь-якому користувачеві, навіть якщо він не володіє основами теорії БД.

Можливості накопичувати і оперативно обробляти великі обсяги інформації, що характеризують діяльність підприємств за досить тривалі періоди і в різних аспектах, дали новий імпульс до розвитку аналітичних систем. Такого роду системи підтримки прийняття рішень зазвичай використовуються для оцінки та вибору альтернативних рішень, прогнозування, ідентифікації об'єктів і станів тощо.

У першій частині посібника висвітлюється історія обчислювальної техніки та еволюція розвитку програмного забезпечення, як основного базису інформатики – від простих механічних пристроїв до сучасних комп’ютерів, від програмування на комутаційних дошках (пультах) до сучасних технологій програмування.

У другій частині розглядаються принципи побудови та напрями еволюції сучасних комп’ютерних мереж, коротко описана їх історія, основні поняття теорії та тенденції розвитку комп’ютерних мереж.

У третій частині описуються загальні відомості про комп'ютер, інформацію, алгоритми та мови програмування.

Четверта частина присвячена основам програмування в середовищі Паскаль, як початкового (базового) засобу навчання програмуванню для студентів. Ми вважаємо, що вивчення будь-якої мови програмування проходить найуспішніше, коли кожне питання розглядається з різних боків. Тому цей посібник містить опис не тільки різних базових елементів мови Паскаль, але і велику кількість прикладів (близько 170) для закріплення матеріалу, самоконтролю, експериментування та аналізу своїх помилок.

На початковому етапі задачі навчання пов'язані не стільки з мовою, скільки з виробленням алгоритмічного мислення. Ці завдання багатогранні та достатньо складні. Потрібне розуміння нових понять: змінних та типів даних. Потрібне вміння описувати процес обчислення обмеженим набором управляючих структур – вибору, циклів, рекурсії та лінійних блоків. Потрібне знання основ подання даних та класичних алгоритмів. І, нарешті, потрібно не тільки знати, але й вміти писати програми!

Мета даної частини посібника – не зробити з вас високопрофесійних програмістів, бо для цього необхідно цілеспрямовано вчитися і створювати багато різних програм, а засвоїти основні поняття програмування, набути початковий досвід самостійної розробки програм та ознайомитись з деякими класичними завданнями та алгоритмами.

Велика увага приділена стилю програмування. Питання дійсно дуже важливе особливо для програміста-початківця. Немає нічого дивного у тому, що різні програмісти дотримуються різних точок зору при написанні програм. Деякі прагнуть добитися, щоб програма містила мінімальну кількість операторів або виконувалася за мінімальний час. Точка зору на програмування, що наводиться у даному посібнику, ґрунтується на іншому критерії якості програм. Хороша програма відповідно до цього критерію – це програма, яка написана так, щоб вона була зрозумілою іншим користувачам і водночас, по можливості, містила мінімальну кількість операторів та виконувалася за мінімальний час. Більш докладно це означає, що програміст повинен писати програму так, щоб її було легко читати, і, щоб її було легко використовувати.

Цілі, які відмічені вище, можуть показатися зрозумілими, проте у реальному житті вони не завжди легко досягаються. Основне завдання, яке полягає у написанні хорошої програми, вимагає для свого розв'язку серйозного та зваженого ставлення. При розробці програми часто виникає спокуса принести ясність у жертву незначному підвищенню ефективності. На щастя, мова Паскаль містить гарні засоби проектування зрозумілих програм. Її систематизована структура накладає серйозні обмеження на дисципліну програмування, і у той же час дозволяє розширювати мову за рахунок введення нових можливостей, які задовольняють спеціальним застосуванням. Проте, слід пригадати «аксіому» програмування: «**Немає, не було, та, скоріше за все, не буде мови програмування, яка б гарантувала нас від написання поганих програм**».

Ми намагались написати цей посібник без складних слів та виразів, звичною мовою, яка зрозуміла кожному. Робота з посібником не вимагає попередніх знань з інформатики та програмування.

Автори усвідомлюють також те, що цей посібник не повністю відображає матеріал, який необхідний для вивчення основ інформатики. За його межами опинився опис цікавих інформаційних технологій, але цим матеріалом довелося пожертвувати через обмежений об’єм посібника.

У другому виданні посібника виправлені деякі помилки, доопрацьовані, доповнені та додані нові розділи.

ЧАСТИНА IV. ОСНОВИ ПРОГРАМУВАННЯ В СЕРЕДОВИЩІ PASCAL

Вступ

Багато певних стандартних одноманітних дій людської діяльності можно подати у вигляді певної послідовності кроків. Як вже було раніше наголошено, таких видів кроків існує три: лінійні блоки, в яких послідовно виконуються певні дії, блоки розгалуження, в яких перевіряються певні умови і обираються, відповідно, певні гілки виконання тих чи інших кроків, та блоки зациклювання, які за певних умов забезпечують повтор виконання певних послідовностей кроків. Ці основні блоки реалізовані мовами програмування у вигляді так званих операторів, з яких складається програма, що описує послідовність кроків. Потім така програма за допомогою програми-транслятора перетворюється у низку машинних команд, яка реалізує виконання заданих кроків. Далі ця програма відлагоджується та впроваджується. В сучасному світі вже важко обійтися без комп'ютерів і, тим більше, без знання спеціалізованих програм для різних сфер людської діяльності, де застосовується комп'ютер. Адже вся комп'ютерна техніка покликана допомагати людині, але для того щоб «роз'яснити» комп'ютеру, що Ви від нього хочете – треба вміти говорити з ним однією мовою. Для розуміння того, як працює комп'ютер, як керувати та використовувати комп'ютерну потужність у роз'язанні будь-яких завдань, покликаний допомогти цей посібник. У цьому розділі посібника пропонується вивчення можливостей створення програм на мовах програмування на прикладі застосування мови Pascal. Засвоївши прийоми та методи програмування на мові Pascal, можна навчитись програмувати на будь-якій інший спеціалізованій мові програмування.

Мова Pascal має багаторічну історію. Перша версія мови, яка була запропонована її автором, – професором кафедри обчислювальної техніки Швейцарського федерального інституту технології, лауреатом премії Тьюринга, – Никлаусом Віртом, з'явилася ще у 1968 році як альтернатива існуючим на той час мовам програмування, які постійно ускладнювались, таким, як АЛГОЛ та ФОРТРАН. Мова названа на честь видатного французького математика і філософа Блеза Паскаля (1623-1662). Н. Вірт є автором ще таких мов програмування, як Ейлер, Модула, Модула-2,3. Крім цього ним запропонована методика покрокової розробки програм від глобального до локального, від загального до часткового, тобто занурення в алгоритм зверху донизу.

Мова Pascal покликана полегшити вивчення та використання мов програмування при застосуванні різних інструментальних засобів. Інтенсивний розвиток мови Pascal привів до появи вже у 1973 році її стандарту у вигляді переглянутого повідомлення, а кількість трансляторів з цієї мови вже у 1979 році перевищило, за оцінками Н. Вірта, за 80. На початку 80-х років мова Pascal ще більш зміцнила свої позиції з появою трансляторів MS–PASCAL та

TURBO–PASCAL для персональних ЕОМ. З тієї пори мова Pascal стає однією з самих популярних мов програмування для персональних ЕОМ. Істотно те, що мова вже давно вийшла за рамки академічного і вузькопрофесійного інтересу і використовується у більшості університетів та інших навчальних закладів як засіб навчання студентів програмуванню.

Знайомство з мовою розпочнемо з версії TURBO–PASCAL 7.0 (7.1). TURBO–PASCAL задовольняє вимогам найрізноманітніших користувачів, які працюють на персональних комп'ютерах IBM PC та сумісних з ними. TURBO–PASCAL є структурованою мовою високого рівня, яку можна використовувати для написання програм будь-якого розміру та будь-якого призначення. Версія TURBO-PASCAL є 7-ою за поколінням передового програмного виробу фірми Борланд (нині Inprise). Версією 1.0 TURBO–PASCAL фірма Борланд започаткувала створення швидкодіючих компіляторів для мікрокомп'ютерів. Версія 7.0 ще більше підвищує значення TURBO – PASCAL, як мови серйозних розробок. Тому не випадково мова Pascal набула широкого розповсюдження у зв'язку з розвитком комп'ютерної мережі у школах та середніх і вищих навчальних закладах.

Чому треба починати з мови Pascal на початкових етапах навчання програмуванню, а не одразу з .NET або інших сучасних технологій? Щоб навчити людину програмуванню треба перебудувати спосіб її мислення. Необхідно відібрати у студентів, які тільки починають вивчення програмування, всілякі красощі і змусити їх думати. Програмування – це філософія, і будь-яка краса візуальних середовищ тільки заважає її розуміти. Якщо Ви дійсно хочете навчитися програмувати, то з власного досвіду зауважимо, що досить вивчити одну базову мову програмування і Ви зможете засвоїти, також, інші мови програмування без особливих проблем. А потім людина, яка вже вміє програмувати, сама обере собі середовище розробки (платформу) до душі (для роботи) та зуміє її засвоїти.

Чому саме мова Pascal обрана як базова мова? Та все дуже просто:

1. Найважливішою особливістю мови Pascal є втілена ідея структурного програмування.

2. Іншою істотною особливістю є концепція структури даних як одного з фундаментальних понять.

3. Простота мови дозволяє швидко засвоїти її та створювати алгоритмічно складні програми.

4. Розвинені засоби подання структур даних забезпечують зручність роботи як з числовою, так і з символьною та бітовою інформацією.

5. Оптимізуючі властивості трансляторів з мови Pascal дозволяють створювати ефективні програми.

6. У мові Pascal реалізуються ідеї структурного програмування, що робить програму наочною і дає гарні можливості для розробки та відлагодження.

Тому не випадково мова Pascal набула широкого поширення у зв'язку з розвитком комп'ютерної мережі в середніх школах та вищих навчальних закладах.

Цей посібник не має на меті зробити з вас висококласних програмістів, цьому необхідно цілеспрямовано вчитися та постійно створювати багато різних програм, а допомогти засвоїти основні поняття програмування, навчитися програмувати, отримати початковий досвід самостійної розробки програм та ознайомитись з деякими класичними задачами і алгоритмами.

У цьому посібнику наводяться початкові відомості з мови програмування Pascal, матеріали, які побудовані так, що не вимагають від читача попередньої підготовки в області програмування. Матеріал посібника підібраний таким чином, що робить процес вивчення мови Pascal простим, зрозумілим та послідовним. Тому, якщо ви ніколи раніше не займалися програмуванням, то цей посібник дасть вам ключ для засвоєння не лише мови Pascal, але вам відкриються двері до вивчення будь-якої іншої мови програмування.

На якомусь етапі занурення у світ програмування на мові Pascal можуть знадобитись додаткові відомості. Опанувавши запропонований посібник, ви вже будете готові до сприйняття лекцій, до виконання лабораторних робіт, до роботи з довідниками з більш детальною та повнішою інформацією, яка знайдеться в книжкових магазинах, на полицях бібліотек або в мережі Інтернет.

Якщо ви хочете вивчити мову Pascal у повному обсязі, то цей посібник для цього не призначений. По-перше, тому, що мова Pascal настільки велика, що в повному обсязі в усьому світі мало кому потрібна. По-друге, тому, що для повного її викладення знадобилось би біля 6 тисяч сторінок тексту, а такий великий об’єм відомостей про мову для програміста-початківця швидше за все зашкодить йому, аніж допоможе. Тому мета даного посібника не в повноті охоплення мови, а у тому, щоб навчити програмувати і використовувати основні засоби мови Pascal. Однак, прочитавши цей посібник, Ви зможете написати декілька простих програм і побачите, як це в принципі робиться. А все інше можна дізнатися в довідкових матеріалах засобів розробки, на лекціях і лабораторних роботах, а також в Інтернеті.

Найчастіше проблеми навчання знаходяться не в людині, а у методах навчання. До програмування це має найбезпосередніше відношення, оскільки навчання програмуванню пов’язано з отриманням великої кількості складних технічних знань. Великі об’єми такого роду інформації не можуть засвоїтися швидко та легко. Щоб знання вклалися у чітку, структуровану систему потрібна постійна ПРАКТИКА. А тепер поговоримо, чому ж окремі методи навчання не завжди дають бажаний результат.

Мову програмування можна сподобити дуже примітивній природній іноземній мові з жорсткими правилами, які не мають виключень. Вивчення іноземної мови зазвичай починають з алфавіту, потім переходять до простих слів, далі розглядають закони побудови фраз, і лише внаслідок тривалої практики стає можливим вільно висловлювати цією мовою свої думки. Приблизно так само поступають багато авторів при вивченні мов програмування та, зокрема, мови Pascal.

У подібному підході є два недоліки. По-перше, він практично зводиться до переказу (дублюванню) документації, тому такі книги нудно читати. По-друге,

якщо читач ще не розуміє концепцій, які представлені в таких "вступах", то йому взагалі ще дуже рано читати подібні книги.

Методика навчання у нас буде іншою, ніж у більшості посібників. Ми не забиватимемо голову масою незрозумілих спочатку розмов, а будемо поступово на прикладах та на комп’ ютері вивчати основні конструкції мови Pascal. Без комп’ютера читати цей посібник просто безглуздо, багато розмов залишаться незрозумілими. Матеріал посібника побудований за принципом "від простого до складного", з поступовим ускладненням при викладанні матеріалу. З перших занять студенти починають розбиратися у складанні програм, які можуть реально виконуватися на доступних ЕОМ, та поступово опановують навички розробки на мові Pascal лінійних алгоритмів, розгалужених та ітеративних алгоритмів, алгоритмів з рядками, масивами та процедурами. Після того, як ви твердо засвоїте основи мови Pascal, до них додасться надбудова у вигляді нових елементів мови. Посібник містить близько 160 учбових прикладів. Усі приклади у посібнику перевірені для мови Pascal версії 7.0 (7.1). Хочеться, також, звернути увагу читача, що найважливішою частиною даного посібника є тексти програм. Інколи вони говорять більше, ніж пояснювальний коментар.

Велика увага приділена стилю програмування. Питання дійсно дуже важливе особливо для програміста-початківця. Немає нічого дивовижного у тому, що різні програмісти дотримуються різних точок зору при написанні програм. Деякі прагнуть досягнути, щоб програма містила мінімальну кількість операторів або виконувалася за мінімальний час. Точка зору на програмування, яка подана у цьому посібнику, грунтується на іншому критерії якості програми. Гарна програма відповідно до цього критерію – це програма, яка написана так, щоб вона була зрозуміла іншим людям. Детальніше це означає, що програміст повинен складати програму так, щоб її було легко читати, та було легко використовувати. Такого стилю програмування, який задовільняє вказаним критеріям, дотримуються практично всі сучасні технології програмування.

Таким чином, завдання полягатиме не лише у тому, щоб вивчити синтаксис деякої мови програмування, але й засвоїти у певній мірі професійний підхід для вирішення задач на цій мові методами спадаючого проектування та структурного програмування.

Об’єктно-орієнтоване програмування, використання мови Assembler, повний опис стандартних процедур та функцій, стандартних модулів у посібнику навмисно не наведені з тим, щоб не перетворити цей посібник на довідник, яких нині дуже багато. Разом з цим усі відібрані питання розглянуті у посібнику досить детально, що дозволяє читачам отримати достатньо широкі, глибоко усвідомлені знання та навички, які дають можливість вирішувати широке коло обчислювальних завдань, а також безперешкодно використовувати будь-які довідники з Turbo Pascal для розширення своїх знань. Наша мета – не зробити з вас високопрофесійних програмістів, цьому необхідно цілеспрямовано вчитися та складати багато різних програм, а засвоїти основні поняття програмування, набути початковий досвід самостійної

розробки програм та ознайомитсь з деякими класичними завданнями та алгоритмами.

І, нарешті, перш ніж приступити до роботи, хотілося б дати невелику пораду. Навчитися програмувати можна тільки програмуючи, розвязуючи конкретні завдання. При цьому досягнуті у програмуванні успіхи значною мірою залежать від досвіду. Тому, щоб отримати максимальну користь від цього посібника, ви повинні активно з ним працювати. Не треба займайтися просто читанням прикладів, реалізуйте та перевіряйте їх на вашому комп'ютері. Ще раз нагадаємо, щоб засвоїти мову програмування, яку-небудь методологію, або середовище розробки потрібна **практика**! Не бійтеся експериментувати – вносьте зміни до запропонованих прикладів.

1. Алгоритм і програма

Перш ніж почати створювати програми, Вам необхідно засвоїти основи знань, які потрібні для початку програмування: що таке алгоритм, як скласти блок-схему та скласти алгоритм вашої програми у текстовому вигляді.

Програмування не можливе без знання мов програмування, але не менше неможливо воно і без знання алгоритмів. Тому у цьому розділі ми ще раз нагадаємо, що являє собою алгоритм, і що являє собою програма, якщо вам, звичайно, лінощі заважають звернутися до попередньої частини посібника.

Написанню програми завжди передує розробка деякого плану (***алгоритму***) розв'язання задачі. Коротко опишемо основні поняття теоретичної інформатики, які пов'язані з алгоритмами.

1.1. Алгоритм

Алгоритм – це однозначно певна послідовність дій, яка записана зрозумілою виконавцю *алгоритмічною мовою,* і визначає процес переходу від вхідних даних до результату.

У цьому визначенні вже вказані основні **властивості алгоритму**. По-перше, алгоритм складається з кінцевого набору *інструкцій* або *кроків*. По-друге, кожен крок трактується виконавцем єдиним чином, що дозволяє гарантовано отримати результат для деякого набору вхідних даних. По-третє, алгоритм завжди зводиться до деякого перетворення вхідних даних у *результат* або результати.

У цьому сенсі формули для розв'язання квадратного рівняння або навіть чітко складену інструкцію із заварки чаю можна вважати алгоритмами, що здійснюються людиною-виконавцем. Для машини, зрозуміло, потрібна чіткіша формалізація завдання, ніж для людини, – звідси необхідність урахування при складанні алгоритму обмеженого набору інструкцій ЕОМ.

1.2. Властивості алгоритму

Коротко опишемо основні властивості алгоритму.

Дискретність – алгоритм складається з окремих інструкцій (*кроків*).

Однозначність – кожен крок розуміється виконавцем єдиним чином.

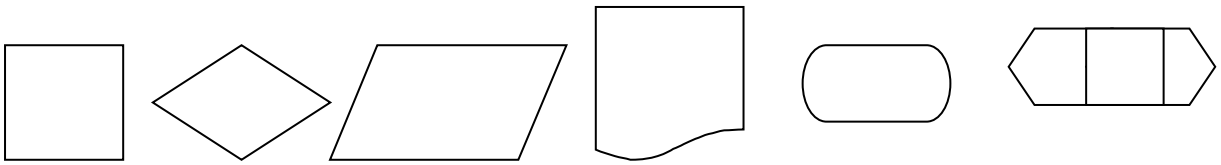
Масовість – алгоритм працює при змінних в деяких припустимих межах вхідних даних.

Результативність – за кінцеву кількість кроків досягається певний результат.

1.3. Форми запису алгоритму

Прийнято виділяти дві основних форми подання алгоритму.

1. <u>Графічна</u> (*блок-схема*) – окремі кроки алгоритму зображаються геометричними фігурами, послідовність виконання кроків – зв'язками між фігурами;



Вказані на малюнку основні фігури блок-схем інтерпретуються так.

Прямокутник – будь-яка послідовність дій; всередині прямокутника записуються формули або мовний опис дій, що виконуються.

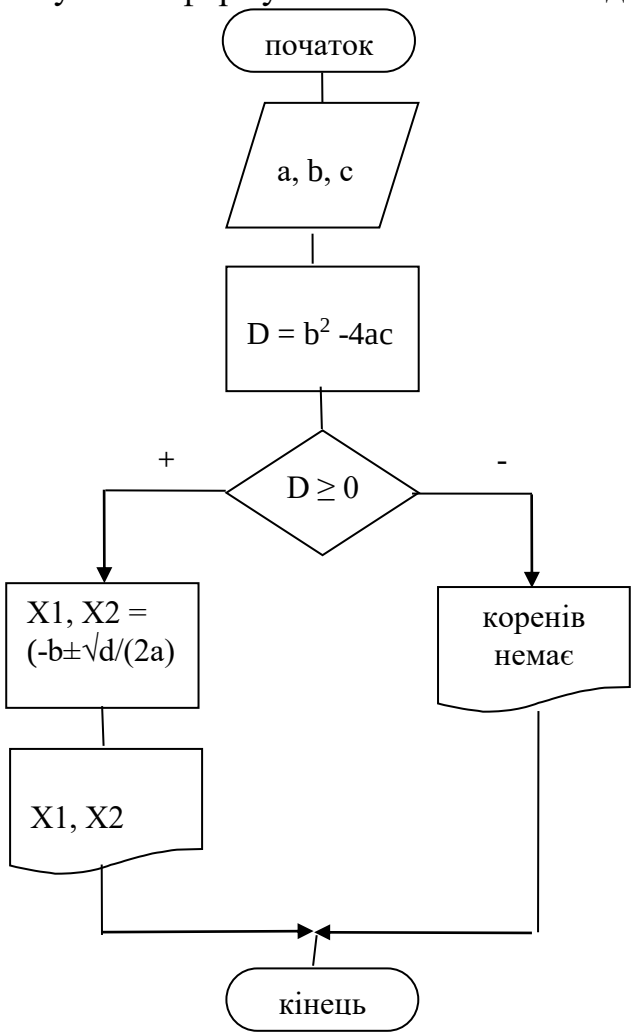


Рис. 1.1. Блок-схема алгоритму вирішення квадратного рівняння

Ромб – блок перевірки умови; оскільки будь-яка умова може бути лише істинною або хибною, то у блоці є один вхід та два виходи, які відповідають діям, що мають виконуватися у випадках, коли умова істинна та коли вона хибна. Виходи підписують символами "+" і "-", або "так" чи "ні", "1" та "0" тощо.

Паралелограм – блок введення вхідних даних. Всередині фігури зазвичай записується, які саме дані мають бути введені.

Аркуш з розривом – блок виведення даних. Всередині фігури вказується, які саме дані або повідомлення програма виводить для надання користувачеві.

Закруглений прямокутник – блоки початку та кінця програми, всередині блоків зазвичай вказується "початок" або "кінець" відповідно.

Остання фігура використовується для зображення циклів, як правило, у неї 2 входи (перший та повторний вхід у цикл) та 1 вихід, який пов'язаний із завершенням циклічного процесу. Ця фігура може бути замінена й іншими конструкціями.

На малюнку наведений спрощений приклад блок-схеми, який ілюструє добре знайомий нам процес вирішення квадратного рівняння.

Мова блок-схем досить громіздка, як правило, вона не застосовується професіоналами, проте, на початковому етапі навчання програмуванню планування нескладних програм у вигляді блок-схем може виявитися дуже корисним.

2. <u>Текстова</u> форма запису алгоритму (***псевдокод***) – кроки алгоритму та послідовність їх виконання задаються набором ключових слів. Ця форма ближче до реальних мов програмування. Існує багато різних варіантів псевдокоду, наприклад, в російськомовній (українській) літературі поширений такий набір ключових слів:

поч - початок програми
кін - кінець програми
якщо-то-інакше - перевірка умови
введення - введення даних
виведення - виведення даних
для-від-до-нц-кц - цикл з лічильником (пц – початок циклу, кц – кінець)
доки-пц-кц - цикл з передумовою
пц-кц-доки - пост цикл з умовою

Всі алгоритмічні конструкції, що відповідають цим ключовим словам, будуть розглянуті нами у даному посібнику. Як правило, програмісти використовують елементи псевдокоду при плануванні частин своєї майбутньої програми.

1.4. Програма та програмне забезпечення

Програма – це *реалізація* алгоритму за допомогою конкретної мови програмування. Сукупність існуючих програм утворює **програмне забезпечення** (ПЗ). ПЗ прийнято ділити на два види:

Системне ПЗ – забезпечує роботу комп'ютера та зовнішніх пристроїв, а також підтримку прикладних програм. Воно розробляється кваліфікованими фахівцями на машинно-орієнтованих мовах, які забезпечують доступ до апаратури комп'ютера на фізичному рівні. Приклади системного ПЗ – операційна система Windows (або будь-яка інша), *драйвери* зовнішніх пристроїв комп'ютера, *утиліти* для технічного обслуговування комп'ютера, *системи програмування*, які призначені для розробки власних додатків.

Прикладне ПЗ призначено для вирішення конкретних задач користувача. Воно розробляється на мовах високого рівня, які полегшують процес програмування за рахунок безлічі готових стандартних рішень. До однієї з таких мов відноситься Pascal, за допомогою які ми будемо вивчати та засвоювати техніку програмування.

1.5. Етапи розробки програми

Розробка будь-якої програми, від нескладного учбового завдання до професійного програмного продукту, може бути розбита на низку етапів. Коротко опишемо і охарактеризуємо їх.

1. Визначення вхідних та вихідних даних, вимог до програми – що завдано і що треба отримати, який буде спосіб взаємодії (*інтерфейсу*) програми з користувачем, якою мовою та в якій системі програмування вона розроблятиметься, які вимоги потрібні до апаратного та системного програмного забезпечення комп'ютерів, на яких має виконуватись програма.

2. Розробка алгоритму – визначення послідовності дій, які ведуть до розв'язання задачі та запис їх в одній із вказаних вище форм.

3. Кодування (програмування) – перетворення алгоритму на одну з мов програмування та створення *вихідного тексту програми* в одній із систем програмування. Програма на будь-якій мові складається з *операторів* – так називаються окремі дії, які передбачені в мові. Кількість операторів у будь-якій мові обмежена, а правила їх написання (синтаксис) жорстко визначені.

4. Компіляція та відладгодження – вихідний текст на мові Pascal безпосередньо не виконуватиметься комп'ютером – для роботи програми його потрібно *відкомпілювати*, тобто, перевести у машинний код. Цю роботу виконує спеціальна програма-компілятор або *оболонка* мови. Оболонка Pascal, за допомогою якої ми розроблятимемо свої програми, називається Turbo Pascal версії 7.0 (7.1). Вона створена компанією Borland International у 1983-97 рр.

Трохи про компілятор, який краще всього використовувати для перших практичних дослідів – це Free Pascal. Чому саме він? Та все просто – це абсолютно безкоштовний компілятор і середовище розробки, за допомогою якої можна розробляти і компілювати програми.

Free Pascal Compiler (FPC) – це вільно поширюваний компілятор мови Паскаль з відкритими вихідними кодами. А це означає, що ви можете розробляти з його допомогою будь-які програми, в тому числі і для комерційних цілей. Він сумісний з Borland Pascal 7 і Object Pascal – Delphi, але при цьому володіє рядом додаткових можливостей. FPC – багатоплатформовий інструмент, що підтримує велику кількість платформ. Серед них – DOS, Linux, FreeBSD, OS/2, MacOS(X) и Win32.

При перетворенні компілятором тексту програми у машинний код отримуємо *файл до виконання*, який можна запустити (*виконати*) в тій *операційній системі* (ОС), для якої розроблений компілятор. Наша оболонка Pascal створювалася для ОС MS-DOS, проте, у сучасних ОС родини Windows програма, яка написана на мові Pascal, працювати все ж таки буде, але без зручних інтерфейсних можливостей Windows.

У загальному випадку **компіляція** – це процес перетворення програми у *машинний код* спеціального формату. Зазвичай після етапу компіляції файл,

який містить відтрансльовану програму, називається файлом ***об'єктного коду*** програми.

З'єднати файл об'єктного коду з іншими файлами об'єктного коду та скомпонувати з них єдину програму – це функція спеціальної програми – ***компоновника, або редактора зв'язків***. Наприклад, програми на Pascal зазвичай використовують бібліотеки. Бібліотека Pascal містить сукупність об'єктних кодів комп'ютерних програм (підпрограм), які називаються *функціями*, та призначені для виконання таких завдань, як відображення інформації на екрані або обчислення квадратного кореня числа. При компонуванні об'єктний код програми об'єднується з *об'єктними кодами функцій*, які використовуються програмою, та певним стандартним *кодом початкового завантаження*, внаслідок чого створюється версія програми до виконання. Цей файл, який містить остаточний продукт, називається ***кодом до виконання***.

Дії з отримання кодів до виконання програми можуть розрізнятися. Але у загальному випадку ці дії можна проілюструвати наступною схемою (Рис. 1.2).

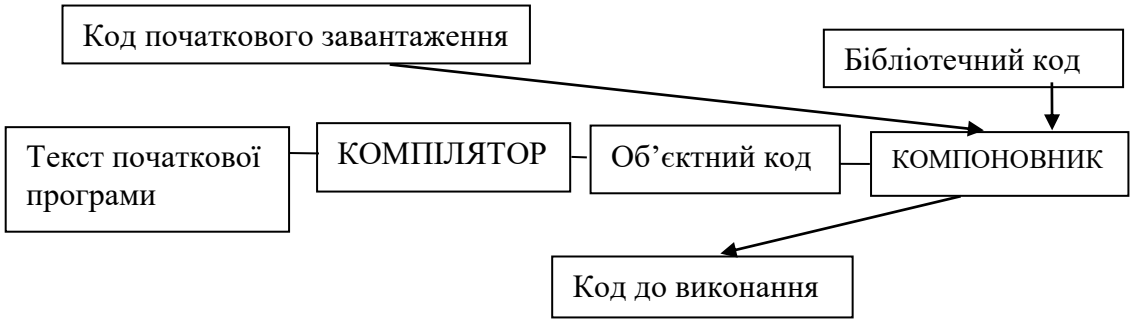


Рис. 1.2. Етапи створення коду програми до виконання

Програма, яку вдалося відкомпілювати та отримати код до виконання, не обов'язково працюватиме правильно. Вона може містити помилки, для виявлення яких призначений етап **відлагодження –** пошуку помилок у програмі та їх виправлення. Як правило, компіляція та відлагодження виконуються програмістом у тісному взаємозв'язку.

Можливі програмні помилки бувають 3-х видів: **синтаксичні** (помилки в правилах мови), **алгоритмічні** (помилки в логіці програми) та **помилки часу виконання**, які виникають у процесі роботи запущеної програми. Природно, що компілятор здатний знайти лише синтаксичні помилки, а для виявлення алгоритмічних помилок призначений етап **тестування** програми. Помилки часу виконання виникають як результат некоректних дій користувача, некоректних операцій над даними (наприклад, некоректної організації циклу) або помилок програмного та апаратного забезпечення ЕОМ.

5. Тестування – перевірка правильності роботи програми на наборах тестових даних із заздалегідь відомим результатом. Звичайно ж, тестування "усієї програми одразу" можливо лише для нескладних навчальних задач.

Реальні програми, як правило, тестуються "по частинах" – окремими функціями та модулями.

6. Документування та підтримка – цей етап включає створення довідкової системи та документації до програми, можливо, розширення її функціональності, випуск нових версій, виправлення помилок, які практично неминучі у будь-якій складній програмній системі. Для навчальних задач етап підтримки, звичайно, буде відсутній.

Контрольні запитання для самоперевірки

1. Що являє собою алгоритм?
2. Які основні властивості алгоритму?
3. Які існують форми подання алгоритму?
4. Що являє собою програма та програмне забезпечення?
5. Які існують види програмного забезпечення?
6. З яких етапів складається розробка програми?
7. Для чого призначений компоновник?
8. Які види помилок виникають при створенні програм?

2. Знайомство з середовищем TURBO – PASCAL

Система програмування TURBO – PASCAL – це сукупність двох до певної міри самостійних додатків компілятора з мови програмування Pascal та деякої інструментальної програмної оболонки, яка сприяє підвищенню ефективності щодо створення програм. Скорочено домовимось надалі називати мову програмування, яка підтримується компілятором Pascal – мовою TURBO – PASCAL, а різноманітні сервісні послуги, які надаються програмною оболонкою, – середовищем TURBO-PASCAL.

Середовище TURBO–PASCAL – це перше, з чим стикається програміст, який приступає до практичної роботи із системою. У цьому розділі наводяться мінімальні відомості про основні прийоми роботи в середовищі TURBO – PASCAL. Найбільш повні відомості про нього можна почерпнути у спеціальних книжках, якими забиті всі прилавки книжкових магазинів.

2.1. Як розпочати роботу з TURBO-PASCAL

При розгортанні системи на жорсткому диску зазвичай створюється каталог з ім'ям *TP* (або *PAS, TURBOPAS, PASCAL* і таке інше), в якому розміщюються всі файли з системи TURBO – PASCAL. Для виклику TURBO-PASCAL необхідно знайти у деревоподібній структурі каталогів ПК цей каталог, а в ньому файл *TURBO.EXE*. Цей файл містить готову до роботи діалогову систему програмування TURBO–PASCAL. До нього входять мінімально необхідні частини TURBO-PASCAL (текстовий редактор, компілятор, компоновник та завантажувач). Для нормальної роботи у діалоговому середовищі знадобляться також основна бібліотека, яка розташована у файлі *TURBO.TPL*, та довідкова служба (файл допомоги *TURBO.HLP*). В принципі, цих файлів цілком достатньо для написання, компіляції та виконання більшості програм.

Нехай перераховані файли розташовані у каталозі *TP* на диску *D*. Тоді для виклику TURBO-PASCAL слід виконати команду: *D:\TP\TURBO* або запустити Pascal з Робочого столу, якщо для нього є ярлик запуску.

Не рекомендується працювати з системою, призначивши каталогом за замовченням (поточний каталог) той, в якому зберігаються файли Pascal (цей каталог будемо називати системним). По-перше, у такому випадку можна помилково стерти будь-який з файлів системи програмування і тим самим порушити її працездатність, а, по-друге, цей каталог може дуже скоро заповнитись іншими файлами, які прямо не відносяться до TURBO-PASCAL. Існує ще одна причина, з якої небажано працювати у системному каталозі. Річ у тім, що TURBO–PASCAL має властивість запам'ятовувати своє налаштування у двох файлах з іменами *TURBO.TP* та *TURBO.PCK*. В разі виклику система починає пошук цих файлів у поточному каталозі. Якщо цей каталог – Ваш індивідуальний, система кожного разу буде налаштовуватись так, як Ви цього

хочете. Якщо ці файли не виявлені у Вашому каталозі (а при першому зверненні до TURBO-PASCAL так воно і буде), система продовжить пошук в системному каталозі, а не знайшовши їх там, буде налаштовуватись стандартним чином. Згодом можна зберегти файли налаштування у своєму каталозі і тим самим позбавити себе від необхідності переналаштовувати систему кожного разу при зверненні до неї.

Одразу ж скажемо, що для виходу з TURBO-PASCAL слід натиснути клавішу *Alt* і, не відпускаючи її, – клавішу з латинською буквою *X*, після чого можна відпустити обидві клавіші.

Верхній рядок системи TURBO-PASCAL містить «меню» можливих режимів роботи, нижній – коротку довідку про призначення основних функціональних клавіш. Решта всієї частини екрану належить вікну редактора, який окреслений подвійною рамкою та призначений для введення та корекції текстів програм. В його верхньому рядку висвітлюється ім'я того дискового файлу, звідки був прочитаний текст програми (новому файлу присвоюється ім'я *NONAME00.PAS*), два спеціальні поля, які використовуються при роботі з пристроєм введення «миша» (ці поля виділені квадратними дужками), та цифра 1 – номер вікна. У TURBO-PASCAL можна працювати одночасно з кількома програмами (або частинами однієї крупної програми), кожна з яких може розташовуватися в окремому вікні редактора. Середовище дозволяє використовувати до дев'яти вікон редактора одночасно.

Окрім вікна (вікон) редактора у TURBO-PASCAL використовуються також вікна режиму налагодження, виведення результатів роботи програми, довідкової служби, стека, регістрів. За бажанням вони можуть викликатися на екран по черзі або бути присутніми на ньому одночасно.

2.2. Функціональні клавіші

Функціональні клавіші використовуються для управління середовищем TURBO-PASCAL. Вони позначаються F1, F2 ..., F12 і розташовуються у верхньому ряду клавіатури. З кожною з цих клавіш пов'язується певна команда меню. Дію майже всіх функціональних клавіш можна модифікувати трьома особливими клавішами: *Alt* (від ALTernative – *додатковий*), *Ctrl* (ConTRoL – *управляючий*) та *Shift* (SHIFT - *зсувний*). Ці клавіші використовуються подібно клавіші тимчасової зміни регістра на друкарській машинці: потрібно натиснути на одну з них і потім, не відпускаючи її, натиснути функціональну клавішу. Надалі таке спільне натиснення двох клавіш позначатимемо символом «+». Наприклад, *Alt+F3* означає, що разом з клавішею *Alt* необхідно натиснути клавішу F3, *Ctrl+F9* – разом з *Ctrl* натискається F9 і так далі

Нижче наводяться команди, які передаються середовищу TURBO-PASCAL за допомогою функціональних клавіш та деякими їх комбінаціями з клавішами *Ctrl* та *Alt*:

Fl – звернутися за довідкою до вбудованої довідкової служби (*Help-допомога*);

F2 – зберегти редагований текст на дисковий файл;

F3 – прочитати текст з дискового файлу у вікно редактора;

F4 – використовується у режимі налагодження: розпочати або продовжити виконання програми та зупинитися перед виконанням того її рядка, на який вказує курсор;

F5 – розкрити активне вікно на весь екран;

F6 – зробити активним наступне вікно;

F7 – використовується у режимі налагодження: виконати наступний рядок програми; якщо в рядку є звернення до процедури (функції), звернутись до цієї процедури та зупинитись перед виконання першого її оператора;

F8 використовується в режимі налагоджування: виконати наступний рядок програми; якщо у рядку є звернення до процедури (функції), виконати її та не простежувати її роботу;

F9 – компілювати програму, але не виконувати її;

F10 – перейти до діалогового вибору режиму роботи за допомогою головного меню;

Ctrl+F9 – виконати прогін програми: компілювати програму, яка знаходиться у вікні редактора, завантажити її в оперативну пам'ять та виконати, після чого повернутися в середовище TURBO-PASCAL.

Alt+F5 – змінити вікно редактора на вікно виведення результатів роботи (прогону) програми.

Дуже часто Вам знадобляться команди *Ctrl+F9* для перевірки роботи Вашої програми і *ALT+X* – для виходу з TURBO-PASCAL. Клавіші F2 і F3 допоможуть Вам у роботі з Вашим каталогом. За допомогою клавіш *Alt+F5* Ви у будь-який момент зможете переглянути дані, які виводяться на екран в результаті прогону програми.

2.3. Текстовий редактор

Текстовий редактор середовища TURBO-PASCAL надає користувачеві зручні засоби створення та редагування текстів програм. Ознакою того, що середовище знаходиться у стані редагування, є наявність у вікні редактора курсору – невеликого миготливого прямокутника. Режим редагування автоматично встановлюється одразу після завантаження TURBO-PASCAL. З режиму редагування можна перейти до будь-якого іншого режиму роботи TURBO-PASCAL за допомогою функціональних клавіш або вибору потрібного режиму з головного меню. Якщо середовище знаходиться у стані вибору з

меню, курсор зникає, а в рядку меню з'являється кольоровий покажчик-прямокутник, який виділяє одне з кодових слів (опцій меню). Для переходу з головного меню у стан редагування потрібно натиснути клавішу *Esc* (*ESCape – покинути, тікати*), а для переходу до вибору функцій з головного меню – *F10*.

Розглянемо основні прийоми роботи з текстовим редактором.

Для створення тексту програми потрібно ввести цей текст за допомогою клавіатури ПК подібно до того, як це робиться при друкуванні тексту на друкарській машинці. Після заповнення чергового рядка слід натиснути на клавішу Enter, щоб перевести курсор на початок наступного рядка (курсор завжди вказує те місце на екрані, куди буде розміщений черговий символ програми, що вводиться).

Вікно редактора імітує довгий та достатньо широкий лист паперу, фрагмент якого видно у вікні. Якщо курсор досяг нижнього краю, здійснюється прокрутка (скролінг) вікна редактора: його вміст зміщується вгору на один рядок і внизу з'являється новий рядок листа. Якщо курсор досяг правої межі екрану, вікно починає по мірі введення символів зміщуватися праворуч, показуючи правий край листа. Розміри листа по горизонталі та вертикалі обмежуються тільки загальним числом символів у файлі, яких не повинно бути більше 65535, проте компілятор TURBO-PASCAL сприймає рядки програми завдовжки не більше 126 символів.

Вікно можна зміщувати щодо листа за допомогою наступних клавіш:

Page Up – на сторінку назад;

Page Down – на сторінку вперед;

Home – на початок поточного рядка;

End – у кінець поточного рядка;

Ctrl+Page Up – на початок тексту;

Ctrl+Page Down – у кінець тексту.

Якщо Ви помилилися при введенні чергового символу, його можна стерти за допомогою клавіші із стрілкою (або написом *Backspace*), яка розташована над клавішею Enter. Клавіша *Delete* усуває символ, на який в даний момент вказує курсор, а команда *CTRL+Y*- усуває весь рядок, на якому розташований курсор.

Слід пам'ятати, що редактор TURBO-PASCAL вставляє у кінці кожного рядка два невидимі символи-роздільники. Ці символи вставляються при натисненні клавіші *Enter*, а усуваються клавішами *Backspace* або *Delete*. За допомогою вставки/вилучення розподільників можна «розрізати»/«склеювати» рядки. Щоб розрізати рядок, слід підвести курсор до потрібного місця і натиснути *Enter*, щоб склеїти сусідні рядки, потрібно встановити курсор в кінець першого рядка (для цього зручно використовувати клавішу *End*) та натиснути *Delete* або встановити курсор на початок другого рядка (клавішею *Home*) таі натиснути *Backspace*.

Нормальний режим роботи редактора – режим вставки, в якому кожен символ, що знов вводиться, як би «розсовує» текст на екрані, зміщуючи праворуч залишок рядка. Слід враховувати, що розрізання тексту та подальша вставка пропущених рядків можливі тільки у цьому режимі. Редактор може також працювати у режимі накладання нових символів на існуючий старий текст: у цьому режимі новий символ замінює собою той символ, на який вказує курсор, а залишок рядка не зміщується праворуч. Для переходу до режиму накладання треба натиснути клавішу *Insert*, а якщо натиснути цю клавішу ще раз, знов встановлюється режим вставки. Ознакою того, в якому режимі працює редактор, є форма курсору: Для режиму вставки він схожий на миготливий символ підкреслення, а у режимі накладення він є крупним миготливим прямокутником, що затуляє символ цілком.

І ще про одну можливість редактора. Зазвичай редактор працює у режимі автовідступу. У цьому режимі кожен новий рядок починається з тієї ж позиції на екрані, що і попередній. Режим автовідступу підтримує хороший стиль оформлення тексту програми: відступи від лівого краю виділяють тіло умовного або складеного оператора та роблять програму наочнішою. Відмовитися від автовідступу можна клавішами *CTRL+O I* (притиснутій *Ctrl* натискається спочатку клавіша з латинською буквою *O*, а потім *O* відпускається і натискається *I*), повторна команда *CTRL+O I* відновить режим автовідступу.

Приємно, звичайно, коли на екран видається інформація російською або українською мовою, а не транслітерацією або корявою англійською мовою. Аби на екран видати російський текст, набирайте вашу програму в редакторі **NotePad** (або **AkelPad**), та оберіть управляючу кнопку меню *Сохранить как …* з кодуванням 866 (**OEM**-русский).

2.4. Основні прийоми роботи в середовищі TURBO-PASCAL

2.4.1. Робота з файлами

Основною формою зберігання текстів програм поза комп’ютерним середовищем є файли. Після завершення роботи з TURBO-PASCAL можна зберегти текст нової програми у файлі на диску для того, щоб використовувати його наступного разу. Для обміну даними між дисковими файлами і редактором середовища призначені клавіші F2 (запис у файл) і F3 (читання з файлу). Якщо Ви створюєте нову програму, то середовище ще не знає імені того файлу, в який Ви захочете помістити текст цієї програми, і тому воно присвоює йому стандартне ім’я *NONAME00.PAS* (*NO NAME* – немає імені). Для збереження тексту програми у файлі потрібно натиснути F2. У цей момент середовище перевірить ім’я програми і, якщо це стандартне ім’я *NONAME*, запитає, чи потрібно його змінювати: на екрані з’явиться невелике вікно запиту з написом Save File as (Зберегти файл з ім’ям...)

Нижче під цим написом розташовується поле для введення імені файлу, в якому можна написати потрібне ім’я і натиснути Enter, – текст буде збережений

у файлі. Якщо в імені немає розширення, середовище присвоить файлу стандартне розширення PAS. Якщо Ви захочете завершити роботу з TURBO-PASCAL, а в редакторі залишився не збережений у файлі текст, на екрані з'явиться вікно із запитом:

NONAMEOO.PAS has been modified.Save? (Файл *NONAME00.PAS* був змінений. Зберегти?) У відповідь слід натиснути Y (*Yes* – так), якщо необхідно зберегти текст у файлі, або N (*No* – ні), якщо зберігати текст не потрібно.

2.4.2. Прогін і відлагодження програми

Після підготовки тексту програми можна спробувати виконати її, тобто відкомпілювати програму, з'єднати її (якщо це необхідно) з бібліотекою стандартних процедур і функцій, завантажити в оперативну пам'ять та передати їй управління. Вся ця послідовність дій називається прогоном програми і реалізується командою *Ctrl+F9*.

Якщо в програмі немає синтаксичних помилок, то всі ці кроки виконуватимуться послідовно один за одним, при цьому у невеликому вікні повідомляється про кількість рядків, що відкомпілювалися, та об'єм доступної оперативної пам'яті. Перед передачею управління завантаженій програмі середовище очищує екран (точніше, виводить на екран вікно прогону програми), а після завершення роботи програми знов бере управління комп'ютером на себе і відновлює на екрані вікно редактора.

Якщо на будь-якому етапі середовище виявить помилку, то подальші дії припиняються, відновлюється вікно редактора і курсор розміщується на той рядок програми, де при компіляції або виконанні була виявлена помилка. При цьому у верхньому рядку редактора з'являється діагностичне повідомлення про причину помилки. Все це дозволяє дуже швидко відлагодити програму, тобто усунути в ній синтаксичні помилки та переконатися у правильності її роботи. Якщо помилка виникла на етапі прогону програми, проста вказівка того місця, де вона виявлена, може не дати потрібної інформації, оскільки помилка може бути наслідком неправильної підготовки даних у попередніх операторах програми.

Наприклад, якщо помилка виникла в результаті обчислення квадратного кореня з від'ємного числа, буде вказаний цей оператор, хоча зрозуміло, що першопричину помилки треба шукати десь раніше, там, де відповідною змінною присвоюється від'ємне значення. У таких ситуаціях зазвичай удаються до покрокового виконання програми за допомогою команд, що пов'язані з клавішами F4, F7 (з входом у функції і процедури, або F8 – без входу).

Поки не накопичений достатній досвід відлагодження, можна скористатися однією клавішею F7, після натиснення на яку середовище здійснить компіляцію, компоновку (зв'язок з бібліотекою стандартних процедур і функцій) і завантаження програми, а потім зупинить прогін перед виконанням першого оператора. Рядок програми, що містить цей оператор, буде виділений на екрані покажчиком (кольором). Тепер кожне нове натиснення F7

викликатиме виконання всіх операцій, які запрограмовані у поточному рядку, і зсув покажчика до наступного рядка програми. У підозрілому місці програми можна проглянути поточне значення змінної або виразу. Для цього потрібно встановити курсор в те місце рядка, де знаходиться ім'я змінної, що Вас цікавить, і натиснути *Ctrl+F4*. На екрані з'явиться діалогове вікно, яке складається з трьох полів (у верхньому полі стоятиме ім'я змінної, два інших поля будуть порожніми). Натисніть *Enter*, щоб у середньому полі отримати поточне значення змінної. Якщо перед натисненням *Ctrl+F4* курсор стояв на порожній ділянці рядка або вказував на ім'я іншої змінної, верхнє поле діалогового вікна також буде порожнім або містити ім'я цієї іншої змінної. У цьому випадку слід ввести за допомогою клавіатури ім'я потрібної змінної і натиснути Enter. До речі, таким чином можна вводити не тільки імена змінних, що простежуються, але і вирази – середовище обчислить і покаже значення введеного виразу.

2.4.3. Довідкова служба TURBO –PASCAL

Невід'ємною складовою частиною середовища TURBO–PASCAL є вбудована довідкова служба. Якщо Ви досить добре володієте англійською мовою, у Вас не буде проблем при роботі з TURBO-PASCAL: у скрутній ситуації досить натиснути F1 і на екрані з'явиться необхідна довідка. Ця довідка залежить від поточного стану середовища, тому довідкову службу називають контекстно-чутливою. Наприклад, якщо натиснути F1 у мить, коли середовище виявило помилку в програмі, в довідці будуть повідомлені додаткові відомості про причини помилки та надані рекомендації щодо її усунення.

Існують чотири способи звернення до.довідковій служби безпосередньо з вікна редактора:

F1 – отримання контекстно-залежної довідки;

Shift+F1 – вибір довідки із списку доступних довідкових повідомлень;

Ctrl+F1 – отримання довідки про потрібну стандартну процедуру, функцію, про стандартну константу або змінну;

Alt+F1 – отримання попередньої довідки.

По команді *Shift+F1* на екрані з'явиться вікно, яке містить впорядкований за абеткою список стандартних процедур, функцій, типів, констант і змінних з довідковою інформацією.

Контрольні запитання для самоперевірки

1. Як запустити середовище TURBO–PASCAL?
2. Які компоненти входять до середовища TURBO–PASCAL?

3. Для чого призначені файли TURBO.TP та TURBO.PCK?
4. Які види вікон існують у TURBO–PASCAL?
5. Яке призначення функціональних клавіш у редакторі?

3. Знайомство з мовою TURBO–PASCAL

У цьому розділі посібника описується ядро TURBO-PASCAL – мінімальний набір засобів, достатній для створення порівняно простих програм. Команди, з яких складається програма на Pascal, і багатьох інших мовах, називаються **операторами**. Багато операторів на Pascal є **зверненнями до функцій**, які вже вбудовані в Pascal. Детальніше про зміст цих назв поговоримо пізніше, а поки не робитимемо між ними відмінності і все підряд називатимемо операторами. Кожен новий вивчений оператор відкриватиме перед вами нові можливості Pascal, тому поставимо завдання спершу вивчити більше операторів на прикладах їх роботи у простих програмах і тільки тоді перейдемо до складніших програм.

Зокрема, ми розглядатимемо оператори, що найбільш використовуються, найбільш популярні типи даних та операції над ними. Ви познайомитеся з прийомами розробки процедур і функцій, які дозволяють створювати структуровані програми. У завершальній частині розділу на прикладах показано застосування методики низхідного програмування, що забезпечує порівняно простий і надійний спосіб детального опрацьовування алгоритму програми.

Мета цієї глави – навчити вас створювати програми, спочатку прості, потім усе складніші. А головна мета – досягти у вас відчуття того, що тепер ви можете *самостійно* створювати програми *складніші*, ніж в цій главі.

3.1. Ваша перша програма

Мова програмування, як і будь-яка інша мова (наприклад, природна), призначена для комунікації, тобто зв’язком між тим, хто говорить і хто слухає. У програмуванні говорить програміст, а слухачем є інтерпретатор мови, деяка комп’ютерна програма, що розуміє цю мову і що виконує дії відповідно до того, що вона зрозуміла.

Був час, коли вважалося, що для полегшення спілкування з комп’ютером необхідно створити мову, досить близьку до природної. Ця ідея в кіні кінців не витримала випробувань часом, хоча і породила декілька чудових мов програмування (наприклад – Кобол). Часто буває, що побічні ефекти деякої діяльності перевершують очікування. Ми зазнаємо труднощів спілкування і природною мовою; що вже говорити про спеціалізовані мови, тим паче про мови спілкування з комп’ютером. Проблема вирішується на подив дуже просто. Головне тут – зрозуміти корінь або причину проблеми. Вся складність полягає не у мові, а у культурі або способі її вживання. Врешті-решт, мова є лише нормативною базою, а люди використовують її лише так і в тій мірі, як змогли або як їм здалося достатнім для своїх життєвих потреб.

Практика використання мови стимулює розвиток її нормативної бази і тим самим підтримує його існування. Що ж до штучних мов, тобто мов

програмування, для яких нормативна база мови обмежена (набір **ключових, зарезервованих слів, констант, змінних, тощо**), то труднощі їх заосвоєння новачками обумовлені, головним чином, нестачею практики та культури програмування. Стандартний (формальний) спосіб опису мови хороший для тих, хто вже вивчав будь-яку мову програмування. Для новачків же стандартний, тим більше формальний спосіб викладу правил мови є, м'яко кажучи, неприйнятним. Ми осягаємо щось інакше, ніж викладаємо те, що вже освоєне. Це добре відома істина, на жаль, чомусь часто забувається як учнями, так і викладачами. Проте автори майже всіх настанов з мов, які здобули широке визнання і які добилися успіху, слідували цьому основному принципу.

А тепер для знайомства з мовою TURBO-PASCAL спробуємо скласти нескладну (елементарну) програму, що здійснює виведення будь-якого повідомлення на екран ПК. Хай це буде фраза «Я програмую на TURBO-PASCAL». Ось можливий варіант такої програми:

Приклад 3.1

```
Program My_First_Program;
Const
Text = 'Я програмую на TURBO-PASCAL';
begin
    WriteLn(Text);
    Readln;
end.
```

Перш за все, проаналізуємо форму представлення тексту. У програмі сім рядків. Рядки програми зазвичай виділяють деякі смислові фрагменти тексту і можуть не зв'язуватися з конкретними діями у програмі: розташування тексту програми по рядках – справа смаку програміста, а не вимога синтаксису мови. Ту ж саму програму можна було б написати, наприклад, так:

```
Program  My_First_Program; const Text =
'Я програмую на TURBO-PASCAL';begin WriteLn(Text); end.
```

Але так писати не треба.

На відміну від деяких інших мов програмування символ пропуску у мові TURBO – PASCAL використовується як роздільник окремих конструкцій мови, тому програма

```
PROGRAM My_First_Program;constText=
'Я програмую на TURBO-PASCAL';BEGINWriteLn(Text);end.
```

буде невірною.

У TURBO-PASCAL ігнорується відмінність у висоті букв (заголовні або строкові), якщо тільки це не пов'язано з текстовими константами. Початок програми міг би, наприклад, виглядати так:

```
program my_first_program;
```

Тепер опишемо значення кожного із рядків.

Перший рядок Program My_First_Program; нічого не виконує, він просто містить назву програми. Починається програма словом Program, за яким слідує ім'я програми. Слово Program ***зарезервоване*** у TURBO-PASCAL, тобто не може використовуватися ні в яких інших цілях, окрім як для оголошення імені програми. У TURBO-PASCAL є безліч зарезервованих слів (див. гл. 3). Будь-яке з них не можна використовувати як ідентифікатор (ім'я) будь-якого об'єкту програми – змінної, константи тощо.

У TURBO-PASCAL оператор заголовка програми може бути опущений. Ім'я програми ніколи в ній фактично не використовується, і воно зовсім не пов'язане з ім'ям зовнішнього файлу, що містить текст програми.

Відмітимо, що редактор середовища TURBO – PASCAL зазвичай виділяє зарезервовані слова кольором. Оскільки ім'я програми ніяк надалі не використовується, вимога його оголошення здається зайвою. У TURBO-PASCAL можна опускати оголошення імені оператором Program без будь-яких наслідків для програми.

У даному прикладі ім'я My_First_Program є не що інше, як англійська фраза «Моя Перша Програма», але тільки написана без пропусків (пробілів) – пропуск є роздільником і не може використовуватися довільно (замість пропусків в ідентифікаторах дозволяється використовувати символ підкреслення). Рядок Program з ім'ям програми в Паськаль-программах може бути відсутнім. Це ніяк не позначається на синтаксис і роботу програми, але все таки програма буде більш "читабельною", коли у цьому рядку буде ім'я програми, яке передає її призначення.

Перший рядок закінчується особливим роздільником – крапкою з комою. Цей роздільник в мові TURBO – PASCAL відзначає кінець оператора або опису. Використання особливого роздільника дозволяє розташовувати декілька операторів на одному рядку. У загальному випадку кожен оператор повинен закінчуватися крапкою з комою. Є тільки декілька виключень з цього правила. Одне з найчастіших таких виключень – це конструкція begin … end.

Важко читати текст програми навіть про себе, якщо не знаєш, як вимовляти спеціальні символи та ключові слова. Тому в дужках ми будемо описувати, як вимовляти деякі ключові слова Pascal.

За зарезервованим словом begin (читається "бегин") ніколи не слідує крапка з комою, так само як вона ніколи не слідує перед зарезервованим словом end (можна і поставити, але це буде просто порожній оператор). Для того щоб зрозуміти, чому конструкція begin ... end є спеціальним випадком, можна порівняти begin і end з операторними дужками. Тому з тих самих міркувань не слід ставити крапку з комою після begin і перед end. Відзначимо, що ці правила дотримані у прикладі 3.1.

Другий рядок const містить єдине зарезервоване слово const, яке означає, що далі будуть описано одна або декілька констант (CONSTants – константи). Константами в мові вважаються такі об'єкти програми, які не

можуть змінювати свого значення. На відміну від багатьох інших мов програмування, константа в TURBO-PASCAL може мати власне ім'я, що відповідає прийнятій в наукових і інженерних розрахунках практиці іменування часто використовуємих констант. Наприклад, з школи ми пам'ятаємо про існування константи pi = 3.1415926. При обробці програми ім'я константи pi замінюватиметься компілятором на її значення. Описати константу в TURBO-PASCAL – означає вказати її ім'я і значення. Така вказівка міститься в третьому рядку

Text = 'Я програмую на Turbo-Pascal'; у якій константі з ім'ям Text присвоюється як значення рядок символів «Я програмую на Turbo-Pascal».

У TURBO-PASCAL можуть використовуватися константи різного типу – цілі або дійсні числа, символи, рядки символів, масиви тощо. Ознакою того, що Text є константою типу рядок символів, служать два апострофи (одинарні лапки), що оирчують рядок, причому самі апострофи цьому рядку не належать, а лише вказують компілятору на те, що всі заключені в них символи слід розглядати як єдине ціле – текстову константу. Якщо знадобиться включити сам апостроф в текстову константу, достатньо його написати двічі підряд. Наприклад, опис

Text = 'Turbo' 'Pascal'; створить константу із значенням Turbo'Pascal.

Все три перші рядки не пов'язані з будь-якими конкретними діями при роботі програми. Вони повідомляють компілятору деякі відомості про саму програму і об'єкти, що використовуються в ній. Речення алгоритмічної мови, в яких даються відомості про типи даних (у нашому випадку – текстова константа), називаються описами або невиконуваними операторами. Ця частина програми називається розділом описів.

Зарезервоване слово **begin** в **четвертому рядку** сигналізує компілятору про початок іншої частини програми – розділу операторів. У нашому прикладі цей розділ містить оператор **WriteLn(Text);** який, власне, і виводить повідомлення на екран комп'ютера.

Оператор **Readln;** – натиснення клавіші «Enter» (« введення»), він потрібний для затримки, щоб можна було побачити результат програми, а потім треба натиснути «Enter» та повернутися в Turbo-Паскаль (замість натиснення клавіш ***Alt+F5*** **–** змінити вікно редактора на вікно виведення результатів роботи програми).

Ми будемо називати Readln і WriteLn операторами, зважаюми на те, що це спеціальні Pascal-процедури введення-виведення.

Завершує всю програму зарезервоване слово **end з крапкою.** Крапка оповіщає компілятор про кінець тексту програми. За поєднанням end. можна розміщувати який завгодно текст – він не оброблятиметься компілятором.

Перш ніж спробувати відкомпілювати і виконати нашу програму, обговоримо її єдиний оператор, що виконується.

WriteLn(Text); Оператор WriteLn (читається "райт'лайн", перекладається – пиши рядок) наказує комп'ютеру відобразити певні дані на терміналі

введення-виведення користувача. Оператор Write та його відмінність від WriteLn ми розглянемо у наступному розділі 3.4.

Цікаво, що в Pascal взагалі і TURBO-PASCAL, зокрема, немає спеціальних операторів введення-виведення. Для обміну інформацією з навколишнім світом в програмах, які написані на мові TURBO–PASCAL, використовуються спеціальні стандартні процедури. Таким чином, за своєю суттю оператор WriteLn(Text); є оператором звернення до вбудованої процедури виведення даних (свою назву вона отримала від WRITE LiNe – записати рядок).

Поняття процедури (див. гл. 6) – одне з центральних понять TURBO–PASCAL. Процедура – це деяка послідовність операторів, до якої можна звернутися по імені. Кожного разу, коли ми називаємо в операторові ім'я процедури, ініціюється послідовність запрограмованих в ній дій.

Процедура WriteLn відноситься до стандартних або вбудованих процедур TURBO-PASCAL. Стандартна процедура не потребує попереднього опису, вона доступна будь-якій програмі, в якій міститься звернення до неї. Різниця між оператором виведення і зверненням до процедури виведення полягає у тому, що ім'я процедури виведення, як і будь-якої іншої процедури TURBO-PASCAL, не є зарезервованим словом, а отже, користувач може написати свою власну процедуру з ім'ям WriteLn. Втім, ця можливість для більшості користувачів залишається лише мовною тонкіщу і дуже рідко використовується на практиці.

Процедура WriteLn – одна з небагатьох процедур TURBO-PASCAL, при зверненні до яких дозволяєится використання довільної кількості параметрів. Параметри, які перераховані через кому, передаються процедурі у вигляді списку, який розташований в круглих дужках одразу за ім'ям процедури. У нашому прикладі процедурі передається єдиний параметр – константа Text. Як ми побачимо далі (див. гл. 5), найпершим параметром при зверненні до процедури WriteLn можна вказати адресу приймача інформації – пристрій або дисковий файл, в який спрямоване виведення. У такий спосіб програміст може легко переадресувати виведення даних. Якщо, як це зроблено у нашому прикладі, адреса виведення не вказана, виведення прямує на екран дисплея.

Всі чотири слова (**Program, const, begin** та **end**), що використовувалися у програмі, є зарезервованими. Слово WriteLn, як вже наголошувалося, не відноситься до зарезервованих, але навряд чи може виникнути необхідність перевизначити його, оскільки в цьому випадку програма позбудеться могутнього і зручного засобу виведення даних. Два слова My_First_Program і Text служать ***ідентифікаторами*** (іменами) деяких об'єктів програми. Програміст може використовувати як ідентифікатори будь-які послідовності символів, які задовольняють наступним обмеженням:

- ідентифікатор може складатися з букв латинського алфавіту, цифр, знаку підкреслення;
- ніякі інші символи в ідентифікаторі неприпустимі;
- ідентифікатор не може починатися з цифри;
- ідентифікатор не може співпадати з жодним із зарезервованих слів;

– довжина ідентифікатора може бути довільною, але значущими вважаються перші 63 символи.

Аналізуючи програму, можна зробити деяку аналогію між алгоритмічною мовою і природною мовою. Звичайна розмовна мова складається з чотирьох основних елементів: символів, слів, словосполучень і речень. Алгоритмічна мова містить подібні елементи, лише слова називають елементарними конструкціями, словосполучення – виразами, речення – операторами. Символи, елементарні конструкції, вирази та оператори складають ієрархічну структуру, оскільки елементарні конструкції утворюються з послідовності символів. Вирази – це послідовність елементарних конструкцій та символів, а оператор – послідовність виразів, елементарних конструкцій та символів.

Символи мови – це основні неподільні знаки, в термінах яких пишуться всі тексти на алгоритмічній мові.

Елементарні конструкції або лексеми (лексичні одиниці мови) – це мінімальні одиниці мови, що мають самостійний сенс. Вони утворюються з основних символів мови.

Вирази в алгоритмічній мові складаються з елементарних конструкцій та символів, вони задають правила обрахунку деякого значення.

Оператор задає повний опис деякої дії, яку необхідно виконати. Для опису складної дії може бути потрібна група операторів. У цьому випадку оператори об’єднуються в составний оператор або блок.

Дії, які задані операторами, виконуються над даними. Оператори алгоритмічної мови, в яких даються відомості про типи даних, називаються описами або операторами які не виконуються.

Об’єднана єдиним алгоритмом сукупність описів та операторів утворює програму на алгоритмічній мові.

Тепер спробуйте виконати програму. Для цього після набору її тексту натисніть Ctrl-F9. Якщо Ви не помилилися при введенні тексту, то після декількох секунд відмітите швидку зміну зображень на екрані: одразу після завантаження програми очищує екран, надаючи його у розпорядження працюючій програмі користувача. Такий екран називається вікном програми. Після завершення прогону (виконання програми часто називається її прогоном) на екрані знов з’явиться вікно редактора з текстом програми. Якщо Ви не встигли роздивитись зображення вікна програми (наприклад, забули про оператор readln), натисніть Alt+F5. Після натиснення на будь-яку клавішу середовище поверне екран в режим відтворення вікна редактора.

Наша перша програма My_First_Program, як і усі інші, є прикладом консольного застосування (додатку). Консоль – це універсальний пристрій введення/виведення, яким є дисплей разом з клавіатурою. Це застосування не має призначеного для користувача інтерфейсу, тобто списків, кнопок, вікон тощо. Текстове введення/виведення відбувається через стандартну консоль (звичайно це вікно командного рядка або вікно DOS на екрані комп’ютера). Той факт, що знайомство з мовою обмежується консольними додатками,

дозволяє спростити приклади у цьому посібнику і зосередити увагу власно на мові. Надалі ви будете вивчати як веб-, так і Windows-додаткі, і тоді ви за допомогою інструментальних засобів, таких як, наприклад Delphi, навчитеся розробляти призначений для користувача інтерфейс.

У даному прикладі програма My_First_Program усього лише виводить на монітор текст "Я програмую на Turbo-Pascal". Виведення на монітор здійснюється за допомогою вбудованої процедури виведення даних з ім'ям WriteLn. Ця процедура приймає як аргумент рядок (послідовність символів) і виводить його в стандартний потік введення/виведення. При запуску цієї програми на комп'ютерному моніторі з'являється вікно, командне або DOS, в якому знаходиться текст "Я програмую на Turbo-Pascal ".

Приклад 3.2. Програма обчислення суми двох чисел.

```
Program SUMS;
var
      Add1, Add2, Rezult: Integer;
begin
      Add1 := 3;
      Add2 := 4;
      Rezult := Add1 + Add2;
      WriteLn(Rezult)
end.
```

У будь-якій мові програмування дуже важливе поняття ***типу даних***. Якщо не усвідомити цього із самого початку, то потім доведеться часто стикатися з дивною поведінкою створеної вами програми. Тому ми рекомендуємо уважно віднестися до даного розділу і подальших розділів, в яких ми поступово будемо знайомити вас з різними типами даних.

Наступні після заголовка два рядки в прикладі 3.2 представляють ***розділ опису змінних***, що починається **зарезервованим** словом **var** (читається ВАР, це скорочення від англійського variable – змінна). Він вказує комп'ютеру, скільки змінних використовується у програмі, які імена у цих змінних, та дані якого типу зберігатимуться у цих змінних. Усі три змінні мають тип INTEGER, який означає, що змінні можуть приймати тільки цілочисельні значення.

Відмітимо, що для визначення змінних забороняється використання слів, які отримали тип «зарезервовані», вони мають в мові особливу роль. Нам вже зустрічалися зарезервовані слова: program, begin, end, integer, var, тощо.

Навіщо потрібний опис? Після того, як програміст ввів програму у пам'ять, він наказує комп'ютеру її виконати. Але комп'ютер при цьому не одразу починає виконувати програму, а спочатку здійснює **компіляцію,** тобто перетворення програми з мови Pascal на власно машинну мову. (Часто замість терміну «компіляція» використовують термін «трансляція»). Під час компіляції комп'ютер проводить деякі підготовчі дії, однією з яких є відведення у пам'яті місця під змінні величини, які згадані у програмі. При цьому комп'ютер «міркує» так. Оскількі у програмі згадана змінна величина, вона у кожен момент часу матиме якесь значення, яке, хочеш не хочеш, треба пам'ятати.

Тому краще заздалегідь відвести у пам'яті певне місце для запам'ятовування поточного значення кожної змінної величини і тільки потім вже виконувати програму.

Тепер нам зрозуміло, навіщо у програмі на Pascal потрібен опис – щоб перерахувати комп'ютеру змінні, під які він повинен відвести пам'ять. Якщо ми забудемо згадати в описі будь-яку змінну, то під неї в пам'яті не буде відведено місце і комп'ютер не зможе її запам'ятати, а, отже, не зможе з нею працювати. Pascal строгий до програміста, він змушує його самого перераховувати в описі усі змінні, які зустрічаються у програмі.

Який сенс вкладається у поняття «змінна»? Розглядатимемо змінну як «чорну скриню» (контейнер), де можуть зберігатися дані. Наприклад, змінна типу INTEGER може містити тільки цілі числа. Інші типи даних будуть розглянуті у подальшому. Дані, які зберігаються у змінній, називають значеннями цієї змінної. Змінна має ім'я – послідовність літер, цифр і символу підкреслення без пропусків і розділових знаків, яка починається обов'язково з букви або символу підкреслення. Приклади правильних імен змінних: myFamily, my_Name, _tel412_3456. Про це ми ще поговоримо детальніше.

На імена, які присвоюються змінним, накладаються ще деякі обмеження, які ми також розглянемо пізніше. Зараз відмітимо, що імена можуть нести смислове навантаження, наприклад REZULT, але можуть і не нести, наприклад ZBLMCH. Вочевидь, використання усвідомлених імен переважає, оскільки робить програму більш простішою для розуміння.

Що ж ми можемо робити із змінними величинами, програмуючи на мові Pascal? Перш за все, ми можемо задавати комп'ютеру значення тієї або іншої змінної величини. Це ми можемо зробити за допомогою нового оператора, який називається ***оператором присвоєння***. Перші три оператори прикладу 3.2 програми є операторами присвоєння (**:=**). Мета оператора присвоєння полягає у завданні деяких значень змінним, що стоять ліворуч від оператора присвоєння. Цими значеннями можуть бути математичні вирази, наприклад ADD1 + ADD2.

Праворуч від знаку **:=** в операторі присвоєння можна писати не лише числа, але і змінні величини та вирази. Вираз поки будемо розуміти так, як його розуміє шкільна математика. Наприклад, після виконання наступного фрагменту програми**: . . . *a:= 1+2*3; b:=a; c:=a+b+1. . .*** комп'ютер знатиме, що *a* дорівнює 7, *b* дорівнює 7, *c* рівне 15.

Що відбувається із старим значенням змінної, коли їй присвоюється нове значення? Воно просто стирається із пам'яті.

Потрібно мати на увазі, що зліва від знаку **:=** може стояти лише змінна величина, але не число і не вираз.

Перед тим, як рухатися далі, корисно докладніше ознайомитися з деякими можливостями середовища. Натисніть клавішу F10, щоб перейти до режиму вибору з головного меню, підведіть покажчик до опції Debug (відладка) і натисніть клавішу Enter – на екрані розкриється меню другого рівня, яке пов'язане з цією опцією. Нове меню як би «випало» з верхнього рядка, тому

таке меню часто називають випадним. Відшукайте у новому меню опцію Output (виведення програми), підведіть до неї курсор та натисніть клавішу Enter ще раз. На екрані знов з'явиться вікно програми, але воно вже не зникатиме після натиснення на будь-яку клавішу – екран буде пов'язаний з цим вікном постійно. Тепер доб'ємося того, щоб на екрані демонструвалися два вікна одночасно: знов натисніть клавішу F10, виберіть Window, натисніть клавішу Enter, підведіть курсор до опції Tile (черепиця) і натисніть клавішу Enter ще раз.

Подвійна рамка, що окреслює вікно програми, свідчить про те, що саме це вікно активне у даний момент. Зробимо активним вікно редактора: натиснемо клавішу Alt і, не відпускаючи її, – клавішу з цифрою 1 (вікно редактора має номер 1, вікно програми – номер 2). Тепер все готово до подальших експериментів з програмою.

А тепер наведемо приклад, у якому програма вже не «глуха», тобто може запрошувати дані у користувача. Нехай потрібно запросити у користувача два числа, після цього вивести на екран їх добуток:

Приклад 3.2.3.

```
program AxB;
var a,b: integer;
begin
  writeln('Введіть два числа a та b');
  readln(a,b);
  writeln('Добуток a*b дорівнює ',a*b);
  readln;
end;
```

Про те, що робить перший з операторів writeln, нам відомо: він виводить на екран рядок Введіть два числа *a та b.* У цій програмі з'являється новий рядок, який починається словом readln. При виконанні другого оператора **readln(a,b);** програма чекатиме, поки користувач не введе число (числа) з клавіатури і не натисне «Enter». Значення змінних, які вводяться, розділяються пробілом або переведенням на початок нового рядка (натисненням Enter). Після введення значень усіх змінних із списку виконання програми продовжується з наступного оператора.

Readln відрізняється від read, який ми розглядатимемо далі, лише тим, що усі змінні мають бути введені в один рядок екрану, Enter натискується один раз в кінці. Readln використовується в основному для введення рядків тексту, для введення чисел краще використовувати read, оскільки в цьому випадку користувач може вводити дані вільніше (*і в один, і в декілька рядків екрану*).

Якщо користувач вводить дані неприпустимого типу (наприклад, рядок тексту замість числа), то виводиться повідомлення про помилку і виконання програми припиняється.

Наприклад, ви хочете ввести: a=2, b=4, c=6. На екрані вводимо:

```
2_4_6¬
```

_ означає пробіл,

¬ означає натискання Enter, переходимо до наступного рядка, або

```
2¬
4¬
6¬
```

або

```
2¬
4_6¬
```

Пробілів може бути і декілька.

Третім оператором у блоці begin end виводимо на екран спочатку надпис «Добуток a*b дорівнює », а потім значення виразу a*b («*» – знак множення).

Четвертий оператор readln; використовується як затримка і чекатиме введення даних, а ви тим часом переглядатимете на екрані результати свого введення та результати роботи програми, поки не натиснете клавішу Enter.

3.2. Виконуємо програму на комп'ютері

Зараз ми детальніше пояснимо, у якому порядку, і які кнопки потрібно натискати на комп'ютері, щоб це зробити. Пояснення розраховане на тих, кому кортить скоріше сісти за комп'ютер і хто вже має невеликий досвід роботи на ньому, зокрема вміє вводити у комп'ютер два-три рядки тексту і запускати, наприклад, потрібну гру.

Отже, послідовність дій для «досвідчених» по виконанню першої програми на комп'ютері:

1. **Запустіть Pascal**. (файл turbo.exe). Нагорі екрану виникне меню, а під ним синє (зазвичай) вікно на весь екран з миготливим курсором. Можна вводити програму. Якщо вікно не з'явилося, то натисніть клавішу F10, а потім в меню оберіть слово File та New. (Надалі скорочено просто напишемо **File → New**).
2. **Введіть в це вікно програму**, як звичайний текст у звичайному текстовому редакторі.
3. Якщо вийде, **збережіть програму** на жорсткому диску. Для цього **File → Save**, а потім в діалоговому вікні, що відкрилося, виберіть каталог і введіть ім'я файлу, під яким збережете програму.
4. **Виконайте програму**. Для цього виконайте **Run → Run** (або Ctrl+F9), а щоб побачити результати, натисніть Alt+F5, що означає: утримуючи натиснутою клавішу Alt, клацніть по клавіші F5. Виконавши програму вперше, поекспериментуйте – змініть вміст операторів Writeln – і виконайте програму ще раз.
5. Якщо у вашій програмі Pascal помітив **помилку**, він ставить на неї (або поряд з нею) курсор і повідомляє про неї золотими літерами на червоному фоні. Найпоширеніші для початківців повідомлення про

помилки ви знайдете кількома рядками нижче. Виправте помилку та поверніться до пункту 3.

6. Нормальна програма, виконавши усе, що потрібно, закінчує роботу. Проте, якщо ви припустили помилку, і в програмі виконується нескінченний цикл (**зациклення**), то програма не завершиться ніколи. Ви одвічно дивитиметеся на екран, по якому нескінченно біжать незрозумілі числа або слова, або малюються графічні об'єкти, а можливо і нічого не відбувається, екран порожній – все залежить від характеру помилки. Для переривання роботи програми (у тому числі й тієї, що зациклилася) існує комбінація клавіш ***Ctrl+Break***. На екран повернеться вікно редактора. Рядок програми, на якому вона була перервана, виділяється смугою білого кольору. Якщо ви знову запустите програму, вона продовжить роботу з перерваного місця. Щоб почати спочатку, приберіть смужку з екрану клавішами *Ctrl+F2*.

Розпізнаємо повідомлення комп'ютера про помилки

Отже, ви досягли того, щоб ваша програма виводила потрібний результат – Я програмую на Turbo-Pascal (або 7, якщо це приклад 3.2). Тепер давайте поекспериментуємо. Мета експерименту – навчити вас правильно реагувати на повідомлення про помилки, які видає Pascal. Оскільки потрібний результат надрукований, то у вашій програмі помилок немає. Доведеться нам навмисно вводити помилки і спостерігати за реакцією Pascal.

1. Зітріть крапку після END. Тепер запустіть програму. На екрані з'явиться повідомлення Unexpected end of file, яке перекладається Несподіваний кінець файлу. Pascal знайшов цю помилку у програмі і повідомляє нас про неї, поставивши курсор у рядок, який містить помилку. Приберіть повідомлення комп'ютера клавішею Esc.
2. Виправіть цю помилку і введіть іншу – зітріть крапку з комою після Program My_First_Program. Цього разу повідомлення таке – ";" expected, що означає – Чекав крапку з комою. Проте курсор стоїть зовсім не в тому місці, де помилка, а на початку наступного оператора. Вам доведеться звикнути до того, що Pascal не завжди точно визначає місце помилки.
3. Виправіть цю помилку і введіть іншу – напишіть саме ім'я оператора WriteLn з помилкою – WiteLn. Реакція Pascal – Unknown identifier, що означає – Невідоме ім'я. Мається на увазі ім'я процедури WriteLn.
4. Виправіть цю помилку і введіть іншу – зітріть праву лапку в операторі Text = 'Я програмую на Turbo-Pascal', щоб вийшло Text = 'Я програмую на Turbo-Pascal. Реакція Pascal – String constant exeeds line. Переклад означає натяк на те, що раз лапку відкрили, то треба її закривати.
5. Тепер зітріть ліву лапку. Реакція Pascal – Syntax error, що означає Синтаксична помилка. Pascal в утрудненні – він знає, де помилка, але у чому вона полягає – не знає.
6. Виправіть помилки і введіть іншу – зітріть праву дужку в операторові WriteLn(Text), щоб вийшло WriteLn(Text . Реакція Pascalя – ")" expected, що означає – Чекав дужку.

7. Виправіть помилку і введіть іншу – зітріть ліву дужку в операторові WriteLn(Text), щоб вийшло WriteLn Text). Реакція Pascal – ";" expected, що означає – Чекав крапку з комою, причому курсор стоїть на букві T. Ось тут Pascal не має рацію (це не означає, що він дурний, просто неможливо врахувати усі можливі причини помилки). Вам доведеться звикнути і до того, що Pascal іноді неправильно визначає характер помилки.

3.3. Структура програми

А тепер розглянемо структуру програми у загальному вигляді. Будь-яка програма на TURBO-PASCAL складається з трьох блоків: блоку оголошень (описів), блоку опису процедур і функцій та блоку основної програми. Нижче ці блоки розписані детальніше.

Блок оголошень:

program ... (назва програми)
uses ... (зовнішні модулі, які використовуються програмою)
label …(список міток (імен ділянок програм), якщо вони потрібні)
const ... (оголошення констант)
type ... (оголошення типів)
var ... (оголошення змінних)

Блок опису процедур та функцій:

procedure (**function**)
begin
...
end;
...

Блок основної програми:

begin
... (оператори основної програми) ...
end.

Слова Program, begin та end виділяють дві частини програми – **розділ описів** (***Блок оголошень, Блок опису процедур і функцій***) та **розділ операторів**. Така структура обов'язкова для будь-якої програми, що є наслідком жорсткої вимоги мови: будь-який нестандартний ідентифікатор, який використовується в операторах, що виконуються, повинен бути заздалегідь описаний в розділі описів. Стандартні ідентифікатори пов'язані із заздалегідь оголошеними об'єктами і входять в стандартну бібліотеку TURBO-PASCAL. Таким, наприклад, є ідентифікатор WriteLn. Стандартні ідентифікатори, якщо вони використовуються у програмі, описувати не потрібно.

Всі змінні, які використовуються у програмі, повинні бути перераховані в розділі опису змінних (Блоку оголошень). Цей розділ складається з рядків опису змінних. Таких рядків може бути декілька, розміщуються вони між

заголовком програми, підпрограми або модуля і зарезервованим словом begin, який відкриває розділ операторів програми, підпрограми або модуля.

Розділ описів може бути і відсутнім, якщо ж він присутній, то може містити розділи **VAR, LABEL, USES, CONST, PROCEDURE** та інші, ще нами не вивчені.

Будь-який з цих блоків (розділів) програмної одиниці або всі одночасно можуть бути порожніми, тобто не містити ніяких описів або операторів, що виконуються.

У розділі описів повинні міститися описи усіх ідентифікаторів, які використовуються у розділі операторів, що виконуються. Виключенням є ідентифікатори, які визначені в інтерфейсних частинах програмних модулів (бібліотек), а також глобальні для процедури або функції ідентифікатори. Якщо програмна одиниця використовує ідентифікатор з інтерфейсної частини, будь-якого модуля, на початку програми в пропозиції USES (використовувати, читається ““юзез”) необхідно вказати ім’я цього модуля (докладніше про це далі). Останнє не відноситься до ідентифікаторів, які визначені у стандартному модулі SYSTEM, тобто ім’я цього модуля в пропозиції USES вказувати не потрібно. Більш того, модуль SYSTEM вважається заздалегідь оголошеним, тому оголошення **Uses System;** компілятор розцінить як спробу подвійного оголошення модуля SYSTEM і дасть відповідне повідомлення про помилку. У розділі описів оголошуються ідентифікатори типів, об’єктів, констант, змінних, а також міткі, процедури та функції. Опису типів та об’єктів повинно передувати зарезервоване слово TYPE (тип, читається “тайп”), опису констант – CONST, змінних – VAR та міток – LABEL. Але про це ми поговорим пізніше.

На відміну від стандартного Pascal розділи TYPE, CONST, VAR, LABEL можуть слідувати один за одним у будь-якому порядку і зустрічатися у розділі описів скільки завгодно разів. Опис процедури або функції полягає у вказівці заголовка цієї процедури (функції) та її тіла (докладніше ми це розглянемо в розділі 5).

Ви вже, мабуть, помітили, що перші букви розділів програми uses, const, var, а також слова program, begin та end знаходяться в одній колонці. Вміст усіх розділів друкується у стовпчиках, починаючи з колонки, яка зміщена на дві (три) позиції від початку розділу. Програми, які оформлені таким чином, називають структурованими.

Тексти структурованих програм дуже зрозумілі, легко читаються, в них дуже важко припустити синтаксичну помилку, але якщо вона все ж допущена, то її легко помітити і виправити. Нарешті, вони гарно виглядають. Тому розробка структурованих програм є дуже важливим прийомом хорошого стилю програмування (культури програмування). Структурне програмування є стандартом сучасних програм. Ніколи не залишайте процес структуризації програми на кінець її розробки. Привчите себе одразу писати структуровані тексти, і ви будете з лихвою віддячені скороченням часу розробки програми та зручністю її наступної, зазвичай завжди неминучої, модернізації.

Надалі ви познайомитеся з іншими прийомами хорошого стилю програмування, вкажемо також риси поганого стилю складання програм - те, чого бажано не робити.

3.4. Процедури введення-виведення

А тепер прийшов час детальніше поговорити про ті оператори, які ми застосовували для введення-виведення інформації. Майже кожна програма повинна спілкуватися з користувачем, тобто виводити результати своєї роботи на екран та запрошувати у користувача інформацію з клавіатури. Для того щоб це стало можливим, в TURBO-PASCAL є спеціальні процедури (тобто невеликі допоміжні програми, а не оператори), які називаються процедурами **введення-виведення**.

Програма виконується по рядках в порядку їх розташування у тексті доти, доки не зустрінеться виклик деякої процедури (або функції, про це далі). Потім управління передасться рядкам цієї процедури. Після виконання процедури управління повертається тому рядку програми, який слідує одразу за викликом процедури.

Є прекрасна аналогія для роботи програми з процедурою. Наприклад, якщо під час малювання у вас ламається олівець, ви припиняєте малювати і заточуєте його. Після цього ви повертаєтеся до того місця малюнка, де зламався олівець. Коли програма потребує виконання деякої сервісної операції, викликається процедура (або функція), яка відповідає за виконання цієї операції, після чого програма продовжуе свою роботу з того місця, де була викликана процедура.

3.4.1. Формат процедур введення-виведення

Для того, щоб змусити процедуру працювати у нашій програмі, потрібно написати її ім’я, за яким в дужках, через кому перерахувати параметри, які ми хочемо їй передати. Для процедури виведення інформації на екран параметрами можуть бути числа або текстові повідомлення, які повинна виводити наша програма на екран. Опишемо призначення цих процедур.

1. **Write(p1,p2,... pn)**; – виводить на екран значення виразів p1,p2... pn, кількість яких (n) практично необмежена. Вирази можуть бути числові, строкові, символьні та логічні. Під виразом розумітимемо сукупність деяких дій, які застосовуються до змінних, констант або літералів, наприклад: арифметичні дії та математичні функції для чисел, функції для обробки рядків та окремих символів, логічні вирази тощо.
2. **Writeln(p1,p2,... pn)**; – аналогічно writln, виводить значення p1,p2... pn, після чого переводить курсор на початок нового рядка. Зміст параметрів – такий самий, зауваження про форматне виведення залишаються тими самими. Існує варіант writeln; (без параметрів), що означає лише переведення курсора на початок нового рядка.

3. **Readln(v1,v2,...vn)**; – введення з клавіатури значень змінних v1...vn. Змінні можуть бути строкового, символьного або числового типу. При введенні слід розділяти значення пробілами, символами табуляції або переведенням рядка (тобто, натискуючи Enter).
4. **Read(v1,v2,...vn);** – за призначенням схожий з readln; відмінність полягає у тому, що символ переведення рядка (Enter), що натискається при завершенні введення, не «проковтнувся», а чекає наступного оператора введення. Якщо ним виявиться оператор введення строкової змінної або просто readln, то строковою змінною буде присвоєно значення порожнього рядка, а readln без параметрів не стане чекати, поки користувач натискуватиме Enter, а зреагує на вже введений символ.

Відмітимо, що оператор **WriteLn(Text);** у прикладі 3.1 після виведення тексту здійснював переведення рядка і встановлював курсор в початок наступного рядка екрану. Саме у цій простій дії (переведення рядка) полягає єдина відмінність у виконанні процедури WriteLn від процедури Write. Так, після виконання фрагмента програми:

```
begin
    Write( ‘Я вивчаю Pascal’ );
    Write( ‘версії’);
    Write(7)
End.
```

На екрані отримаємо рядок: Я вивчаю Паськальверсии7. У цьому випадку програмістові необхідно розділяти слова пробілами після кожного оператора Write.

3.4.2. Форматне виведення

Якщо ви виводите на екран дійсні числа, то на екрані ви можете побачити такі результати:

x1= 1.5000000000E+00, x2= 2.5000000000E+00

Дійсні числа читати в подібній формі незручно, для їх виведення у «природному форматі» в операторах Write та Writeln використовується **форматне виведення**, тобто явну вказівку на те, скільки треба виділяти позицій на екрані для виведення значення.

В операторі write або writeln дійсне значення (а також ціле або строкове) часто зручніше записувати у вигляді:

змінна:ширина:точність

ширина – ціле додатнє число, яке визначає, сколькі екранних позицій відводиться для виведення всього числа; визначається для будь-яких чисел та рядків.

точність – ціле додатнє число, яке визначає, скільки цифр з ширини відводитися на виведення дробової частини числа. "Точність" візначена толькі для дійсних чисел. Не враховує позицію десяткової точки. Неприпустимі значення ширини та точності не будуть враховані при виведенні.

Наприклад, для дійсних типів: write(r+s:9:4); – вивести значення виразу r+s з виділенням для цього 9 позицій, з них 4 – після коми. Для інших типів все дещо простіше: write(p:10); – вивести значення виразу p, виділивши під це 10 позицій. Виведення на екран у будь-якому випадку здійснюється по правому краю виділеного поля. Пізніше ми ще повернемося до формату виведення.

3.4.3. Виведення результатів на принтер

Інколи потрібно, щоб програма вивела результати своєї роботи на принтер. Для цього досить виконання двох умов. Першим оператором розділу описів програми слід вказати оператор

uses printer;

який підключає стандартну бібліотеку для роботи з принтером, а першим параметром оператора write або writeln вказати символічне ім'я принтера lst, яке описане в бібліотеці printer:

write ('Hello'); рядок 'Hello' виведений на екран

write (lst,'Hello'); а тут – вже на принтер

Детальніше оператор uses ми вивчатимемо пізніше у цьому посібнику. Відмінність між write і writeln зберігається також і при виведенні на принтер – тобто, при використанні writeln позиція друку на принтері буде перенесена на початок наступнго рядка.

Тут не наводиться код, який дозволяє перевірити, чи готовий принтер до друку і чи вдалася операція виведення даних на нього. Подібні перевірки ви навчитеся робити, коли вивчатимете Pascal більш детально.

3.5. Типи даних

Прийшов час ознайомитися з різними типами даних, деякі з них ми вже застосовували у наших програмах. Спочатку ознайомимся з деякими так званими простими (базовими) типами даних, з рештою типів даних, зокрема з непростими (структурованими типами) ви познайомитеся пізніше.

Дані, які зберігаються у пам'яті комп'ютера і піддаються обробці, можна віднести до різних типів. Поняття типу даних виникає природним чином, коли необхідно застосувати до них операції обробки. Наприклад, операція множення застосовується до чисел, тобто до даних числового типу. Це відомо ще з початкової школи. А що вийде, якщо помножити слово «Петя» на число 4? Звідси напрошується висновок: деякі операції не слід застосовувати до різнотипних даних. Ми також не знаємо, що повинно вийти в результаті множення слів «Петя» і «Маша», тому можна зробити висновок, що певні операції взагалі не можуть бути застосовані до даних деяких типів. З іншого боку, існують операції, результат яких залежить від типу даних.

Наприклад, операція додавання, що позначається символом «+», може застосовуватися і до двох чисел, і до двох рядків, що складаються з довільних слів. У першому випадку результатом застосування цієї операції буде деяке

число, а в другому – рядок символів, що виводиться шляхом дописування другого рядка у кінець першого. У разі символьних рядків операцію додавання ще називають склеюванням або конкатенацією. Операції, які застосовуються до різних типів даних, які позначаються одним і тим же символом, називають також переобтяженими. Так, операція, яка позначається символом «+», є переобтяженою: стосовно чисел вона виконує арифметичне додавання цих чисел, а стосовно рядків символів – **склеювання (конкатенацію, зчеплення, дописування)**.

На даному етапі слід звернути особливу увагу на відмінності у поданні числових та символьних даних, які ви вже застосовували у своїх перших програмах у вигляді констант. Числа, як дані числового типу завжди подаються без лапок. Для їх написання ми використовуємо цифри та, при необхідності, знак числа і розділову крапку. Строкові дані оточуються в одинарні лапки «‘». Наприклад, запис -3.14 представляє число, а запис ‘-3.14’ – рядок символів. Це дані різних типів, хоча ми і можемо сказати, що їх змістом є одне і те саме число. Виняток становлять лише дані логічного типу, які можуть мати одне з двох значень: true або false. Ці значення записуються без лапок, і ми розглядатимемо їх пізніше. З точки зору мови програмування число (точніше, дані числового типу) можна коректно використовувати в арифметичних операціях, а символьні рядки (строки) – в строкових операціях.

Вимога попереднього опису ідентифікаторів здається надмірно строгою і такою, що робить мову менш вільною. Насправді в ній виявляється тенденція розвитку мов програмування у бік підвищення надійності створюваних програм. У деяких мовах, наприклад, Фортрані або Бейсіку, не потрібний попередній опис ідентифікаторів. Тому в цих мовах при написанні великої програми буває важко виявити помилково введений або пропущений символ в ідентифікаторі. Якщо, наприклад, усюди у програмі використовується змінна з ім’ям EPSILON, а в одному місці помилково написане EPSLION, то програма може благополучно відкомпілюватися і навіть давати майже правдоподібний результат для деяких наборів даних, але в якийсь момент почне поводитися дивно. Обов’язковий попередній опис ідентифікаторів у TURBO-PASCAL захищає програми від такого роду помилок і підвищує їх надійність.

Описати ідентифікатор – це означає вказати тип пов’язаного з ним об’єкту програми (константи або змінної). Поняття типу – одне з фундаментальних понять TURBO-PASCAL. У розділі 4 детально будуть розглянуті різні типи.

Щоб пояснити описані нижче особливості мови і при цьому не дуже забігати наперед, вкажемо, що тип визначає, по-перше, спосіб внутрішнього для комп’ютера подання об’єкту і, по-друге, дії, які дозволяється над ним виконувати.

У програмах, які розглядаються далі у цьому розділі, знадобляться такі типи даних:

- **INTEGER –** цілочисельні дані (читається " ‘інтеджер", перекладається “цілий”), у внутрішньому уявленні займають 2 байти; діапазон можливих значень – від -32768 до +32767; дані представляються точно;
- **REAL** – (читається "‘ріел", перекладається "дійсний") дійсні дані, займають 6 байт; діапазон можливих значень – від -2.9E-39 до 1.7E+38; точність представлення даних – 11..12 значущих десяткових цифр;
- **CHAR –** (читається – "кар", скорочення від Character – "символ") символ, займає 1 байт;
- **STRING –** (читається "стринг", перекладається "рядок") рядок символів, займає Max+1 байт, де MAX – максимальна кількість символів у рядку;
- **BOOLEAN –** логічний тип (читається "буліен", перекладається "булевий, логічний"), займає 1 байт і має два значення: FALSE (хиба, читається "фолс") та TRUE (істина, читається "тру").

Тип константи визначається, як вже наводилося вище, способом запису її значення. Наприклад:

const cl = 17; c2 = 3 .14; c3 = ‘A’; c4 = ‘3.14 ‘; c5 = False;

При аналізі цього фрагмента програми компілятор віднесе першу константу до типу INTEGER, другу – до типу REAL, третю – до CHAR, четверту – до STRING і останню – до BOOLEAN. Ознакою, що дозволяє віднести константу до REAL або до INTEGER, є наявність або відсутність десяткової крапки в її значенні або/та наявність десяткового ступеня (починається з літери E | e). Зрозуміло, константи C2 та C4 відносяться до різних типів: C2 – до REAL (у константі є десяткова крапка), а C4 – до STRING (константа оточена апострофами). Константу C3 компілятор рахуватиме CHAR, що відноситься до типу: одиночний символ у лапках відноситься до CHAR, тоді як низка символів – до STRING.

На відміну від константи **змінна** іменує об’єкт програми, який може змінювати своє значення у процесі виконання програми. Поняття змінної величини вам відомо з курсу шкільної математики. Нехай кілька років тому ваше зріст рівнявся 150 см. Позначимо цей факт так: *Rost = 150*. Тепер він дорівнює 170 см, тобто *Rost = 170*. Виходить, що величина *Rost* змінилася. Тому вона називається змінною величиною. Числа 150 і 170 називаються **значеннями** змінної величини *Rost*.

Будь-яка мова програмування вміє поводитися із змінними величинами. Без них вона була б дуже слабкою, і змогла би отримувати з комп’ютера лише можливості звичайного калькулятора. Так само алгебра без змінної величини перетворилася б на арифметику. Проте переваги застосування змінних величин нам відкриються пізніше, а тепер наше завдання – до них звикнути.

При описі змінних за ідентифікатором ставляться двокрапка та ім’я типу змінної. Декілька однотипних змінних можна об’єднувати у список, розділяючи їх комами. На початку розділу опису змінних повинно стояти зарезервоване

слово VAR (читається “вар”, – скорочення від англійського VARiables – змінні). Наприклад:

```
var     sigma :Real;   a,b,c,d :Char;
text1   :String[15];
text2  :String;
flag    :Boolean;
```

Символьний та строковий тип опишемо детальніше.

Символьний тип. Тип даних, змінні яких зберігають рівно один символ (букву, цифру, розділовий знак, тощо) називається символьним, а щодо мови Pascal – **CHAR**. Оголосити змінну такого типу можна так: `var ch1: char;`. Для того щоб присвоити цій змінній символ, потрібно використовувати оператор присвоєння, а символ записувати у лапках, наприклад: `ch1:='A';`. Для символьних змінних можливо також використання процедури readln, наприклад:

```
write('Закінчити роботу? (Y/N)');  readln(ch);
if ch='Y' then ...{закінчити}...
else ...          {продовжувати}...;
```

Символьні змінні у пам’яті комп’ютера **зберігаються як числові коди.** Інакше кажучи, у кожного символа порядковий номер, починаючи з нуля. Наприклад, код пробілу дорівнює 32, код ‘ A’ – 65, ‘ B’ – 66, ‘ C’ – 67, код символу ‘ 1’ – 49, ‘ 2’ – 50, ‘.’ – 46, тощо. Деякі символи (з кодами, меншими 32) є символами *управління,* при виведенні таких символів на екран відбувається якась певна дія. Наприклад, символ з кодом 10 переносить курсор на новий рядок, з кодом 7 – викликає звуковий сигнал, з кодом 8 – зрушує курсор на одну позицію ліворуч. Під зберігання символу виділяється 1 байт (байт складається з 8 біт, а біт може набувати значень 0 або 1), тому всього можна закодувати 2^8=256 різних символів. Система кодування символів, яка використовується у мові, називається ASCII (American Standard Code for Information Interchange – американський стандартний код для обміну інформацією).

Для того щоб отримати у програмі символ за його кодом потрібно застосовувати функцію **`Chr`**, наприклад:

```
var i: byte; {число, яке займає 1 байт, значення – від 0 до 255}
    ch: char;

readln(i); writeln(символ з кодом ‘,i,’ – це ‘,chr(i));
```

Якщо замість коду символу використовується конкретне число, а не вираз і не змінна, то можна використовувати символ «#», скажімо так: `ch:=#7;`. Для переходу від символу до коду використовується функція **`Ord`** (від слова ordinal – порядковий). Наприклад, за допомогою наступної програми можна вивести на екран таблицю з кодами символів:

```
program ASCII;
```

```
var ch: char;
begin
  for ch:=#32 to #255 do write(ord(ch),'–>',ch,'  ');
  readln;
end.
```

У цій програмі в якості лічильника циклу була використана символьна змінна. Це припускається, оскільки цикл for може використовувати як лічильник змінні будь-якого типу, значення якого зберігаються у вигляді цілих чисел.

З використанням кодів працюють ще дві функції, значення яких символьні:

1) **succ** (від succeedent – наступний), вона видає символ з наступним кодом.
2) **pred** (від predecessor – попередній), видає символ з попереднім кодом.

Якщо спробувати у програмі отримати succ(#255) або pred(#0), то виникне помилка.

Строковий тип. Строковий тип взагалі не відноситься до структурованих типів даних, але ми так часто користуємося цим типом даних, що з ними треба познайомитися заздалегідь.

Для зберігання рядків (тобто послідовностей із символів) TURBO-PASCAL має тип **STRING**. Значеннями строкових змінних можуть бути послідовності різної довжини (від нуля і більше, довжині 0 відповідає порожній рядок). Оголосити строкову змінну можна двома способами: або var str: string; (максимальна довжина рядка – 255 символів), або var str: string[n]; (максимальна довжина – n символів, n – константа або конкретне число).

У наведеному вище прикладі змінна text1 описана з вказівкою її максимальної довжини (15 символів), а в описі змінної text2 максимальна довжина не вказана і компілятор встановить для неї гранично припустиму в TURBO-PASCAL довжину – 255 символів.

Для того щоб присвоїти значення строковій змінній, використовуються ті самі прийоми, що і при роботі з символами. У разі присвоєння значення для конкретного рядка, цей рядок повинен записуватися у лапках (ми вже використовували цей прийом – Text := 'Я вивчаю Pascal').

Розглянемо ще одну нескладну програму (приклад 3.3). Її призначення – ввести з клавіатури два цілі числа, знайти результат ділення першого числа на друге і вивести отриманий результат на екран.

Приклад 3.3.

```
{Програма вводить два цілі числа і виводить результат ділення 1-го на 2-е число}
Program Input_0utput;
var   n1,n2 : Integer;   {n1 і n2 –  цілі, що вводяться}
      x : Real;{x –  результат}
BEGIN
      Write( 'n1 = ');    {Повідомляємо про введення n1}
```

```
ReadLn (n1);            {Вводимо n1}
Write ( 'n2 = ');   {Повідомляємо про введення n2}
ReadLn (n2);            {Вводимо n2}
x := n1/n2;         {Знаходимо результат, / – позначає ділення}
WriteLn('n1/n2 =',x);   {Виводимо його}
ReadLn
END.
```

Важливою частиною вихідного тексту програми є коментарі. І в нашій програмі, перш за все, впадає у вічі поява у програмі пояснюючих коментарів. Коментар в TURBO-PASCAL – це довільна послідовність будь-яких символів, яка оточена фігурними дужками. Коментар дозволяється вставляти у будь-яке місце програми. Як обмежувачі коментаря припускається використання фігурних дужок «{» і «}», а також пари символів: «(*» – зліва від коментаря та «*)» – праворуч від нього:

{ Це – коментар }

(Це – теж коментар *)*

Редактор TURBO-PASCAL виділяє коментарі нахиленим шрифтам (курсивом).

Коментарі з однотипними обмежувачами не можна вкладати один в одного, тобто неприпустимі послідовності, що мають вигляд

{...{...}...} або (*...(*...*)...*)

Проте можна вкладати коментарі з обмежувачами різних типів (не більш за одну глибину вкладення):

{ ... (* ...*)...} або (* ... { ... } ... *)

Коментарі дозволяють включити докладний опис програми та пояснення до неї прямо у вихідний текст. Грамотне і доречне застосування коментарів спрощує розуміння програми, полегшує життя її автору і програмістам, що працюють з уже готовим текстом. У фігурних дужок є і нестандартне застосування – під час налагодження часто виникає необхідність тимчасово прибрати з програми якісь оператори, зберігши, проте, їх запис. Найпростіший спосіб – укласти відповідний фрагмент програми у фігурні дужки.

Декілька слів про введення даних. Пари операторів **Write (..); ReadLn(..);** працюють, як ми вже знаємо, таким чином. Спочатку оператор Write виводить рядок на екран і залишає курсор в кінці тільки що виведеного рядка тексту.

Оператором ReadLn викликається вбудована процедура введення даних, і програма зупиняється в очікуванні введення. У цей момент необхідно набрати на клавіатурі потрібне число і натиснути клавішу Enter. Одразу після цього програма продовжить роботу: проаналізує введене число і перейде до введення наступного числа або обчислення результату. Таким чином, сигналом закінчення підготовки чергового числа є натиснення на клавішу Enter, до цього

моменту можна стирати будь-який помилково введений символ клавішею Backspace.

Для обчислення відношення введених чисел використовується один з основних операторів TURBO-PASCAL – оператор присвоєння. У його лівій частині вказується ім'я змінної, права частина є виразом того ж типу, що і змінна. Пара символів ":=", яка зв'язує ліву та праву частини оператора присвоєння, означає «**присвоїти значення**». Запам'ятаємо: в операторах присвоєння TURBO-PASCAL завжди використовуються символи ":=", тоді як при описі констант – одиночний символ "=". З погляду синтаксису мови, два символи «:=» розглядаються як один спеціальний символ і обов'язково пишуться разом. Позначка «/» в операторі x := n1/n2 означає ділення.

Оператор присвоєння використовується практично в усіх мовах програмування. У деяких мовах, наприклад, у Фортрані або Бейсіку, символом присвоєння є знак рівності, проте новачка, який звик до строгості математичних формул, може спантеличити типова форма запису фортран-оператора присвоєння, наприклад, така:

X = X + 1

Варіант запису цього ж оператора на TURBO-PASCAL:

X := X + 1;

у цьому сенсі здається логічнішим. Зрозуміло, що навряд чи кому-небудь прийде в голову бачити рівняння там, де їх немає і не може бути. Звичайно ж, і в тому, і в іншому випадку реалізується одна і та сама алгоритмічна дія: до вмісту X додається 1 і отриманий результат знов присвоюється змінній X. Зверніть увагу на оператор виведення результатів

WriteLn('n1/n2 = ',x);

У ньому як один з параметрів явно вказується константа типу рядок символів 'nl/n2 = '. Звичайно ж, константи (на відміну від змінних) зовсім не обов'язково описувати в розділі описів, оскільки їх тип легко визначається компілятором за формою запису константи. Наведені константи називають літералами.

Ще раз нагадаємо, що оператор **ReadLn**; (без аргументу – очікування введення) **застосовується у кінці програми для затримки перегляду результатів програми при виведенні на екран**, щоб одразу побачити результат роботи програми, не використовуючи Alt+F5. Цей спосіб (*Alt+F5*) поганий при роботі у графічному режимі: зображення або спотворено, або взагалі не є видимим. Продовжити роботу програми можна після натиснення клавіші Esc (або Enter).

У таблиці 3.1 наведені далеко не всі типи даних Free Pascal. Більш докладно про типи даних Free Pascal можна прочитати в довідковому матеріалі середовища розробки (довідка викликається клавішею F1). Для тих, хто працює з Turbo Pascal, зауважу, що не всі з наведених у таблиці 3.1 типів даних підтримуються компілятором Turbo Pascal.

Таблиця 3.1. Основні типи даних Free Pascal

Тип	Размір, байт	Діапазон (множина) значень
Цілочисельні типи		
Byte	1	0…255
Shortint	1	-128…127
Integer	2	-35768…32767
Word	2	0…65535
Longint	4	-2147483648…2147483647
Cardinal	4	0…4294967295
Дійсні типи		
Single	4	$1{,}5*10^{-45}$…$3{,}4*10^{38}$
Real	8	$5*10^{-324}$…$1{,}7*10^{308}$
Extended	10	$1{,}9*10^{-4951}$…$1{,}1*10^{4932}$
Логічний тип		
Boolean	1	TRUE, FALSE
Символьний тип		
Char	1	Всі символи таблиці ASCII
Строкові типи		
String	0…255	Масив ASCII-символів, разміром від 0 до 255
AnsiString		Масив ANSI-символів, размір не обмежений

3.6. Порядок складання простої програми

Навіть створення простої програми для розв’язання будь-якого завдання вимагає певного порядку, прийомів та методів, які прийнято називати **технологією програмування**. Детальніше цю тему ми розглядатимемо у розділі 12. Розв’язання завдань за допомогою комп’ютера включає такі основні етапи, частина з яких здійснюється без участі комп’ютера.

1. Постановка завдання:
   - збір інформації про завдання;
   - формулювання умови завдання;
   - визначення кінцевої мети розв’язання задачі;
   - визначення форми видачі результатів;
   - опис даних (їх типів, діапазонів величин, структури тощо).
2. Аналіз та дослідження завдання, моделі:
   - аналіз існуючих аналогів;
   - аналіз технічних та програмних засобів;
   - розробка математичної моделі;
   - розробка структур даних.
3. Розробка алгоритму:
   - вибір методу проектування алгоритму;
   - вибір форми запису алгоритму (блок-схеми, псевдокод тощо);

- вибір методу тестування та тестів;
- проектування алгоритму.

4. Програмування:
   - вибір мови програмування;
   - уточнення способів організації даних;
   - запис алгоритму на обраній мові програмування.
5. Тестування та відлагодження:
   - синтаксичне відлагодження;
   - відладка семантики та логічної структури;
   - тестові розрахунки та аналіз результатів тестування;
   - вдосконалення програми.
6. Аналіз результатів розв'язання задачі та уточнення у разі потреби математичної моделі з повторним виконанням етапів 2-5.
7. Супровід програми:
   - доопрацювання програми для розв'язання конкретних завдань;
   - складання документації до розв'язанного завдання, до математичної моделі, до алгоритму, до програми, до набору тестів, до використання.

Наведемо приклад простої програми з використанням константи (π = 3.1415926), змінної, операторів Write(..);, ReadLn(..); та вбудованої функції SQR(X) – піднесення у квадрат X (тобто X^2). Нам відомі: число π, радіус кола (запитується у користувача). Необхідно обчислити довжину кола та площу кола.

1 етап. Сформулюємо умову завдання. Нам відомі: число π, радіус кола (запитується у користувача). Необхідно обчислити довжину кола та площу кола. Результат вивести на екран. Вихідні дані мають тип real.

2 етап. Програміст сам повинен знати розв'язання задачі. Адже програма – це інструкція по її розв'язанню. Не можна давати інструкцію, не знаючи, як розв'язувати. У нашому випадку програміст повинен знати формули для обчислення довжини кола та площі кола. Це завдання досить типове. Математичною моделлю цього завдання є знаходження довжини кола та площі кола за відомими формулами: $L=2\pi R$, $S=\pi R^2$.

3 етап. Потрібно надати імена змінним. Ім'я змінної (або константи) повинне говорити про її зміст. Якщо змістом є довжина кола, то не лінуйтеся і не називайте її L, тому що через місяць, розбираючись у своїй напівзабутій програмі, ви довго тертимете лоба і згадувати: «Що я позначив через L»? Назвіть її dlina_okr (якщо ви не знаєте англійської, або, наприклад, length, якщо знаєте). Для площі – Plochad (area). Так роблять усі професійні програмісти. Задовольнимося такими іменами:

PI	– число π
Radius	– радіус кола
Dlina_okr	– довжина кола
Plochad	– площа круга

Потрібно визначити, якого типу будуть змінні. Оскільки число π і радіус – дійсні числа, то і довжина, і площа будуть також дійсними. Перші два рядки програми будуть такими:

```
var     PI = 3.1415926;
        Radius, Plochad_Okr, Dlina_Okr: real;
```

Перед обчисленнями потрібно задати початкові дані розв'язку задачі. Наступні рядки програми:

```
begin
        WriteLn('Який радіус кола?');
        Read(Radius);
```

Тепер потрібно задати комп'ютеру дії, які потрібно виконати з початковими даними, щоб отримати результат.

```
        Dlina_Okr := 2*PI*Radius;
        Plochad := PI*SQR(Radius);
```

Після отримання результату його необхідно надрукувати. Дійсно, всі оператори присвоєння комп'ютер виконує "в думці". Після їх виконання в елементах пам'яті Dlina_Okr і Plochad знаходитимуться числові результати розв'язку задачі. Щоб їх дізнатися, користувачу даної програми необхідно використовувати оператор WriteLn, після чого програму можна закінчувати:

```
        WriteLn('Довжина кола = ', Dlina_Okr);
        WriteLn('Площа кола = ', Plochad);
        ReadLn    {Для затримки результатів програми та перегляду їх на екрані}
end.
```

4 етап. Складаємо програму на мові Pascal.

Наша програма цілком буде виглядати так:

Приклад 3.4. Програма обчислення площі кола та довжини кола.

```
Program Circle;
const
        PI = 3.1415926;
var
        Radius, Plochad_Okr, Dlina_Okr: real;
begin
        WriteLn('Який радіус кола?');
        Read(Radius);
        Dlina_Okr := 2*PI*Radius;
        Plochad_Okr := PI*SQR(Radius); : 8 : 5
        WriteLn('Довжина кола = ', Dlina_Okr);
        WriteLn('Площа кола = ', Plochad_Okr); : 8 : 5
        ReadLn    {Для затримки результатів виконання програми на екрані}
end.
```

Позначка «*» у виразах 2*PI*Radius і PI*SQR(Radius) позначає операцію множення.

5 етап. Після реалізації програми на комп'ютері були введені такі тестові вхідні дані та отримані відповідні результати:

Який радіус кола? 8
Довжина кола = 50,26548
Площа кола = 201,06193

Це підтверджує правильність сствореної програми.

6 етап. Проаналізуйте результати розв'язку задачі та уточніть, у разі потреби, математичну модель або типи даних з повторним виконанням етапів 2-5.

7 етап. Супровід програми. Якщо необхідно, то доопрацюйте програму для розв'язку конкретних завдань. Складіть документацію до розв'язанного завдання, до математичної моделі, до алгоритму, до програми, до набору тестів, до використання.

3.7. Інтерфейс користувача

Коли користувач запускає програму, що наведена у прикладі 3.3, і вона робить паузу на операторі ReadLn(n1), користувач бачить перед собою n1= або порожній екран монітора (якби не було оператора WriteLn('n1='')), на якому немає ніяких натяків на запрошення вводити інформацію. Стороння людина ні за що і не здогадається, що комп'ютер чогось чекає. Це не дуже зручно. Було б набагато краще, якби на екрані все-таки можна було у потрібний момент бачити відповідне запрошення, щоб користувач чого-небудь не переплутав.

Це ж стосується і видачі результатів. На порожньому екрані з'являється, наприклад, число 740. Стороння людина ні за що не зрозуміє, який воно має сенс: чи то це 740 гривень, чи так зашифровано назва єврейського танцю "сім сорок". Говорять, що у нашій програмі незручний інтерфейс користувача, тобто людині, що використовує нашу програму, незручно з нею спілкуватися. Слово "інтерфейс" можна перекласти, як "взаємодія", у даному випадку взаємодія людини з комп'ютером.

Якщо доповнити нашу програму зрозумілими запрошеннями для введення (виведення) даних, щоб інтерфейс став зручнішим на кшталт:

```
WriteLn ( 'Введіть 1-е число для отримання результату від ділення:' );
WriteLn ( 'Введіть 2-е число (дільник):' );
WriteLn ( 'Результат від ділення ', n1, '/',n2,'=',x );
```

то ця програма працюватиме так само, як і попередня, з тією відмінністю, що під час паузи, що викликана оператором ReadLn, на екрані горітимуть зручні пояснюючі написи.

Оператор ReadLn без дужок у кінці програми потрібний для нейтралізації однієї неприємної особливості в роботі Pascal. Річ у тім, що виконавши програму, Pascal поспішає погасити екран з результатами розв'язку задачі і робить це так швидко, що людина просто не встигає ці результати розгледіти. Оператор ReadLn, поставлений після оператора WriteLn, що виводить результати на екран, задає паузу. Під час цієї паузи екран не гасне, оскільки

програма ще не виконалася до кінця, і людина може спокійно переглянути результати. Після цього оператор натисненням клавіші Enter дозволить комп'ютеру продовжити виконання програми.

Часто можна обійтися і без ReadLn. Натиснувши пару клавіш на клавіатурі (Alt+F5), ми можемо знову відтворити згаслий екран з результатами.

3.8. Перетворення типів даних та дії над ними

Як уже відзначалось, тип змінної дозволяє не тільки встановлювати довжину її внутрішнього подання, але і контролювати ті дії, які виконуються над нею у програмі. Контроль за використанням змінних ще на етапі компіляції програми – важлива перевага TURBO-PASCAL перед іншими мовами програмування, у яких припускається автоматичне перетворення типів. У TURBO-PASCAL майже неможливі неявні (автоматичні) перетворення типів. Виключення зроблене тільки стосовно констант і змінних типу INTEGER (цілі), яким дозволяється використовувати у виразах типу REAL (дійсні). Якщо, наприклад, змінні X і Y описані таким чином:

```
var
x: Integer;
y: Real;
```

то оператор `y := x + 2;`

буде синтаксично правильним; хоча праворуч від знаку присвоєння знаходиться цілочисельний вираз, а зліва – дійсна змінна, компілятор зробить необхідні перетворення автоматично. У той же час оператор

```
x := 2.0;
```

буде невірним, оскільки автоматичне перетворення типу REAL (константа 2.0 містить десяткову крапку і, отже, належить до типу REAL) в тип INTEGER в TURBO-PASCAL **заборонено**.

При присвоєнні значення змінній *x* ми записали дійсне число, розділяючи цілу і дробову частини крапкою. Майже в усіх мовах програмування і, зазвичай, у Pascal, в десяткових дробах прийнято замість коми ставити крапку. Приклад: 62.5 – шістдесят дві цілих п'ять десятих.

Зрозуміло, заборона на автоматичне перетворення типів ще не означає, що в TURBO-PASCAL немає засобів перетворення даних. Вони є, але їх потрібно використовувати явно (докладніше про це див. розділ 4). Для перетворення даних у мові існують **вбудовані математичні функції**, які набувають як параметр значення одного типу, а повертають результат у вигляді значення іншого типу. Зокрема, для перетворення REAL в INTEGER є навіть дві вбудовані функції такого роду: **ROUND** округлює REAL до найближчого цілого, а **TRUNC** усікає REAL шляхом відкидання дробової частини числа *x* (таб. 3.2). Наприклад, помилковим буде оператор `x := y/x;` але правильним `x := round(y/x);` (оголошення змінних див. вище).

Таблиця 3.2. Функції перетворення типів

Назва	*Значення (тип поверненого числа)*
frac(x)	дробова частина x (дійсне)
int(x)	ціла частина x (дійсне)
round(x)	x, округлене до найближчого цілого (ціле)
trunc(x)	число, отримане з x відкиданням дробової частини (ціле)

Функції для дійсних чисел після обробки повертають також дійсне число. Функція **FRAC(x)** повертає дробову частину дійсного числа *x*, а функція **INT(x)** повертає цілу частина дійсного числа *x*.

Розглянемо такий приклад. Дано дійсне число A, що містить два знаки до коми і два після. Отримати нове число, помінявши в числі A цілу і дробову частини. Це завдання не належить до **цілочисельної арифметики**, адже дано дійсне, а не ціле число. Спробуємо знайти цілу і дробову частини. А потім просто зберемо нове число збільшивши дробову частину в 100 разів і зменшивши цілу частина теж в 100 разів.

```
Var a,b,x1,x2:real;
Begin
  Write('введіть число');
  Readln(a);
  X1 := int(a);            {ціла частина}
  X2 := frac(a);           {дробова частина}
  b := x1/100+x2*100;      {нове число}
  Writeln(b);
  Readln
End.
```

Поняття функції у TURBO-PASCAL близьке до поняття процедури. Як і процедура, функція викликається за своїм ім'ям і може містити довільне число операторів TURBO-PASCAL і навіть внутрішніх процедур і функцій. Істотною відмінністю функції від процедури є та обставина, що функція має власне значення і, отже, може використовуватися нарівні із змінними у виразах відповідного типу.

Для перетворення даних типу CHAR (символ) у ціле число призначена функція ORD, зворотне перетворення INTEGER в CHAR здійснює функція CHR.

За допомогою наступної нескладної програми (приклад 3.5) Ви зможете дізнатись про внутрішній код довільного символу.

Приклад 3.6.

```
Program Code_pf_Char;
{Програма читає символ з клавіатури і виводить на екран цей символ  і відповідний йому внутрішній код}
var
    ch: Char;              {У цю змінну читається символ}
begin
    Write('Введіть будь-який символ: ');
    ReadLn(ch);            {Читаємо один символ}
    WriteLn(ch,' = ',ord(ch));   {Перетворимо до цілого}
```

END.

Зверніть увагу на виклик

WriteLn(ch,' = ',ord(ch));

третім параметром звернення вказаний виклик функції ORD(CH), що з погляду мови є виразом, і як ми побачимо далі, у багатьох випадках у разі виклику процедур і функцій параметрами виклику можна вказувати не тільки змінні або константи, але і вирази з їх участю.

По мірі потреби ми будемо знайомитися з іншими функціями перетворення типів даних, а зараз – про ті операції, які дозволені над різними типами. В TURBO-PASCAL є всі чотири арифметичні операції (та ще дві) над змінними REAL і INTEGER:

+ – додавання;
- – віднімання;
* – множення;
/ – ділення дійсне;
div – ділення цілочисельне;
mod – залишок від цілочисельного ділення.

Наявність двох операцій ділення є ще один прояв основоположного принципу TURBO-PASCAL: програміст повинен явно підтверджувати компілятору, що він готовий до можливих наслідків перетворення типів. Якщо, наприклад, у мові Фортран використовується вираз 1/2, то результат цього виразу залежитиме від того, змінній якого типу він буде присвоєний: якщо N є змінна цілого типу, а X- дійсного, то у програмі на Фортрані присвоєння

N = 1/2
X = 1/2

дадуть значення 0 для N і 0.5 для X. У TURBO-PASCAL такої двозначності немає: вираз 1/2 завжди має значення 0.5 і тому оператор

```
var
    N :Integer;
begin
    N := 1/2;
```

просто неприпустимий. У той же час припустимий у TURBO-PASCAL оператор

```
var
    X : Real;
begin
    X := 1 div 2;
```

самим фактом використання операції цілочисельного ділення DIV свідчить про те, що програміст свідомо відкидає дробову частину результату.

Для даних типу INTEGER у TURBO-PASCAL є ще одна операція MOD – отримання залишку від цілочисельного ділення. Наприклад:

5 mod 2 = 1, 31 mod 16 = 15, 18 mod 3 = 0.

Наведемо фрагмент програми, що обчислює суму цифр тризначного числа. Програма ілюструє вживання функцій DIV і MOD.

```
var     i,first,second,third,sum:  integer;
begin
        write('Введите целое трехзначное число:   ');  readln(i);
        first := i div 100;             { виділення першої цифри числа }
        second := i div 10 mod 10;      { виділення другої цифри числа }
        third := i mod 10;              { виділення третьої цифри числа}
        sum := first+second+third;
        writeln('Сумма цифр числа ' ,i,' = ' ,sum);
end.
```

За допомогою арифметичних операцій можна складати різні вирази. Такі вирази називаються **арифметичними**. У майбутньому ми побачимо, що вирази можуть бути не тільки арифметичними. У правій частині оператора присвоєння і в операторі WriteLn ми записували вирази, які мають чисельне значення (наприклад, *a+b-12*). На уроках математики ми звикали писати *ab+cd* , маючи на увазі: *a* помножити на *b* плюс помножити на *d*. У Pascal цей вираз ми зобов'язані писати так: *a*b+c*d*. Інакше комп'ютер подумає, що потрібне до змінної, що має ім'я *ab*, додати змінну, що має ім'я *cd*. Щоб уникнути двозначності знак множення належить писати завжди, навіть якщо це дужки. Наприклад *a * (b+c)*.

З огляду на те, що з клавіатури всю інформацію доводиться вводити символ за символом в один рядок, введення двоповерхових виразів, таких як

$$\frac{a+1}{b+1}$$

неможливе. Тому для позначення ділення і вибрана коса риска. Цей вираз на Pascal належить записувати так: *((a+1)/(b+1))*. Якби ми не поставили дужок, то вираз вийшов би таким *a+1/b+1*, а це неправильно, оскільки комп'ютер, як і ми, завжди перед додаванням і відніманням виконує множення і ділення, тому в останньому випадку він би спочатку розділив *1* на *b*, а потім до результату додав *a* і *1*.

У TURBO-PASCAL відсутня операція піднесення до ступеня, що, очевидно, викликатиме певні незручності при реалізації обчислювальних алгоритмів. Деякою втіхою може стати наявність вбудованої функції SQR, що повертає квадрат від значення параметра, причому тип результату визначається типом параметра.

При роздруківці цілочисельних значень можна задати мінімальний розмір поля для кожної величини. Наприклад, числу 999 потрібно три позиції. Якщо вказаний мінімальний розмір поля більший, то зліва від значення вставляються пропуски. Якщо ж мінімальний розмір поля менше потрібного або мінімальний розмір не вказаний, значення друкується без будь-яких додаткових пропусків.

В операторі WriteLn після змінної або виразу (вираз береться у дужки) число за двокрапкою вказує бажаний мінімальний розмір поля. У програмі з

прикладу 3.6 визначені мінімальні розміри полів для всіх цілочисельних значень, що виводяться.

Приклад 3.6. Знаходження середнього трьох чисел

```
Program AVERAGE ;
Var   First, Second, Third, Sum: integer;
begin
      First    := 5;
      Second := 17;
      Third    := 8;
      Sum     := First + Second + Third;
      WriteLn('Середнє значення', First:4, Second:4, Third:4 ' дорівнює ', (Sum div 3):3)
End.
```

Тут в операторі WriteLn('Середнє значення', First:4, Second:4, Third:4 ' дорівнює ', (Sum div 3):3) ми використовували специфікацію розмірів (:4 :3) поля для виведення значень. У загальному випадку специфікація розмірів поля виведення записується як **:w:d**, де:

w – загальний розмір поля виведення;

d – кількість позицій праворуч від десяткової крапки.

Якщо d менше, ніж кількість цифр після крапки, то Pascal автоматично зробить округлення до потрібної кількості розрядів.

Якщо замість вказівки ширини поля для виведення числа (w) поставити 0, тоді компілятор відведе стільки позицій під цілу частину, скільки відповідно вийшло, а для дробової частини залишить *d* позицій (якщо вказане d).

Наприклад, нехай змінній Rez присвоєно значення: Rez := 17.693, тоді оператори WriteLn надрукують наступні результати, які наведені у коментарях (щоб були видимі пропуски, ми їх позначили символом □):

```
WriteLn(Rez:7:3); {□17.693}
WriteLn(Rez:7:2); {□□17.69}
WriteLn(Rez:7:1); {□□□17.7}
WriteLn(Rez:5:1); {□17.7}
WriteLn(Rez:10);  {□1.8E+0001}
WriteLn(Rez:0:3); {17.693}
WriteLn(Rez);             {□1.76929999999993E+0001}
```

Є можливість роздрукувати рядок з одних пропусків. Це може бути потрібно для розділення даних. Це досягається виконанням простого оператора:

```
WriteLn
```

При роботі з цілими числами можуть виявитися корисними дві процедури (тут і далі у квадратних дужках вказуються необов'язкові параметри):

DEC(X [, N]) – зменшує вміст змінної X на значення виразу N (якщо N не задане, то на 1); тип змінної X і вирази N – INTEGER (точніше, будь-який цілий тип, див. розділ 4);

INC(X [, N]) – збільшує значення X на N (якщо N не задане, то на 1).

Над символами і рядками символів визначена єдина операція – ***зчеплення*** (***конкатенація***) двох рядків. Операція позначається символом «+». Наприклад, програма

```
var
      st: String;
begin
      st := 'Turbo'+'-'+'паскаль';
      WriteLn(st);
end.
```

надрукує рядок Turbo-Pascal.

Решта всіх дій над рядками і символами реалізується за допомогою вбудованих процедур і функцій (див. гл. 4).

І, нарешті, про операції відношення і логічні операції. Над даними типу REAL, INTEGER, CHAR, STRING визначені такі операції відношення (порівняння):

= - дорівнює;
<> - не дорівнює;
< - менше;
> - більше;
<= - менше або дорівнює
>= - більше або дорівнює.

В операціях порівняння повинні брати участь однотипні операнди. Виключення зроблене знову-таки стосовно **REAL** та **INTEGER**, які можуть порівнюватися один з одним. Результат застосування операції відношення до будь-яких операндів має тип BOOLEAN.

Порівняння двох рядків здійснюється таким чином. Символи рядків порівнюються попарно один з одним так, що перший символ першого рядка порівнюється з першим символом другого рядка, другий символ першого рядка – з другим символом другої і так далі Символи порівнюються шляхом порівняння їхніх кодів у внутрішньому уявленні (див. гл. 4). Якщо один рядок коротший за інший, символи, яких не вистачає, замінюються нулем. Відношення першої неспівпадаючої один з одним пари символів і береться за відношення двох рядків.

При порівнянні даних типу BOOLEAN враховується внутрішня угода TURBO-PASCAL, відповідно до якої FALSE є нульовий байт, а TRUE – байт з одиницею в молодшому розряді. Відмітимо, що функція ORD перетворить до цілого не тільки символи, але і логічні величини, тому

ord(false)= 0, ord(true) = 1.

У TURBO-PASCAL визначені такі логічні операції:

not – логічне НІ; **or –** логічне (додавання) АБО;

and – логічне (множення) ТА; **xor –** виключне АБО (додавання за модулем 2).

Логічні операції, які застосовні до операндів цілого та логічного типів. Якщо операнди – цілі числа, то результат логічної операції є також ціле число. Логічні операції над логічними даними дають результат логічного типу.

При обчисленні виразів будь-якого типу пріоритет обчислень визначається розставленими дужками, а при їх відсутності – за табл. 3.3 (в порядку зменшення пріоритету).

Таблиця 3.3. Пріоритет операцій

Приорітет	Операція
1	not, @
2	*, /, div, mod, and, shl , shr
3	+, – , or, xor
4	=, <>, >, >=, <, <=, in

Примітка. Операції @ (отримання адреси), shl (зсув ліворуч), shr (зсув праворуч) та in (приналежність до множини) будуть описані пізніше**.**

Слід урахувати, що на відміну від багатьох інших мов програмування у TURBO-PASCAL логічні операції мають вищий пріоритет, ніж операції відношення. У зв'язку з цим, у складних логічних виразах зазвичай необхідно розставляти дужки. Якщо, наприклад, b та c мають тип INTEGER, то вираз

a = b and c < d

викличе повідомлення про синтаксичну помилку, оскільки спочатку виконається операція b and c. Правильним буде вираз:

(a = b) and (c < d)

3.9. Оператори мови Pascal

З одним з операторів мови, який найчастіше використовується – оператором присвоєння ми вже познайомилися. Але при виконанні алгоритмів доводиться не лише знаходити значения величин, але й аналізувати їх властивості, порівнювати їх один з одним, і в залежності від результату порівняння обирати ту чи іншу гілку алгоритму. Алгоритми, які мають декілька гілок (дій), називаються ***нелінійними***. До таких відносяться розгалуджуючі та циклічні алгоритми. Для їх запису застосовуються составні (складені) оператори. Йдеться про оператори "розгалуження" та "повтору". Будь-який з них містить умову, залежно від якої виконуються або не виконуються оператори з послідовності, що входять до її складу.

Перш ніж перейти до опису операторов, зробимо кілька зауважень з приводу формату запису тексту програми, оскільки це питання не тільки естетичне, але і питання ефективності праці програміста.

При наборі тексту програми її оператори слід розташовувати таким чином, щоб зрозумілою була логіка роботи програми. Для цього використовуються пробіли між операторами або елементами операторів, а також відступи.

Кожен програміст має свій стиль використання пробілів та відступів. Пробіли та відступи допомагають читачеві програми визначити рівень підпорядкованості кожного рядка. Комп'ютерні програми містять велику кількість інформації в концентрованому вигляді, і гарне форматування може виявитися дуже корисним.

Треба також мати на увазі, що навряд чи знову написана «солідна» програма одразу буде працювати правильно. На налагодження програми може піти більше половини часу її розробки. Тому належна увага до хорошого форматування, ретельному вибору імен змінних і інших об'єктів програми, розумне документування програм може значно збільшити продуктивність праці.

Не слід також розміщувати в одному рядку більше одного-двох операторів. При записі програми на Паскалі допускаються порожні рядки, які можна використовувати для виділення логічно пов'язаних груп операторів.

3.9.1. Складений оператор та порожній оператор

Складений (“составний”) оператор – це послідовність довільних операторів програми, яка оточена операторними дужками – зарезервовані слова begin . . . end. Складені оператори – важливий інструмент TURBO–PASCAL, що дає можливість писати програми за сучасною технологією структурного програмування (без операторів переходу GOTO).

Мова не накладає ніяких обмежень на характер операторів, що входять у складений оператор. Серед них можуть бути і інші складені оператори – припускає довільну глибину їх вкладеності:

```
begin
   .......
   begin
        .......
   end;
   ......
end;
```

Фактично, весь розділ операторів, який оточений словами begin . . . end, є один складений оператор. Оскільки зарезервоване слово end є закриваючою операторною дужкою, то воно одночасно вказує і кінець попереднього оператора, тому ставити перед ним символ «;» необов’язково, і далі в усіх прикладах ми не будемо цього робити. Наявність крапки з комою перед end у попередніх прикладах означала, що між останнім оператором та операторною дужкою end розташовується порожній оператор. Порожній оператор не містить ніяких дій, просто у програму додається зайва крапка з комою. В основному

порожній оператор використовується для передачі управління у кінець складеного оператора.

3.9.2. Умовний оператор

Іноді потрібно, щоб частина програми виконувалася не завжди, а лише при виконанні деякої умови, а при невиконанні цієї умови виконувалася інша частина програми.

Умовний оператор дозволяє перевірити деяку умову і залежно від результатів перевірки виконувати ту чи іншу дію. Таким чином, умовний оператор – це засіб розгалуження обчислювального процесу. Структура умовного оператора має такий вигляд:

IF <умова> THEN <оператор1> [ELSE <оператор2>]

де IF (читається "іф", перекладається "якщо"), THEN (читається "зен", перекладається "то"), ELSE (читається "елз", перекладається "інакше") – зарезервовані слова (якщо, то, інакше); <умова> – довільний вираз логічного типу; <оператор1>, <оператор2> – будь-які оператори мови.

Під оператором розуміється один оператор (наприклад, присвоєння), або так званий **складений оператор**, який складається з кількох простих операторів, що поміщені між словами begin та end;. Важливо відмітити, що **перед else не ставиться крапка з комою**.

Умовний оператор працює за наступним алгоритмом (схемою). Спочатку обчислюється умовний вираз <умова>. Якщо результат є TRUE (істина), то виконується <оператор1>, а <оператор2> пропускається; якщо результат є FALSE (хиба), навпаки, <оператор1> пропускається, а виконується <оператор2>. Наприклад:

```
var
      x, y, max: Integer;
begin
.......
      if x > max then
          y := max
      else  y := x;
```

При виконанні цього фрагмента змінна Y набуває значення змінної X, якщо тільки це значення не перевищує MAX, інакше Y стане рівним MAX.

Фрагмент програми з оператором if (якщо, то, інакше) можна записати і в іншому “благороднішому” вигляді структурного програмування:

```
      if x > max then
         y := max
      else
        y := x;
```

Частина ELSE <оператор2> умовного оператора може бути опущена. Тоді при значенні TRUE умовного виразу виконується <оператор1>, інакше цей оператор пропускається:

```
var
      x, y, max: Integer;
begin
.....
      if x > max then
         max := x;                {else –  нічого не робити}
      Y := x;
```

У даному прикладі змінна Y завжди матиме значення змінної X, а в MAX запам’ятовується максимальне значення X.

Оскільки будь-який з операторів, <оператор1> та <оператор2>, може бути будь-якого типу, у тому числі і умовним, і у той же час не кожен з «**вкладених**» умовних операторів може мати частину ELSE <оператор2>, то виникає неоднозначність трактування умов. Ця неоднозначність в TURBO-PASCAL вирішується таким чином: будь-яка частина ELSE, що зустрілася, відповідає найближчою до неї «зверху» частині THEN умовного оператора. Наприклад:

```
var
a,b,c,d : Integer;
begin
a := 1; b := 2; c:= 3; d := 4;
if  a > b then
   if  c < d then
     if  c < 0 then
        c := 0
     else
        a := b;
if a < b then
   if c > 0 then
     c := 0
   else
     a := b;
end.
```

Розглянемо програму (приклад 3.7), яка вводить довільне десяткове ціле число у діапазоні 0...15, перетворить його у шістнадцяткове число, і виводить на екран отриманий результат.

Приклад 3.7.

```
Program Hex;
{Програма вводить з клавіатури ціле число в діапазоні від 0 до 15, перетворить
його до шістнадцяткової системи числення і виводить результат на екран}
var
      n : Integer;          {Число, що вводиться}
      ch : Char;   {Результат}
begin
      Write ( ‘,n = ‘ ) ;
```

```
    ReadLn(n);          { Вводимо число }
    {Перевіряємо число на приналежність до діапазону 0...15}
    if (n >= 0) and (n <= 15) then
    begin               {Так, належить діапазону}
      if n < 10 then
        ch := chr(ord('0')+ n)
      else
        ch := chr(ord('A')+ n- 10);
      WriteLn('n = ',ch)
    end
    else            {Не належить діапазону}
      WriteLn('Помилка')
end.
```

У шістнадцятковій системі числення використовується 16 цифр у кожному розряді: цифри 0...9 позначають перші 10 можливих значень розряду, букви A...F – останні шість. У програмі враховується неперервність та впорядкованість множини цифр 0...9, букв A...F та їх код.

Розглянемо ще декілька прикладів.

Приклад 3.8.

```
{ Програма визначення коренів квадратного рівняння}
Program Quadrat;
Var   A, B, C: Real;
      Rad, Root1, Root2: Real;
Begin
  WriteLn('Введіть коеффициенти квадратного рівняння Ax^2+Bx+C=0:');
  Write('A=');
  ReadLn(A);
  Write('B=');
  ReadLn(B);
  Write('C=');
  ReadLn(C);
  Rad := Sqr(B)-4*A*C;
  If Rad < 0 Then
     WriteLn('Рівняння не має дійсного корня')
  Else
     If Rad = 0 Then
        WriteLn('Рівняння має один корінь = ',-B/(2*A):6:3)
    Else     Begin
       Root1 := (-B+Sqrt(Rad))/(2*A);
       Root2 := (-B-Sqrt(Rad))/(2*A);
       WriteLn('Рівняння має два корені: X1 = ',Root1:6:3', X2 = ',Root2:6:3)
    End;
  ReadLn
```

End.

У програмі прикладу 3.8 використовується вбудована функція Pascal Sqrt(X) – квадратний корень з X (при X>0).

Щоб не заплутатися у структурі цієї програми, слід пам'ятати таке правило: **else завжди відноситься до останнього оператора if.** Якщо ж у програмі потрібно, щоб else відносилося до одного з попередніх if, то доведеться скористатися складеним оператором. Наприклад, користувач вводить натуральне число, задача програми – поставити слово «студент» в потрібну форму у поєднанні з числівником (наприклад: 1 студент, 3 студента, 9 студентів, тощо).

```
begin
  write('Число студентів (1..20)--> '); readln(n);
  write(n,' студент');
  if n<5 then begin
     if n>1 then writeln('а');
  end
  else
     writeln('ів');
  readln;
end.
```

У даному прикладі довелося використовувати складений оператор (begin ... end;), для того щоб частина else відносилася не до оператора if n>1, а до if n<5.

Приклад 3.9.

```
{Відомі площі кола і квадрата. Визначити: }
{  a. Чи вміщується коло у квадраті; }
 { b. Чи вміщується квадрат у колі. }
Program Square_Circle;
Var a,r,s1,s2:real;
Begin
      Write('Введіть площу кола та квадрата: ');
      Readln(s1, s2);
      A:=sqrt(s2);        {сторона квадрата}
      R:=sqrt(s1/3.14); {радіус кола}
      If (r <= a/2) then
      writeln (' коло вміщується у квадрат')
         Else writeln (' коло не вміщується у квадраті');
         If (a*sqrt(2)) <= (2*r) then
      writeln ('квадрат вміщується у колі')
  Else Writeln('квадрат не вміщується у колі');
  Readln
End.
```

Приклад 3.10.

```
{Дано тризначне число. З'ясувати, чи є воно паліндромом ("перевернутим"), }
{тобто таким числом, десятковий запис якого читается одинаково зліва }
```

{ направо і справа наліво. }

```
Program Palindrom;
Var a,b,c:integer;
Begin
      Write('введите трехзначное число: ');    Readln(a);
      If (a div 100)=(a mod 10) then
      writeln (' це паліндром ')
            Else Writeln(' це не паліндром');
            Readln
      End.
```

Відзначимо, що не кожен оператор у прикладах з оператором if закінчується крапкою з комою. Хоча більшість операторів у програмі на Pascal закінчуються крапкою з комою, після деяких операторів крапка з комою не ставиться (не обов'язкова). Ми вже згадували одне виключення – зарезервовані слова begin і end. Правила вживання крапки з комою декілька умовні, але узгоджені. Необхідно пам'ятати три принципи:

1. Кожен опис змінної і константи закінчується крапкою з комою.
2. Кожен оператор в тілі програми завершується крапкою з комою, якщо одразу вслід за ним не йдуть зарезервовані слова end, else або until.
3. Після певних зарезервованих слів, таких як then, else, var, const та begin, ніколи не ставиться крапка з комою.

3.9.3. Розгалуження за низкою умов

Оператор if дозволяє виконувати переходи на ту або іншу гілку за значенням умови. Використовуючи декілька операторів if, можна проводити розгалуження за послідовностю умов. Але коли умов багато, такий підхід є достатньо одноманітним і виснажливим. Мова Pascal надає для цих цілей іншу управляючу структуру – оператор **case** (читається "кэйс", перекладається "випадок"), яка дозволяє побудувати розгалуження за низкою умов у формі значно зручнішій для читання програми.

```
CASE  <вираз> of
      <значення>:  <оператор>;
            . . . . .
      <значення>:  <оператор>;
[Else              <оператор>;]
End;
```

Квадратні дужки означають те, що частина else може бути відсутньою.

Вираз у простих випадках може бути цілочисельним або символьним. Як варіанти можна застосовувати:

1. Константний вираз такого ж типу, як і вираз після case. Константний вираз відрізняється від звичайного тим, що не містить змінних і викликів функцій, тим самим він може бути обчислений на етапі компіляції програми, а не під час виконання.
2. Інтервал, наприклад: 1..5, 'a'..'z'.
3. Список значень або інтервалів, наприклад: 1, 3, 6..8, 10, 12.

Виконується оператор case таким чином: обчислюється вираз після слова case і по порядку перевіряється, підходить отримане значення під будь-який варіант, чи ні. Якщо підходить, то виконується відповідний цьому варіанту оператор, інакше – є два варіанти. Якщо в операторі case записана частина else, то виконується оператор після else, якщо ж цієї частини немає, то не відбувається взагалі нічого.

Ось приклад простого калькулятора із застосуванням оператора Case.

```
Program Calc;
VAR a,b,rez      : Real;   {a i b –  два числа, rez-результат}
    Oper  : Char;   {oper  –  знак арифметичної дії}
BEGIN
    WriteLn('Програма-Калькулятор');
    Write('Введіть 1-е число: ');
    ReadLn (a);          {введення 1-го операнда}
    Write('Введіть операцію арифметичної дії (+, –  * або /): ');
    ReadLn (oper);       {введення арифметичної операції}
    Write('Введіть 2-е число: ');
    ReadLn (b);          {введення 2-го операнда}
    case oper of
        '+'     : rez:=a+b;
        '-'     : rez:=a-b;
        '*'     : rez:=a*b;
        '/'     : rez:=a/b;
        else    WriteLn('Неопізнана операція')
    end;
    WriteLn(a:11:8, oper, b:11:8, '=',rez :11:8);
    ReadLn
END.
```

Приклад 3.11.

```
{ Ця програма вводить цілі числа у діапазоні 0-9 і друкує їх у словесній формі}
Program Digits;
Var   Digit: integer;      {Введення користувача}
Begin
    WriteLn('Введіть число в діапазоні 0-9:');
    Read(Digit);
    Case Digit of
        0: writeln('нуль');
        1: writeln('один');
        2: writeln('два');
        3: writeln('три');
        4: writeln('чотири');
        5: writeln('п'ять');
        6: writeln('шість');
        7: writeln('сім');
```

```
            8: writeln('вісім');
            9: writeln('дев'ять');
        Else    Writeln('Це число поза діапазоном 0-9');
        End;
        Readln;
End.
```

3.9.4. Оператори повторів (циклів)

На попередніх заняттях ми познайомилися з оператором присвоєння (:=); оператором умови *if_then_else;* простим і складеним операторами; функціями *Sqrt, Sqr;* процедурами *ClrScr, Write, WriteLn, Read, ReadLn*; правилами складання ідентифікаторів і програм.

В принципі, цих знань і знайомства з іншими вбудованими функціями і процедурами вже достатньо для виконання обчислень за будь-якою формулою із заданими значеннями її аргументів. Проте в реальних завданнях часто вимагається виконувати одні і ті ж оператори по кілька разів. Наприклад, виникає необхідність обчислення функції одразу для цілого ряду значень її аргументів.

Наведемо приклад подібного завдання. Вимагається обчислити значення функції $y=ax^2+x$ в інтервалі зміни аргументу x від 0 до 10 з кроком $dx=2$ для $a=5$. Далі показаний варіант програмної реалізації рішення цієї дуже простої задачі.

```
Program Raschet;
Uses Crt;
Const a = 5;
Var  x,y:Integer;   { Цілі числа в інтервалі від -32 768 до 32 767, 2 байта}
BEGIN
        ClrScr;
        x :=  0;
        y := a*Sqr(x)+x;
        WriteLn('x= ', x:2,  ' y=', y:4);
        x:= 2;
        y := a*Sqr(x)+x;
        WriteLn('x= ', x:2,  ' y=', y:4);
        . . . .{ і так далі }
        x := 10;
        y := a'Sqr(x)+x;
        WriteLn('x= ', x:2,  ' y=', y:4);
        ReadLn;
END. { Raschet }
```

Як ви, напевно, помітили, тіло програми складається з низки операторів, які призначені для розрахунку та друку значень y при конкретних значеннях x,

що повторюється. Якщо інтервал *x* збільшити, наприклад, у 100 разів (чи крок *dx* зменшити у 100 разів), то й у 100 разів збільшиться тіло програми.

Зрозуміло, що проілюстрований у програмі Raschet шлях рішення задачі нерозумний. Для виконання обчислень, що повторюються, у будь-якій мові програмування є оператор циклу. Познайомимося з його можливостями для Turbo Pascal на прикладі деяких конкретних завдань.

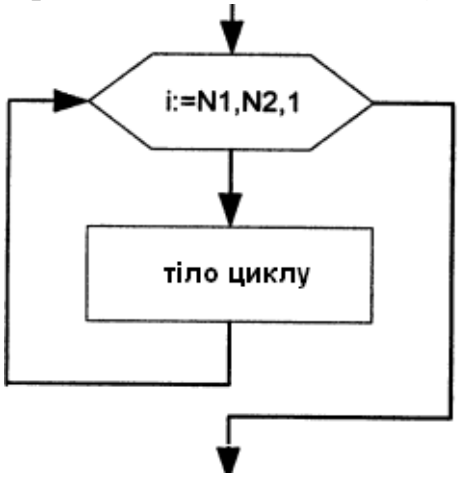
Цикл – це послідовність операторів, яка може виконуватися більше одного разу.

Можливі різні варіанти: виконувати фрагмент програми фіксовану кількість разів, виконувати, поки деяка умова є істинною, тощо. У зв'язку з наявністю варіантів в Pascal існує 3 типи циклів, за допомогою яких можна запрограмувати фрагменти програм, що повторюються.

Рахунковий оператор циклу FOR має таку структуру:

FOR <пар_цик> := <поч_знач> TO <кін_знач> DO <оператор>.

Тут FOR, TO, DO – зарезервовані слова (FOR читається "фо", перекладається "для"; TO читається "ту", перекладається "до"; DO читається "ду", перекладається "виконати");



<**пар_цик**> – параметр циклу – змінна типу INTEGER (точніше, будь-якого порядкового типу, див. гл.4);

<**поч_знач**> **–** початкове значення – вираз того ж типу;

<кін_знач> – кінцеве значення – вираз того ж типу;

<**оператор**> **–** довільний оператор TURBO-PASCAL.

Умовне позначення на схемах алгоритмів оператора FOR має такий вигляд:

При виконанні оператора FOR спочатку обчислюється вираз <почат_знач> і здійснюється присвоєння <пар_цик> : = <почат_знач>. Після цього циклічно повторюється:

1) перевірка умови <пар_цик> ≤ <кінц_знач>; якщо умова не виконана, оператор FOR завершує свою роботу;
2) виконання оператора <оператор>; збільшення змінної <пар_цик> на одиницю.

Наприклад, конструкція for i:=1 to 200 do читається так: *Для **i**, що змінюється від **1** до **200**, виконуй оператор (оператори), що стоїть після слова **do**.*

При виконанні циклу відбувається наступне: змінній *i* присвоюється початкове значення (1), потім виконується оператор (простий або складений), після цього до *i* додається 1, *i* перевіряється, чи не стало значення *i* рівним кінцевому (200). Якщо ні, то знову виконується оператор, додається 1, і т.і.

Як ілюстрація застосування оператора FOR розглянемо програму, що здійснює введення з клавіатури довільного цілого числа N та обчислення суми всіх цілих чисел від 1 до N (приклад 3.12).

Приклад 3.12.

```
Program Summ_of_Integer;
var
     i, n, s : Integer;
begin
     Write('N = ');
     ReadLn(n);                    {Вводимо N}
     s := 0;                       {Початкове значення суми}
     for i : = 1 to n do           {Цикл підрахунку суми}
        s : = s + i;
     writeln('Сума = ',s)          {Виводимо результат}
end.
```

Приклад 3.13. Золотий переріз та числа Фібоначчі.

Ціле завжди складається з частин, частини різної величини знаходяться у певному відношенні одна до одної та до цілого (пропорції). Принцип золотого перерізу – вищий прояв структурної та функціональної досконалості цілого і його частин в мистецтві, науці, техніці та природі.

У математиці ***пропорцією*** називають рівність двох відношень:

$$a : b = c : d.$$

Відрізок прямої *AB* можна розділити на дві частини такими способами:

- на дві рівні частини – $AB : AC = AB : BC$;
- на дві нерівні частини в будь-якому відношенні (такі частини пропорції не утворюють);
- коли $AB : AC = AC : BC$, то останнє і є золотий переріз або ділення відрізка у крайньому та середньому відношенні. **Золотий переріз** – це таке пропорційне ділення відрізка на нерівні частини, при якому весь відрізок так відноситься до більшої частини, як сама велика частина відноситься до меншої; або іншими словами, менший відрізок так відноситься до більшого, як більший до всього

$$a : b = b : c \text{ або } c : b = b : a.$$

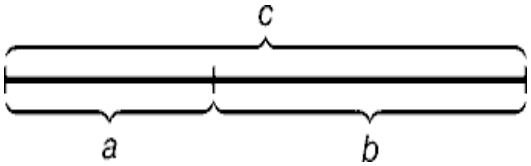


Рис. 3.1. Геометричне зображення золотої пропорції

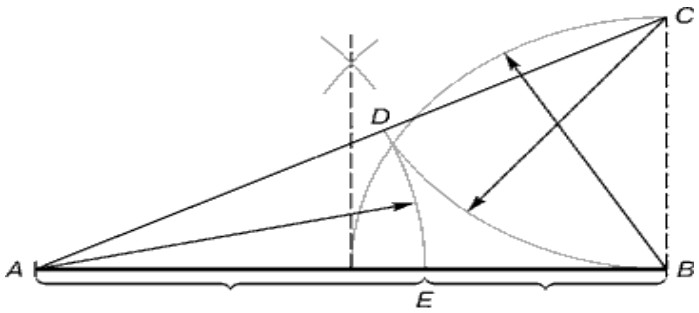


Рис. 3.2. Ділення відрізка прямої за золотомим перерізом
BC = 1/2 *AB*; *CD* = *BC*

Практичне знайомство із золотим перерізом починають з ділення відрізка прямої у золотій пропорції за допомогою циркуля та лінійки (рис. 3.2).

З точки *B* ставиться перпендикуляр, який дорівнює половині *AB*. Отримана точка *C* з'єднується лінією з точкою *A*. *На* отриманій лінії відкладається відрізок *BC*, що закінчується точкою *D*. Відрізок *AD* переноситься на пряму *AB*. Отримана при цьому точка *E* ділить відрізок *AB* у співвідношенні золотої пропорції.

Відрізки золотої пропорції виражаються нескінченим ірраціональним дробом *AE* = 0,618..., якщо *AB* прийняти за одиницю, *BE* = 0,382... Для практичних цілей часто використовують наближені значення 0,62 і 0,38. Якщо відрізок *AB* прийняти за 100 частин, то велика частина відрізання дорівнює 62, а менша – 38 частинам. Властивості золотого перерізу описуються рівнянням: $x^2 + x - 1 = 0$. Роз'вязання цього рівняння:

$$X_{1,2} = (-1 \pm \sqrt{5})/2$$

З історією золотого перерізу непрямим чином пов'язано ім'я італійського математика монаха Леонардо з Пізи (Пізанського), відомішого під ім'ям Фібоначчі (син Боначчі). Він багато подорожував по Сходу, познайомив Європу з індійськими (арабськими) цифрами. У 1202 р. вийшла у світ його математична праця «Книга про абак» (рахункова дошка), в якій були зібрані всі відомі на той час задачі. Одна із задач свідчила «Скільки пар кроликів в один рік від однієї пари народиться». Роздумуючи на цю тему, Фібоначчі збудував такий ряд чисел:

Місяці	0	1	2	3	4	5	6	7	8	9	10	11	12	…
Пари кроликів	0	1	1	2	3	5	8	13	21	34	55	89	144	…

Ряд чисел 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 і так далі відомий як ряд Фібоначчі. Особливість послідовності чисел полягає у тому, що кожен її член, починаючи з третього, дорівнює сумі двох попередніх 2 + 3 = 5; 3 + 5 = 8; 5 + 8 = 13, 8 + 13 = 21; 13 + 21 = 34 і так далі. Тобто, підрахувати наступний (n-ий) член послідовності Фібоначчі (Фn) можна знайти по наступній рекурентній формулі (використання числових послідовностей, наступні значення яких

можна знайти із попередніх, називають ***рекурентними***): Фn= Фn-1 + Фn-2 (для n > 2). А відношення суміжних чисел ряду наближається до відношення золотого перерізу. Отже, 21 : 34 = 0,6176, а 34 : 55 = 0,618. Це відношення позначається символом *Ф.* Тільки це відношення – 0,618 : 0,382 – дає безперервне ділення відрізка прямої у золотій пропорції, збільшення його або зменшення до безкінечності, коли менший відрізок так відноситься до більшого, як більший до всього.

Фібоначчі так само займався розв'язанням практичних потреб торгівлі: за допомогою якої найменшої кількості гирь можна зважити товар? Фібоначчі доводить, що оптимальною є така система гирь: 1, 2, 4, 8, 16... Ряд Фібоначчі (1, 1, 2, 3, 5, 8) і відкритий ним же «двійковий» ряд гирь 1, 2, 4, 8, 16... на перший погляд абсолютно різні. Але алгоритми їх побудови дуже схожі один на одного: у першому випадку кожне число є сума двух попередніх чисел 2 = 1 + 1, 3 = 2 + 1, 5 = 3 + 2..., у другому – це сума попереднього числа з самим собою 2 = 1 + 1, 4 = 2 + 2, 8 = 4 + 4 .

Приклад 3.13.

```
{ Обчислити і надрукувати перші 10 чисел Фібоначчі }
Program Fib;
Const n=10; {кількість чисел}
Var i,f1,f2,f3:Integer;
Begin
    f1 := 1; f2 := 1; {перші два числа Фібоначчі}
    For i := 3 To n Do
    Begin
        f3 := f1 + f2; {наступне число}
        Writeln(f3);
        f1 := f2; f2 := f3
    End
End.
```

Відзначимо дві обставини. По-перше, умова, яка керує роботою оператора FOR, перевіряється перед виконанням оператора <оператор>: якщо умова не виконується на самому початку роботи оператора FOR, то цикл не буде виконаний жодного разу. Інша обставина – крок нарощування параметра циклу постійний і дорівнює +1.

Існує інша форма оператора:

FOR <пар_цик>: = <поч_знач> DOWNTO <кін_знач> DO <оператор>

Заміна зарезервованого слова TO на DOWNTO (читається downto – "'даунту", перекладається буквально "знизу до") означає, що крок нарощування параметра циклу дорівнює (-1), а умова, що управляє, набуває вид <пар_цик> = <кін_знач>. Відповідно, для виходу з циклу повинна бути істиною не умова <пар_цик> > <кін_знач>, а умова <пар_цик> < <поч_знач>.

Приклад 3.12 можна модифікувати так, щоб зробити його придатним для підрахунку будь-яких сум – додатніх і від'ємних:

```
s := 0;
```

```
if n >= 0 then
  for i := 1 to n do
      s := s + i
else
  for i := -1 downto n do
      s : = s + i ;
```

Умовний оператор перевіряє умову виконання повтору циклу, а два оператори повтору лише по різному змінюють значення лічильника циклу (з інкрементом або з декрементом).

Оператор циклу WHILE з попередньою перевіркою умови має таку структуру:

WHILE <умова> DO <оператор>.

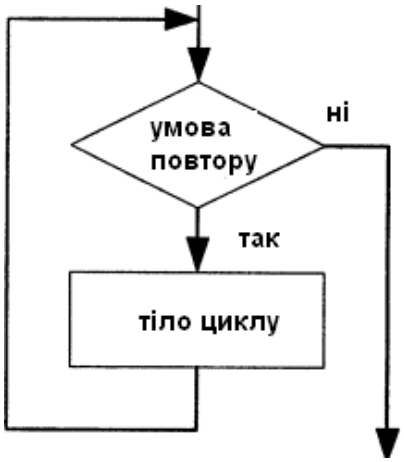
Тут WHILE, DO – зарезервовані слова (читається "вайл", слово do – "ду", уся конструкція перекладається так – Поки умова є істинною, виконуй оператор);

<умова> – вираз логічного типу;

<оператор> – довільний оператор TURBO-PASCAL.

Умовне позначення на схемах алгоритмів оператора WHILE має вигляд:

Якщо вираз <умова> має значення TRUE, то виконується <оператор>, після чого обчислення виразу <умова> та його перевірка повторюються. Якщо <умова> має значення FALSE, оператор WHILE припиняє свою роботу. У цьому випадку оператор може не виконатися жодного разу, оскільки умова перевіряється до виконання. Під оператором тут розуміється або простий, або складений оператор (тобто декілька операторів, в операторних дужках begin ... end).



Розглянемо приклад 3.14, який ілюструє використання оператора WHILE. Знайдемо так зване «машинне эпсилон» – таке мінімальне дійсне число, що не дорівнює нулю і яке після збільшення його до 1.0 ще дає результат, який відрізняється від 1.0.

Приклад 3.14.

```
Program EpsilonDetect;
{Програма обчислює і виводить на екран значення "Машиннй эпсилон"}
var
    epsilon: Real;
begin
    epsilon := 1;
```

```
        while epsilon/2 + 1 > 1 do
            epsilon := epsilon/2;
        WriteLn('Машинне эпсилон = ',epsilon)
end.
```

У читача, який звик до безперервної дійсної арифметики, може викликати подив твердження про те, що у дискретній машинній арифметиці завжди існують такі числа 0<X<eps, що 1.0+X=1.0. Річ у тім, що внутрішнє представлення типу REAL може дати «лише» приблизно 10^{14} можливих комбінацій значущих розрядів у відведених для нього 6 байтах.

Звичайно ж, це дуже велике число, але воно неспівставне з нескінченною безліччю дійсних чисел. Апроксимація нескінченної безперервної множини дійсних чисел є кінцевою (нехай навіть і дуже великою) множиною їх внутрішнього машинного уявлення і призводить до появи «машинного епсилон».

Наведемо ще декілька прикладів, які ілюструють використання оператора WHILE.

Приклад 3.15.

```
{ Знаходження середнього арифметичного послідовності чисел }
program Aver1;
Const   ListSize = 5;
Var   Sum, Number: Real;
        Count: Integer;
Begin
  Sum   := 0;
  Count := 0;
  WriteLn('Введіть ',ListSize,'-елементний список чисел:');
  While Count < ListSize do
    Begin
      Read(Number);
      Sum := Sum + Number;
      Count := Count + 1;
    end;
  WriteLn('Середнє = ',Sum/Count);
  ReadLn;   {Для затримки результатів програми на екрані}
End.
```

Приклад 3.16.

```
{ Знаходження середнього арифметичного послідовності чисел }
{ Припинення введення за допомогою стоп-коду}
program Aver2;
Const   Stop = -1; {Стоп-код}
Var   Sum, Number: Real;
        Count: Integer;
Begin
        Sum   := 0;
        Count := 0;
        WriteLn('Введіть список чисел ',Stop, 'що закінчується');
```

```
    Read(Number);
    While Number <> Stop do
    Begin
       Sum := Sum + Number;
        Count := Count + 1;
        Read(Number)
    end;
    WriteLn('Середнє = ',Sum/Count);
    ReadLn;     {Для затримки результатів програми на екрані}
End.
```

Приклад 3.17.

Багато з математичних величин або значень функцій можуть бути представлені як суми безконечних послідовностей. Наприклад, число π може бути обчислено за формулою:

$$\pi/4 = 1-1/3+1/5-1/7+\ldots+(-1)^{n-1}/(2n-1)+\ldots$$

Чим більше членів ряду бере участь у додаванні, тим точнішим виходить потрібне значення. Різниця отриманої суми ряду і дійсної суми називається похибкою додавання. Цю похибку можна оцінити різними способами. Часто оцінюють n-й член. Якщо він досить малий, тобто менше деякого даного числа **eps**, то вважається, що знайдена сума задовільняє вимогам по точності.

```
{ Обчислення числа "ПІ" }
var   p, eps, elem: real;
      n: integer;
begin
      p := 0;
      n := 1;
      elem := 1; { початкове значення}
      write('Задайте точність обчислення ПІ-> ');
      readln(eps);
      writeln('Обчислення ПІ з точностью',eps:12:9);
      while elem >= eps  do begin
        elem := 1/(2*n-1);
        if (n MOD 2) = 0 then {визначення знаку}
             p := p-elem
        else
             p := p+elem;
        n := n+1;
      end;
      p := p*4;
      writeln('Значення ПІ з точністю ',eps:12:9', дорівнює ',p:12:9);
      writeln('При заданій точності підсумовано ',n', членів ряду.');
      readln;
end.
```

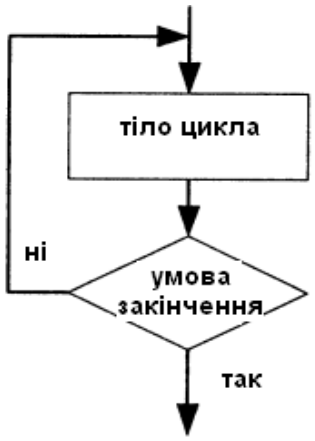
Оператор циклу REPEAT... UNTIL з постперевіркою умови:

REPEAT <тіло_цикла> UNTIL <умова>.

Тут REPEAT, UNTIL – зарезервовані слова (конструкція repeat . until читається "ри'пит.....ан'тил...", а перекладається "повторюй.....доти, доки виконується умова);

<**тіло_циклу**> **–** довільна послідовність операторів TURBO-PASCAL, які розділені крапкою з комою;

<**умова**> **–** вираз логічного типу.



Умовне позначення на схемах алгоритмів оператора REPEAT має вигляд:

На відміну від оператора WHILE, оператори <тіло_циклу> у REPEAT виконуються хоча б один раз, після чого обчислюється вираз <умова>: якщо його значення є FALSE, оператори <тіло_циклу> повторюють своє виконання, інакше оператор REPEAT. . . UNTIL завершує свою роботу.

Для ілюстрації застосування оператора REPEAT... UNTIL модифікуємо програму з прикладу 3.6. Модифікація (приклад 3.18) полягає у тому, що програма весь час повторюватиме цикл введення символу та друку його коду доти, доки черговим символом не буде символ CR (вводиться клавішею Enter).

Приклад 3.18.

```
Program Codes_of_Chars;
{Програма вводить символ і виводить на екран його код. Для завершення роботи програми потрібно двічі натиснути Enter}
var
      ch : Char;   {Символ, який вводиться}
const
      CR = 13;     {Код символу CR}
begin
      repeat
         ReadLn(ch);
         WriteLn(ch' = ',ord(ch));
      until ord(ch)= CR
end.
```

Приклад 3.19.

```
{ Ця програма визначає, чи є число простим }
program Prime;
Var          Number: Integer;        { Число, яке досліджується }
      Count:  Integer;          {Можливий дільник}
Begin
   WriteLn('Яке число має бути перевірено?: ');
   Read(Number);
   Count := 1;
   Repeat
```

```
      Count := Count+1;
    until (Number mod Count)=0;
    If Count = Number then
      WriteLn(Number', є простим числом')
    else
      WriteLn(Number', ділиться на ',Count);
  ReadLn;
  End.
```

Приклад 3.20.

```
{ Гра "Вгадай число" }
{ Використовується функція Random(X), що генерує випадкове число від 0 до
X}
Const NPOP=4; { кількість спроб, що надається гравцеві }
Var    comp: integer; { число, "яке замислив" комп'ютер }
       igrok: integer; { варіант гравця }
       n: integer; { к-ть спроб, зроблених гравцем }
begin
  Randomize; { ініціалізація генератора випадкових чисел }
  comp:=Random(9)+1; { комп'ютер замислив число від 1 до 10}
  Writeln('Гра "Вгадай число". ');
  writeln('Комп'ютер "замисли" число від 1 до 10.');
  writeln('Вгадайте його за ',NPOP', спроб.');
  writeln('Введіть число і натисніть <Enter>.');
  n := 0;
  repeat
    n := n+1;
    write(n, ' спроба->');
    readln(igrok);
  until (n = NPOP) or (comp = igrok);
  if comp = igrok then
    writeln('Ви виграли!')
  else
    writeln('Ви програли! Комп'ютер задумав число ',comp);
  readln;
end.
```

Зверніть увагу на те, що пара REPEAT... UNTIL подібна до операторних дужок begin. .. end, тому перед UNTIL ставити крапку з комою необов'язково.

Для гнучкого управління циклічними операторами FOR, WHILE та REPEAT до складу TURBO-PASCAL включено дві процедури:

BREAK – реалізує негайний вихід із циклу; дія процедури полягає у передачі управління операторові, що стоїть одразу за кінцем циклічного оператора;

CONTINUE – забезпечує дострокове завершення чергового проходу циклу; еквівалент передачі управління в самий кінець циклічного оператора.

Введення у мову цих процедур практично виключає необхідність використання операторів безумовного переходу GOTO (читається " ‘гоуту", перекладається "йди до").

3.9.5. Вкладені цикли

Реальна програма на Pascal є складною мозаїкою з циклічних та розгалужених частин, які вкладені одна в одну. Ми вже бачили у прикладах, як в оператор if був вкладений інший оператор if. У свою чергу в оператор циклу можуть бути вкладені інші оператори, у тому числі і оператори циклу, і так до «безкінечності».

Поставимо, наприклад, собі завдання – надрукувати таблицю множення у такому вигляді:

1*1= 1 1*2= 2 1*3= 3 1*4= 4 1*5= 5 1*6= 6 1*7= 7 1*8= 8 1*9= 9

2*1= 2 2*2= 4 2*3= 6 2*4= 8 2*5= 10 2*6= 12 2*7= 14 2*8= 16 2*9= 18

. . .

9*1= 9 9*2= 18 9*3= 27 9*4= 36 9*5= 45 9*6= 42 9*7= 63 9*8= 72 9*9= 81

Почнемо з малого – нехай потрібно надрукувати 1*1=1

Ось фрагмент № 1 програми:

```
a:=1;            { фрагмент № 1 }
b:=1;
proizv:=a*b;
Write(a, ‘*’ ,b, ‘=‘ ,proizv)
```

Тут в операторі Write 5 елементів:

- співмножник a;
- символ знаку множення ‘*’;
- співмножник b;
- символ ‘=‘;
- значення добутку proizv.

Ускладнимо завдання. Спробуємо змусити комп’ютер надрукувати перший рядок таблиці.

Тут нам потрібно розв’язати 9 елементарних задач на обчислення добутку, першу з яких розв’язує фрагмент 1. Усі вони дуже схожі і розрізняються лише значенням другого співмножника. Таким чином, для розв’язання кожної з 9 задач підійшов би наш фрагмент 1, якби у ньому в операторі *b:=1* замість одиниці стояла потрібна цифра. У нашому випадку ідеально підходить оператор for:

```
a:=1;          {фрагмент № 2}
for b:=1 to 9 do begin
    proizv:=a*b;
   Write(a, ‘*’ ,b, ‘=‘ ,proizv,  ‘ ‘ )
```

```
end;
```

Для того щоб друк був акуратним, оператор Write ми доповнили символом пробілу ‘ ‘. Він потрібний для того, щоб окремі стовпчики таблиці не зливалися.

Наступний рівень ускладнення – останній – надрукувати не один рядок таблиці, а дев’ять. Для цього фрагмент 2 має бути виконаний 9 разів, кожного разу – з новим значенням *a*. Щоб цього досягти, “обіймемо” фрагмент 2 оператором for так само, як ми це зробили з фрагментом 1.

```
for a:=1 to 9 do    {фрагмент № 3}
    for b:=1 to 9 do begin
        proizv:=a*b;
       Write(a, ‘*’ ,b, ‘=‘ ,proizv,  ‘ ‘ )
    end;
end;
```

Друкувати фрагмент 3 програми буде неакуратно. Наведемо остаточний запис програми з необхідними доповненнями для акуратного друку, а також для зручності пояснень забезпечимо програму коментарями з нумерацією рядків:

```
VAR a,b,proizv: Integer;                    {1}
BEGIN                                       {2}
   for a:=1 to 9 do begin                   {3}
      WriteLn;                              {4}
      for b:=1 to 9 do begin                {5}
         proizv:=a*b;                       {6}
        Write(a, ‘*’, b, ‘=‘, proizv:3, ‘ ‘ ) {7}
     end; {for b}                           {8}
  end; {for a}                              {9}
END.                                        {10}
```

WriteLn потрібний для того, щоб кожен новий рядок таблиці починався з нового рядка екрану.

Формат *:3* для змінної proizv (7 рядок) означає, що на зображення добутку на екрані відведено три позиції. Формат у нашому прикладі потрібний для того, щоб різні за кількістю цифр добутки (наприклад, 5 і 25) займали на екрані однакове за розміром місце, інакше не вийде у нас акуратних стовпчиків у таблиці.

У цілому програма ілюструє ідею вкладених циклів, коли один внутрішній цикл вкладений всередину іншого, зовнішнього. У нас тіло зовнішнього циклу (рядки 4 і 5) виконується 9 разів, а тіло внутрішнього (рядки 6, 7) – 81 раз, оскільки на кожне виконання рядка 5 воно виконується 9 разів.

3.9.6. Мітки та оператори переходу

Можна теоретично показати, що розглянутих операторів цілком достатньо для написання програм будь-якої складності. У цьому плані наявність у мові операторів переходу здається зайвою. Більш того, сучасна технологія

структурного програмування заснована на принципі «програмувати без GOTO»: вважається, що зловживання операторами переходу ускладнює розуміння програми, робить її заплутаною і складною при відлагодженні. Проте, у деяких випадках використання операторів переходу може спростити програму.

Оператор переходу має вигляд:

GOTO <мітка>

Тут GOTO – зарезервоване слово (перейти [на мітку]); <мітка> – мітка.

Мітка в TURBO-PASCAL – це довільний ідентифікатор, що дозволяє іменувати деякий оператор програми і таким чином посилатися на нього. В цілях сумісності із стандартною мовою Pascal у мові припускається використання також цілих чисел без знаку як міток.

Мітка розташовується безпосередньо перед оператором, який помічається, та відділяється від нього двокрапкою. Оператор можна позначати кількома мітками, які у цьому випадку відділяються одна від одної двокрапкою. Перш ніж з'явитися у програмі, мітка повинна бути описана. Опис міток складається із зарезервованого слова LABEL (читається "лейбл", перекладається "мітка"), за яким слідує список міток:

```
label
      loop, 1b1, 1b2;
begin
.......
      goto 1b1;
.......
       loop: ......
.......
      1b1:1b2: ......
........
      goto 1b2;
.......
```

Дія оператора GOTO полягає у передачі управління відповідному поміченому оператору.

При використанні міток необхідно керуватися наступними правилами:

- мітка, на яку посилається оператор GOTO, повинна бути описана у розділі описів і вона обов'язково повинна зустрітися будь-де у тілі програми;
- мітки, які описані у процедурі (функції), локалізуються у ній, тому передача управління ззовні процедури (функції) на мітку всередині неї неможлива.

Контрольні запитання для самоперевірки

1. Що являє собою ідентифікатор та з чого він складається?

2. Що являє собою елементарна конструкція?
3. З чого утворюється програма?
4. Що являє собою компіляція?
5. Який склад оператора присвоєння та його призначення?
6. Який порядок переходу до режиму відлагодження?
7. Який порядок створення програми на мові PASCAL?
8. Як достроково припинити виконання програми?
9. З яких частин складається структура програми?
10. У чому полягає форматне виведення дійсних даних?
11. Як вивести результати на принтер?
12. Які елементарні типи даних обробляються комп'ютером?
13. Які існують математичні функції для обробки дійсних даних?
14. Які існують функції для обробки символьних даних?
15. У чому полягає сумісність типів даних та припустимі операції над ними?
16. Яке призначення умовного та складового операторів?
17. Який оператор реалізує розгалуження за низкою умов?
18. Яке призначення та конструкція оператора циклу повтору?
19. Яке призначення та конструкція оператора циклу з передумовою?
20. Яке призначення та конструкція оператора циклу з після умовою?
21. У чому полягають правила використання міток?

4. Елементи мови

4.1. Алфавіт

Алфавіт – сукупність припустимих у мові символів (або груп символів, що розглядаються як єдине ціле). Алфавіт мови включає букви, цифри, шістнадцяткові цифри, спеціальні символи, пропуски та зарезервовані слова.

Букви – це букви латинського алфавіту від a до z та від A до Z, а також знак підкреслення _ (код ASCII 95). У TURBO-PASCAL немає відмінності між прописними та строковими буквами алфавіту, якщо тільки вони не входять у символьні та строкові вирази.

Кожна шістнадцяткова цифра має значення від 0 до 15. Перші 10 значень позначаються арабськими цифрами 0...9, останні шість – латинськими буквами A...F або a...f.

Спеціальні символи TURBO-PASCAL – це символи: + – * / = ,’ . : ; < > [ ] ( ) { } ^ @ $ #

До спеціальних символів належать також такі пари символів: <> <= >= := (* *). У програмі ці пари символів не можна розділяти пробілами, якщо вони використовуються як знаки операцій відношення або обмежувачі коментаря. У TURBO-PASCAL є такі зарезервовані слова:

and	**end**	**nil**	**shr**
asm	**file**	**not**	**string**
array	**for**	**object**	**then**
begin	**function**	**of**	**to**
case	**goto**	**or**	**type**
const	**if**	**packed**	**unit**
constructor	**implementation**	**procedure**	**until**
destructor	**in**	**program**	**uses**
div	**inline**	**record**	**var**
do	**interface**	**repeat**	**while**
downto	**label**	**set**	**with**
else	**mod**	**shl**	**xor**

Зарезервовані слова не можуть використовуватися як ідентифікатори.

4.2. Ідентифікатори

Ідентифікатори у TURBO-PASCAL – це імена констант, змінних, міток, типів, об’єктів, процедур, функцій, модулів, програм та ділянок у записах. Ідентифікатори можуть мати довільну довжину, але значущими (унікальними в області визначення) є тільки перші 63 символи.

Ідентифікатор завжди починається буквою, за якою можуть слідувати букви та цифри. Згадаємо, що буквою вважається також символ підкреслення, тому ідентифікатор може починатися цим символом і навіть складатися тільки з

одного або декількох символів підкреслення. Пробіли та спеціальні символи алфавіту не можуть входити в ідентифікатор.

Приклади правильних ідентифікаторів:

a, ALPHA, MyProgramIsBestProgram
date_27_sep_39, external, _beta

Приклади неправильних ідентифікаторів:

1Program {починається цифрою}
block#l {містить спеціальний символ}
My Prog {містить пробіл}
mod {зарезервоване слово}

4.3. Константи

Як константи у TURBO-PASCAL можуть використовуватися цілі, дійсні та шістнадцяткові числа, логічні константи, символи, рядки символів та ознака невизначеного покажчика NIL.

Цілі числа записуються із знаком або без нього за звичайними правилами і можуть мати значення -2147483648 до +2147483647. Слід врахувати, що, коли цілочисельна константа виходить за вказані межі, компілятор дає повідомлення про помилку. Такі константи повинні записуватися з десятковою крапкою, тобто визначатися як дійсні числа.

Дійсні числа записуються із знаком або без нього з використанням десяткової крапки та/або експоненціальної частини. Експоненціальна частина починається символом *e* або *E*, за яким можуть слідувати знаки «+» або «-» та десятковий порядок. Символ e (E) означає десятковий порядок і має сенс «помножити на 10 у ступені». Наприклад

3.14E5 – 3.14 помножити на 10 в ступені 5;

-17e-2 – мінус 17 помножити на 10 в ступені мінус 2.

Якщо у записі дійсного числа присутня десяткова крапка, перед крапкою та за нею повинна бути, хоча б одна цифра. Якщо використовується символ експоненціальної частини e (E), то за ним повинна слідувати, хоча б одна цифра десяткового порядку.

Шістнадцяткове число складається з шістнадцяткових цифр, яким передує знак долара $ (код 36 в ASCII). Діапазон шістнадцяткових чисел – від $00000000 ДО $FFFFFFFF.

Логічна константа – це або слово FALSE (хиба), або слово TRUE (істина).

Символьна константа – це будь-який символ ПК, оточений лапками:

‘z’ – символ z;

‘Ф’ – символ Ф.

Якщо необхідно записати власно символ апострофа, він подвоюється:

‘‘‘‘ – символ ‘ (апостроф).

Припускається використання запису символу шляхом вказівки його внутрішнього коду, якому передує символ # (код 35), наприклад:

#97 – символ a;

#90 – символ Z;
#39 – символ ‘;
#13 – символ CR.

Строкова константа – будь-яка низка символів (окрім символу CR – повернення каретки), яка оточена лапками. Якщо у рядку потрібно вказати символ апострофу, то він подвоюється, наприклад:

‘Це – рядок символів’;
‘That’‘s string.’.

Рядок символів може бути порожнім, тобто не мати ніяких символів у лапках, що оточують його. Рядок можна складати з кодів потрібних символів з передуючими кожному коду символами #, наприклад, рядок #83#121#109#98#11#108 еквівалентний рядку ‘ Symbol’.

4.4. Вирази

Основними елементами, з яких конструюється частина програми, яка виконується, є константи, змінні та звернення до функцій. Кожен з цих елементів характеризується своїм значенням і належить до будь-якого типу даних. За допомогою знаків операцій та дужок з них можна складати вирази, які фактично є правилами набуття нових значень.

Окремим випадком виразу може бути просто одиночний елемент, тобто константа, змінна або звернення до функції. Значення такого виразу має той самий тип, що й сам елемент. У більш загальному випадку вираз складається з кількох елементів (операндів) та знаків операцій, а тип його значення визначається типом операндів та видом застосованих до них операцій. Приклади виразів:

Y, 21, (a + b) * c, sin(t), a > 2, not Flag and (a = b).

4.5. Операції

У TURBO-PASCAL визначені наступні операції:

- унарні not @;
- мультиплікативні *, /, div, mod, and, shl, shr;
- адитивні +, -, or, xor;
- відношення =, <> <, >, <=, >=,in.

Пріоритет операцій спадає у вказаному порядку, тобто найвищим пріоритетом володіють унарні операції, найнижчим – операції відношення. Порядок виконання кількох операцій рівного пріоритету встановлюється компілятором за умови оптимізації кодів програми та не обов’язково зліва направо. При обчисленні логічних виразів операції рівного пріоритету завжди обчислюються зліва направо. Правила використання операцій з операндами різного типу наводяться у табл. 4.1.

Таблиця 4.1

Операція	Дія	Тип операндів	Тип результату
not	Заперечення	Логічний	Логічний
not	Те саме	Будь-який цілий	Тип операнда
@	Адреса	Будь-який	Вказівник
*	Множення	Будь-який цілий	Найменший цілий
*	Те саме	Будь-який дійсний	Extended
*	Перетин множин	Множинний	Множинний
/	Ділення	Будь-який дійсний	Extended
div	Цілочисельне ділення	Будь-який цілий	Найменший цілий
mod	Залишок від ділення	Те саме	Те саме
and	Логічне І	Логічний	Логічний
and	Те саме	Будь-який цілий	Найменший цілий
shl	Лівий зсув	Те саме	Те саме
shr	Правий зсув	Те саме	Те саме
+	Додавання	Те саме	Те саме
+	Те саме	Будь-який дійсний	Extended
+	Об’єднання множин	Множинний	Множинний
+	Зчеплення строк	Строковий	Строковий
-	Віднімання	Будь-який цілий	Найменший цілий
-	Те саме	Будь-який дійсний	Extended
or	Логічне АБО	Логічний	Логічний
or	Те саме	Будь-який цілий	Найменший цілий
=	Дорівнює	Простий або строковий	Логічний
<>	Не дорівнює	Те саме	Те саме

<	Менше	Логічний	Логічний
<=	Менше або дорівнює	Те саме	Те саме
>	Більше	Те саме	Те саме
>=	Більше або дорівнює	Те саме	Те саме

Унарна операція @ застосовується до операнда будь-якого типу та повертає результат типу POINTER, у якому міститься адреса операнда.

У TURBO-PASCAL визначені такі логічні операції:

not – логічне НІ (читається "нот"); **and –** логічне ТА (читається "енд");

or – логічне АБО (читається "о"); **xor –** виключнє АБО (читається "ксо", побітове логічне додавання); побітове додавання за модулем 2).

Логічні операції застосовні до операндів цілого і логічного типів. Якщо операнди – цілі числа, то результат логічної операції є також ціле число, біти якого (двійкові розряди) формуються з бітів операндів за правилами, які вказані в табл. 4.2.

Таблиця 4.2

Логічні операції над даними типу INTEGER (поразрядно)					
Операнд 1	Операнд 2	not	and	or	xor
1	-	0	-	-	-
0	-	1	-	-	-
0	0	-	0	0	0
0	1	-	0	1	1
1	0	-	0	1	1
1	1	-	1	1	0

До логічних операцій у TURBO-PASCAL зазвичай відносяться ще дві операції зсуву над цілими числами:

і **shl** j – зсув вмісту і на j розрядів ліворуч; молодші розряди, що звільнилися, заповнюються нулями;

і **shr** j – зсув вмісту і на j розрядів праворуч; старші розряди, що звільнилися, заповнюються нулями.

У цих операціях *i* та *j* – вирази будь-якого цілого типу.

Як бачимо, Pascal не має операції піднесення до ступеня. Але цей недолік легко вирішується, якщо скористатися наступною тотожністю: $X^Y = e^{YLn\ X}$.

Приклад 4.1

```
{Обчислення ступеня числа з використанням властивостей логарифмів }
{ A у ступені B дорівнює C. Логаріфмуємо обидві частини рівності і
отримуємо: B*Ln (A) = Ln (C) }
{ Нас цікавить значення C, тому обчислюемо E (exp) в ступені B*ln(A). }
{ Значення цього виразу дорівнює C, що і треба було обчислити. }
Program InStep;
var
   a: real; { число }
   b: real; { ступінь }
   c: real; { число у ступені }
begin
    writeln('Введіть число та покажчик ступеня');
    readln(a,b);
    c := exp(b*Ln(a));
    writeln(a:6:3', в ступені ',b:6:3' = ',c:6:3);
end.
```

Хоча це завдання можна розв'язати більш грубим способом, використовуючи асоціативність операції множення. Наприклад, для заданого цілого (дійсного) X обчислити X^{13}. Використовуючи асоціативність операції множення, вираз для X^{13} можна записати з таким розставлянням дужок:

X13 = X*X*X*X*X*X*X*X*X*X*X*X*X = {виконує 12 операцій множення}

(X*X)*(X*X)*(X*X)*(X*X)*(X*X)*(X*X)*X

З цього запису зрозуміло, що вираз (X*X) обчислюється тут шість разів (для того самого значення). Ми можемо обчислити його один раз, запам'ятати у новій змінній, а потім використовувати це значення: Y = X*X. Цей же прийом можна застосувати ще раз та отримати Z = Y*Y. Остаточно для обчислення X^{13} використовується вже лише п'ять операцій множення.

```
Program StepX13;
   var X,Y,Z : iteger;
begin
   read(X);
   Y := X*X;
   Z := Y*Y;
   writeln(X,'^13= ', Z*Z*Z*X)
end.
```

За допомогою програми прикладу 4.2 можна вивести на екран результат застосування логічних операцій до двох цілих чисел.

Приклад 4.2

{Програма вводить два числа і друкує результат застосування до них логічних операцій}

```
var
      n, m : integer;
begin
      while not EOF do begin
            Write('n,m=');   ReadLn(n,m);
            WriteLn( 'not  = ', not n '  ', not m);
            WriteLn( 'and = ', n and m);
            WriteLn( 'or    = ', n  or  m);
            WriteLn( 'xor  = ', n xor m);
            WriteLn( 'shl  = ', n shl m);
            WriteLn( 'shr  = ', n shr m);
      end
end.
```

У програмі організовується введення двох довільних цілих чисел та друк результату застосування до них всіх логічних операцій. Для виходу з програми слід натиснути Ctrl-z, та Enter. Клавіші Ctrl-z необхідно натиснути для закінчення введення з клавіатури масиву (файлу) чисел.

Ключове слово array (масив, читається "е'рэй") з визначенням ['A'.. 'Z'] of integer визначає змінну Leters як масив цілих чисел від 'A' до 'Z' (про масиви ми поговоримо пізніше). Функція EOF (End Of File –Кінець Файлу) перевіряє натиснення Ctrl-z.

Приклад 4.3.

{ Програма підрахунку частоти букв у веденному англійською мовою тексті}

```
Program  Leterfrec;
var    Leters: array ['A'.. 'Z'] of  integer;
       symbol, index: char;
begin
  while  not EOF(input)  do
  begin
    read (symbol);
    if (symbol >= 'A')  and  (symbol <= 'Z')  then
      leters[symbol]:= leters[symbol]+1;
  end;
  writeln ('Літера     Частота');
  for index:='A' to 'Z' do
     if  leters[index] <> 0 then
      writeln (index, leters[index]:15);
   readln;
end.
```

Логічні операції над логічними даними дають результат логічного типу за правилами, які вказані у табл. 4.3.

Таблиця 4.3

Логічні операції над даними типу Boolean					
Операнд 1	Операнд 2	not	and	or	xor
True	-	False	-	-	-
False	-	True	-	-	-
False	False	-	False	False	False
False	True	-	False	True	True
True	False	-	False	True	True
True	True	-	True	True	False

Контрольні запитання для самоперевірки

1. Що являють собою константи та якого типу вони бувають?
2. Що є виразом та який тип він може мати?
3. Які припустимі операції використовуються над даними різного типу?
4. Яка пріоритетність виконання операцій?
5. Як визначити адресу будь-якої змінної?
6. Як визначається тип виразу?
7. Які значення приймає цілий тип при застосуванні до нього логічних операцій?

5. Робота з різними типами даних Pascal

Прийшов час познайомитися з екзотичнішими типами даних, які застосовуються у Pascal. Подекуди ми повторимося в описі та властивостях цих типів даних, але це повинно принести вам тільки користь.

Будь-які дані, тобто константи, змінні, значення функцій або виразу, у TURBO-PASCAL характеризуються своїми типами. Тип визначає **діапазон припустимих значень**, які може мати той чи інший об'єкт, а також безліч **припустимих операцій**, які застосовні до нього. Крім того, тип визначає також і формат внутрішнього подання даних у пам'яті ПК.

TURBO-PASCAL характеризується розгалуженою структурою типів даних (рис.5.1). У TURBO-PASCAL передбачений механізм створення нових типів даних, завдяки чому загальна кількість типів, яка використовується у програмі, може бути скільки завгодно великою.

У цьому розділі наводиться докладний опис усіх типів, за винятком файлів, які розглядаються у наступному розділі, вказівники та процедурні типи нами розглядатися не будуть. Їх опис можна знайти в інших джерелах, і в наші плани це не входить, адже у нас "вступ до програмування", а не повний опис Pascal.

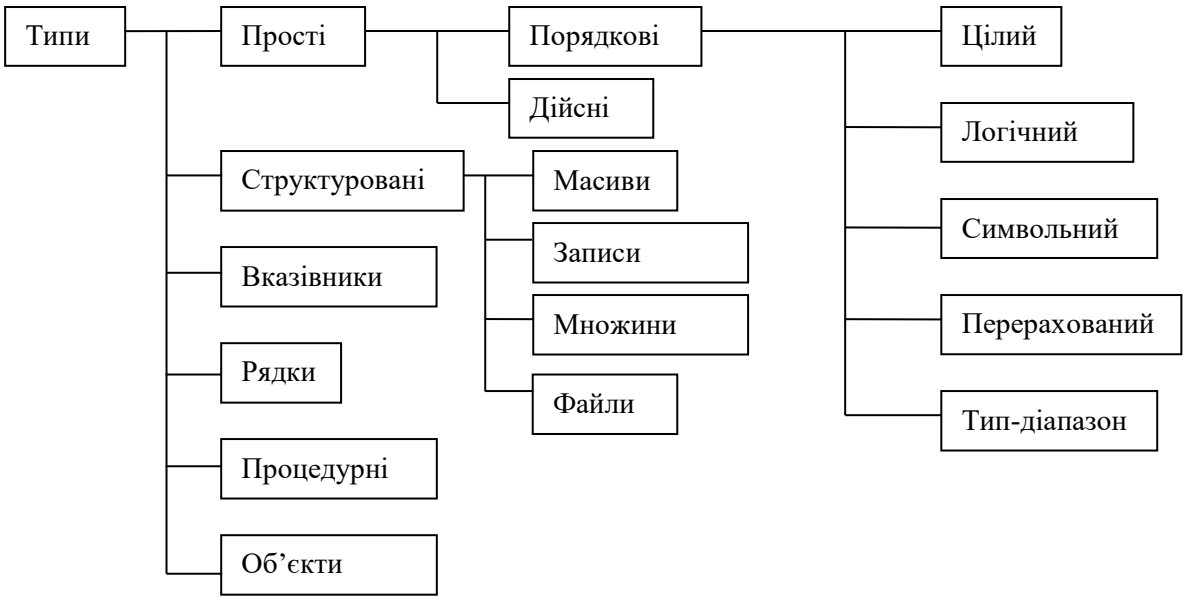


Рис. 5.1. Структура типів даних

5.1. Прості типи

До простих типів належать **порядкові типи та дійсні типи**. Порядкові ипи відрізняються тим, що кожен з них має кінцеву кількість можливих значень. Ці значення можна певним чином впорядкувати (звідси – назва типів) і, отже, з кожним з них можна співставити деяке ціле число – порядковий номер значення.

Дійсні типи, строго кажучи, теж мають кінцеве число значень, які визначаються форматом внутрішнього представлення дійсного числа. Проте кількість можливих значень дійсних типів настільки велика, що співставити з кожним з них ціле число (його номер) не уявляється можливим.

5.1.1. Порядкові типи

До порядкових типів належать (див. рис. 5.1) **цілий**, **логічний**, **символьний**, **такий, що перераховується** та **інтервальний тип** (або **обмежений тип**, **тип-діапазон**). До будь-якого з них застосовна функція ORD(X), яка повертає порядковий номер значення виразу X. Для цілих типів функція ORD(X) повертає само значення X, тобто ORD(X) = X для X, що належить будь-якому цілому типу. Застосування ORD(X) до логічного, символьного та такого, що перераховується, типів дає додатне ціле число в діапазоні від 0 до 1 (логічний тип), від 0 до 255 (символьний), від 0 до 65535 (такий, що перераховується). Тип–діапазон зберігає всі властивості базового порядкового типу, тому результат застосування до нього функції ORD(X) залежить від властивостей цього типу.

До порядкових типів можна також застосовувати функції:

PRED(X) – повертає попереднє значення порядкового типу (значення, яке відповідає порядковому номеру ORD(X), – 1), тобто ORD(PRED(X))= ORD(X) – 1;

SUCC(X) – повертає наступне значення порядкового типу, яке відповідає порядковому номеру ORD(X)+1, тобто ORD(SUCC(X))= ORD(X)+ 1.

Наприклад, якщо у програмі визначена змінна

```
var
c : Char;
begin
c := ‘5’ ;
end.
```

то функція PRED(C) повертає значення ‘4’, а функція SUCC(C) – значення ‘6’. Якщо уявити собі будь-який порядковий тип як впорядковану множину значень, що зростають зліва направо і що займають на числовій вісі деякий відрізок, то функція PRED(X) не визначена для лівого, а SUCC(X) – для правого кінця цього відрізка.

Цілі типи. Цілі числа в Pascal записуються у звичайному вигляді, наприклад: **0 +100, -56498.** Діапазон можливих значень цілих типів залежить від їх внутрішнього уявлення, яке може займати один, два або чотири байти. У табл. 5.1 наводиться назва цілих типів, довжина їх внутрішнього уявлення в байтах та діапазон можливих значень.

Таблиця 5.1. Цілі типи

Цілі типи		
Назва	Довжина, байт	Діапазон значень
Byte	1	0. . .255
ShortInt	1	-128. . .+127
Word	2	0. . .65535
Integer	2	-32768.. .+32767
LongInt	4	-2 147 483 648... +2 147 483 647

При використанні процедур та функцій з цілочисельними параметрами слід керуватися «вкладеністю» типів, тобто скрізь, де може використовуватися WORD (читається “ворд”), припускається використання BYTE (читається “байт”), але не навпаки. У LONGINT (скорочення від Long Integer – Довге Ціле, читається “лонг’инт”) «входить» INTEGER, який, у свою чергу, включає SHORTINT (читається “шорт’инт”).

Перелік процедур та функцій, які можуть бути застосовані до цілочисельних типів, наведений у табл. 5.2. Літерами b, s, w, i, l позначені вирази відповідно типу BYTE, SHORTINT, WORD, INTEGER та LONGINT, x – вираз будь-якого з цих типів; літери vb, vs, vw, vi, vl, vx позначають змінні відповідних типів. У квадратних дужках вказується необов’язковий параметр.

Таблиця 5.2. Процедури та функції цілочисельних типів

Стандартні процедури та функції, які можуть бути застосовані до цілих типів		
Звернення	Тип результату	Дія
Abs (x)	X	Повертає модуль x
Chr(b)	Char	Повертає символ за його кодом
dec (vx[, i])	-	Зменшує значення vx на i, а за відсутності i -на 1
inc(vx[, i])	-	Збільшує значення vx на i, а за відсутності i – на 1
Random (w)	Як у параметра	Повертає псевдовипадкове число, рівномірно розподілене в діапазоні 0...(w-l)
Sqr (x)	X	Повертає квадрат аргументу

При діях з цілими числами тип результату відповідатиме типу операндів, а якщо операнди відносяться до різних цілих типів, – типу того операнда, який має максимальну потужність (максимальний діапазон значень). Можливе

переповнення результату ніяким чином не контролюється, що може призвести до непорозумінь, наприклад:

```
var
        a : Integer;
        x, y : Real;
begin
  a := 32767;  {Максимально можливе значення типу INTEGER}
  x := a + 2;   {Переповнення при обчисленні цього виразу!}
  y := LongInt(a)+2; {Переповнення немає після приведення змінної до типу LongInt}
  WriteLn(x:10:0, y:10:0)
end.
```

Після виконання програми отримаємо результат

```
-32767      32769
```

Логічний тип. Значеннями логічного типу може бути одна із заздалегідь оголошених констант FALSE (хиба) або TRUE (істина). Для них справедливі правила:

```
ord(False)= 0;
ord(True)= 1;
False < True;
succ(False)= True;
pred(True)= False.
```

Оскільки логічний тип відноситься до порядкових типів, його можна використовувати в операторі рахункового типу, наприклад:

```
var
        I: Boolean;
begin
        for I := False to True do ....
```

Теоретично для запису змінної типу Boolean вистачає 1 біта, але мінімальна одиниця пам'яті, що адресується, – 1 байт.

Символьний тип. Значенням символьного типу є безліч усіх символів ПК. Кожному символу приписується ціле число в діапазоні 0...255. Це число служить кодом внутрішнього представлення символу, його повертає функція **ORD.**

Для кодування використовується код **ASCII** (American Standard Code for Information Interchange – американський стандартний код для обміну інформацією). Це 7-бітовий код, тобто за його допомогою можна закодувати лише 128 символів в діапазоні від 0 до 127. У той же час у 8-бітовому байті, відведеному для зберігання символу в TURBO-PASCAL, можна закодувати вдвічі більше символів в діапазоні від 0 до 256. Перша половина символів ПК з кодами 0...127 відповідає стандарту ASCII (табл. 5.3). Друга половина символів з кодами 128...255 не обмежена жорсткими рамками стандарту і може мінятися

за допомогою миши або клавіатури та підключати потрібну таблицю на ПК різних типів.

Таблиця 5.3. Перша половина символів ASCII

Dec	Hex	Char	Description	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	(null)	32	20		64	40	@	96	60	`
1	1	SOH	(start of heading)	33	21	!	65	41	A	97	61	a
2	2	STX	(start of text)	34	22	"	66	42	B	98	62	b
3	3	ETX	(end of text)	35	23	#	67	43	C	99	63	c
4	4	EOT	(end of transmission)	36	24	$	68	44	D	100	64	d
5	5	ENQ	(enquiry)	37	25	%	69	45	E	101	65	e
6	6	ACK	(acknowledge)	38	26	&	70	46	F	102	66	f
7	7	BEL	(bell)	39	27	'	71	47	G	103	67	g
8	8	BS	(backspace)	40	28	(	72	48	H	104	68	h
9	9	TAB	(horizontal tab)	41	29	)	73	49	I	105	69	i
10	A	LF	(NL line feed, new line)	42	2A	*	74	4A	J	106	6A	j
11	B	VT	(vertical tab)	43	2B	+	75	4B	K	107	6B	k
12	C	FF	(NP form feed, new page)	44	2C	,	76	4C	L	108	6C	l
13	D	CR	(carriage return)	45	2D	-	77	4D	M	109	6D	m
14	E	SO	(shift out)	46	2E	.	78	4E	N	110	6E	n
15	F	SI	(shift in)	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	(data link escape)	48	30	0	80	50	P	112	70	p
17	11	DC1	(device control 1)	49	31	1	81	51	Q	113	71	q
18	12	DC2	(device control 2)	50	32	2	82	52	R	114	72	r
19	13	DC3	(device control 3)	51	33	3	83	53	S	115	73	s
20	14	DC4	(device control 4)	52	34	4	84	54	T	116	74	t
21	15	NAK	(negative acknowledge)	53	35	5	85	55	U	117	75	u
22	16	SYN	(synchronous idle)	54	36	6	86	56	V	118	76	v
23	17	ETB	(end of trans. block)	55	37	7	87	57	W	119	77	w
24	18	CAN	(cancel)	56	38	8	88	58	X	120	78	x
25	19	EM	(end of medium)	57	39	9	89	59	Y	121	79	y
26	1A	SUB	(substitute)	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	(escape)	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	(file separator)	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	(group separator)	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	(record separator)	62	3E	>	94	5E	^	126	7E	~
31	1F	US	(unit separator)	63	3F	?	95	5F	_	127	7F	DEL

Символи з кодами 0...31 належать до службових кодів. Якщо ці коди використовуються в символьному тексті програми, вони вважаються пропусками. При використанні їх в операціях введення-виведення вони можуть мати таке самостійне значення:

Символ	Код	Значення
EL	7	Дзвінок; виведення на екран цього символу супроводжується звуковим сигналом
HT	9	Горизонтальна табуляція; при виведенні на екран зміщує курсор в позицію, кратну 8, плюс 1 (9, 17, 25 і так далі)
LF	10	Перехід рядка; при виведенні його на екран всі подальші символи виводитимуться, починаючи з тієї ж позиції, але на наступному рядку
VT	11	Вертикальна табуляція; при виведенні на екран замінюється спеціальним знаком
FF	12	Прогін сторінки; при виведенні на принтер формує сторінку, при виведенні на екран замінюється спеціальним знаком
CR	13	Повернення каретки; вводиться натисненням на клавішу Enter (при введенні за допомогою READ або READLN означає команду «Введення» і в буфер введення не поміщається; при виводі означає команду «Продовжити виведення з початку поточного рядка»)
SUB	26	Кінець файлу; вводиться з клавіатури натисненням CTRL-Z; при виведенні замінюється спеціальним знаком
SSC	27	Кінець роботи; вводиться з клавіатури натисненням на клавішу ESC; при виведенні замінюється спеціальним знаком

До типу **CHAR** застосовні операції відношення, а також вбудовані функції:

CHR(B) – функція типу CHAR; перетворить вираз B типу BYTE в символ і поверне його своїм значенням;

UPCASE(CH) – функція типу CHAR; повертає прописну букву, якщо CH – строкова латинська буква, інакше повертає сам символ CH, наприклад:

```
var
c1, c2: Char;
begin
c1 := UpCase('s');
c2 := UpCase ('Ф') ;
WriteLn(c1',  ',c2)
end.
```

Оскільки функція UPCASE не обробляє кирилицю, в результаті прогону цієї програми на екран буде видано **S Ф**

Приклад 5.1.

```
{ Програма виводить код введеного символу }
var      sim: char;      { символ }
  code: integer; { код символу }
begin
  writeln('Введіть символ і натисніть <Enter>.');
  writeln('Для завершення роботи програми введіть крапку.');
  repeat
  write('->'); readln(sim); code:=Ord(sim);
    writeln('Символ: ',sim, ' Код: ',code);
  until sim = '.';
```

end.

Тип, що перераховується. Припустимо, що нам потрібна змінна для зберігання дня тижня. У цьому випадку можна скористатися цілим типом (наприклад byte) та зберігати дні тижня у вигляді чисел 1, 2, ... 7, але це буде не дуже зручно. Для наочності надається зручніший варіант, а саме тип, що перераховується. Тип, що перераховується, задається перерахуванням тих значень, які він може приймати. Кожне значення іменується деяким ідентифікатором та розташовується у списку, який оточується круглими дужками, наприклад:

type Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

Після цього можна завести змінну цього типу (**var** day: Days;) та використовувати її. Нижче наведені приклади використання:

Day := Wed;

if day > Fri **then** writeln('Сьогодні останній день тижня');

if day = Mon **then** writeln('Почався робочий тиждень');

Як ви вже помітили значення даних типу, що перераховується, можна порівнювати; при цьому меншим вважається те, яке оголошене раніше (лівіше) у визначенні типу.

Для змінних типів, що перераховуються, можливе застосування функцій succ та pred, наприклад, succ(Wed) дає Thu, Pred(Sun) дає Sat. Якщо спробувати написати Pred(Mon) або Succ(Sun), то вже на етапі перевірки програми компілятором станеться помилка.

Зберігання значень типу, що перераховужться, влаштоване всередині досить просто: зберігаються цілі числа від 0 до n, у нашому випадку n=6. Максимальна потужність типу, що перераховується, складає 65536 значень, тому фактично цей тип задає деяку підмножину цілого типу WORD і може розглядатися як компактне оголошення групи цілочисельних констант із значеннями 0, 1 і т.і.

Існує функція Ord, яка дозволяє отримати те число, у вигляді якого зберігається будь-яке значення типу, що перераховується, наприклад Ord(Wed) дає 2. При необхідності можна отримати значення типу, що перераховується, за його чисельним уявленням, наприклад, Days(1) є Tue. Після всього наведеного можна відмітити, що при порівнянні величин типу, що перераховується, насправді порівнюються їх порядкові номери (Ord).

Інтервальний тип. Нехай у деякій змінній потрібно зберігати поточне число, наприклад, номер дня в місяці. У TURBO-PASCAL можна задати тип DaysInMonth = 1..31;. Змінні та константи цього типу можуть набувати тільки таких значень. Якщо спробувати задати будь-що інше, то компілятор видасть помилку. В якості границь можуть вживатися і від'ємні числа, наприклад Temperature = -50..50;

Як **базовий** тип (тобто тип, з якого обирається діапазон значень) можуть використовуватися майже всі **порядкові** типи, тобто ті, які зберігаються у вигляді цілих чисел. До порядкових типів належать: всі цілі типи (byte, integer, і т. п.), char, boolean, типи, що перераховуються.

Не можна як базовий тип використовувати дійсні числа, таке оголошення призведе до помилки:

Type Wrong = -1.26..1.25;

При визначенні **типу-діапазону** (або **інтервального типу**) потрібно керуватися наступними правилами:

- два символи «..» розглядаються як один символ-розподільник, тому між ними неприпустимими є пробіли;
- ліва межа діапазону не повинна перевищувати його праву межу.

Тип-діапазон успадковує усі властивості свого базового типу, але з обмеженнями, які пов'язані з його меншою потужністю. Зокрема, якщо визначена змінна

```
type   days = (mo,tu,we,th,fr,sa,su);
       WeekEnd = sa .. su;
var    w : WeekEnd;
begin
.....
   w := sa;
.....
end.
```

то **ORD(W)** поверне значення 6, тоді як **PRED(W)** призведе до помилки.

У стандартну бібліотеку включено дві функції, які підтримують роботу з типами-діапазонами:

- **HIGH(X)** – повертає максимальне значення типу-діапазону, до якого належить змінна X;
- **LOW(X)** -повертає мінімальне значення типу-діапазону.

Наступна коротка програма виведе на екран рядок

-32768...32767

```
var    k: Integer;
begin
       WriteLn(Low(k) , '..',High(k))
end.
```

Приклад 5.2.

```
{ Ця програма визначає, чи є рік високосним за Григоріанським календарем }
program Leap_Year
Var          Year: 1582..9999;
       Bool: Boolean;
Begin
  WriteLn('Який рік Ви хочете перевірити?: ');
  Read(Year);
  Bool := False;
```

```
  If (Year mod 4 = 0) and (Year mod 100 <> 0) then {перевірка для років, кратних 4}
        Bool := True;
  If Year mod 400 = 0 then          {перевірка для століть, кратних 400}
       Bool := True;
  If Bool then
       WriteLn(Year:4', – високосний рік')
  else
       WriteLn(Year:4', – не високосний рік');
   ReadLn
End.
```

5.1.2. Дійсні типи

На відміну від порядкових типів, значення яких завжди співставляються з рядком цілих чисел і, отже, представляються в ПК абсолютно точно, значення дійсних типів визначають довільне число лише з деякою кінцевою точністю, яка залежить від внутрішнього формату дійсного числа.

Дійсні числа в Pascal представляються в одній з двох форм, які називаються записом числа з фіксованою крапкою та записом числа з плаваючою крапкою. Перша з них – це запис числа у вигляді цілої та дробової частин, які розділені крапкою, наприклад: -3.15, 0.1, +23.0126. Друга форма – з плаваючою крапкою – це запис числа з мантисою та десятковим порядком, які розділені латинською буквою E (e). Такий запис означає, що мантиса (яка може бути цілим числом або дійсним числом у формі з фіксованою крапкою) перемножується на 10 у ступені, що задається порядком (завжди ціле число), наприклад -18.7E+3, 2.123E4, 2.34E-2, 6E-1.

У мові програмування Pascal забороняється запис дійсних чисел у вигляді .5 або 6. Їх необхідно записувати як 0.5 і 6.0 відповідно. Якщо у записі числа міститься крапка, то принаймні одна цифра повинна передувати їй та/або слідувати за нею.

Як видно з табл. 5.4, дійсне число в TURBO-PASCAL займає від 4 до 10 суміжних байт і має наступну структуру у пам'яті ПК:

S	E	M

Таблиця 5.4. Дійсні числа у Pascal

Доввжина, байт	Назва	Кількість значущих цифр	Діапазон значень
4	Single	7…8	$1.5*10^{(-45)}..3.4*10^{38}$
6	Real	11. . .12	$2.9*10^{(-39)}..1.7*10^{38}$
8	Double	16. . .16	$4.9*10^{(-324)}..1.7*10^{308}$
10	extended	19. . .20	3.1*10^(-4944)..1.2*10^4932
8	comp	19. . .20	-2*E+63. . .+2*E-63-1

Тут s – знаковий розряд числа; e – експоненціальна частина містить двійковий порядок з деяким зміщенням, яке вказує на нульовий порядок; m – мантиса числа. Мантиса m має довжину від 23 (для SINGLE) до 64 (для EXTENDED) двійкових розрядів, що і забезпечує точність 7...8 для SINGLE та 19...20 для EXTENDED десяткових цифр. Десяткова крапка мається на увазі перед лівим (старшим) розрядом мантиси, але при діях з числом її положення зсувається ліворуч або правопуч відповідно до двійкового порядку числа, що зберігається в експоненціальній частині, тому дії над дійсними числами називають арифметикою з плаваючою крапкою.

Як бачимо, TURBO-PASCAL характеризується багатою гаммою дійсних типів, проте доступ до типів SINGLE (читається “сингл”), DOUBLE (читається “дабл”) та EXTENDED (читається “икс’тендед”) можливий тільки при особливих режимах компіляції. Річ у тім, що ці типи розраховані на апаратну підтримку арифметики з плаваючою крапкою, і для їхнього ефективного використання до складу ПК повинен входити арифметичний співпроцесор. Компілятор TURBO-PASCAL дозволяє створювати програми, які працюють на будь-яких ПК (із співпроцесором або без нього) і використовують будь-які дійсні типи.

Відзначимо, що арифметичний співпроцесор завжди обробляє числа у форматі EXTENDED, а три інших дійсних типи у цьому випадку одержуються простим усіканням результатів до потрібних розмірів і застосовуються в основному для економії пам’яті.

Слід врахувати, що тип REAL оптимізований для роботи без співпроцесора. Якщо Ваш ПК оснащений співпроцесором, використання типу REAL призведе до додаткових витрат часу на перетворення REAL до EXTENDED. Тому ніколи не використовуйте REAL на ПК із співпроцесором, оскільки додаткові витрати часу на перетворення типів можуть звести нанівець усі переваги співпроцесора. При розробці програм, критичних до часу обрахунку, слід замінювати його типами SINGLE або DOUBLE: у порівнянні з типом REAL швидкість обчислень на комп'ютерах із співпроцесором в цьому випадку збільшується у 2...3 рази. Якщо в ПК немає арифметичного співпроцесора, швидкість обробки даних усіх дійсних типів приблизно однакова.

Особливе положення у TURBO-PASCAL займає тип COMP (від англ. Compound – составний), який трактується як дійсне число без експоненціальної та дробової частин. Фактично, COMP – це «велике» ціле число із знаком, що зберігає 19...20 значущих десяткових цифр (у внутрішньому уявленні COMP займає 8 суміжних байтів). У той же час у виразах COMP повністю сумісний з будь-якими іншими дійсними типами: над ним визначені усі дійсні операції, він може використовуватися як аргумент математичних функцій тощо. Найвідповіднішою областю застосування типу COMP є бухгалтерські розрахунки: грошові суми виражаються в копійках або центах і дії над ними зводяться до операцій з достатньо довгими цілими числами.

5.1.3. Стандартні вбудовані математичні функції

Для роботи з дійсними даними можуть використовуватися вбудовані математичні функції, які представлені у табл. 5.5. У цій таблиці REAL означає будь-який дійсний тип, INTEGER – будь-який цілий тип.

Таблиця 5.5. Вбудовані математичні функції

Стандартні математичні функції TURBO-PASCAL			
Звернення	Тип параметра	Тип результату	Ркзультат
abs (x)	Real, Integer	Тип аргументу	Модуль аргументу (|x|)
arctan (x)	Real	Real	Арктангенс (значення в радіанах)
cos (x)	То же	Real	Косинус, кут в радіанах
exp (x)	Real	Real	Експонента
frac (x)	Real	Real	Дробова частина числа
trunc(x)	Real	Integer	Повертає цілу частина значення x
round(x)	Real	Integer	Округлює x до найближчого цілого
int(x)	Real	Real	Ціла частина числа
ln(x)	Real	Real	Логарифм натуральний
Pi	-	Real	Функція поверне π= 3.141592653...
Random	-	Real	Псевдовипадкове число, рівномірно розподілене в діапазоні 0...[1]
Random(x)	Integer	Integer	Псевдовипадкове ціле число, рівномірно розподілене в діапазоні 0...(x-1)
Randomize	-	-	Ініціація генератора псевдовипадкових чисел
sin(x)	Real	Real	Синус, кут в радіанах
sqr (x)	Real	Real	Квадрат аргументу
sqrt (x)	Real	Real	Корінь квадратний
Sizeof(x)	Любий	Integer	Повертає кількість байтів, які відводяться під змінну

Тут *x* означає будь-яку відповідну за типом змінну, або результат обчислення виразу відповідного типу, або відповідний за типом результат, який обчислюється іншою стандартною функцією. Функція Pi не має аргументів і повертає число π.

Аргументи тригонометричних функцій sin і cos задаються у радіанах. Для перетворення значення кута з радіанної міри у градусну необхідно помножити

величину кута на число 180/π. Якщо потрібно визначити, чи ділиться дійсне число "A" без остачі на число "B", то можна застосовувати функції Frac(x); і Int(x):

```
x:= A/B;
If Frac(x) = 0  then
      writeln('Число "A" ділиться без остачі на число "B"');
If Int(x) = x   then
      writeln('Число "A" ділиться без остачі на число "B"');
```

5.1.4. Приклади арифметичних виразів із вбудованими функціями

1. Піднести *x* у п'яту ступінь.

x*x*x*x*x або sqr(x)*sqr(x)*x, або sqr(sqr(x))*x, останнє показує, що результати одних функцій можуть бути аргументами інших – це називають **вкладенням функцій**. Зрозуміло, що тип результату, який повертається вкладеною функцією, має бути відповідним для аргументу зовнішньої функції.

2. Піднести величину *a* у довільну ступінь *x*.

Оскільки в Pascal немає функції піднесення до довільного ступеня, скористаємося формулою $\mathbf{a^x=e^{x*\ln a}}$

A := 2.5; x := 0.25; a^x := exp(x*ln(a));

зверніть увагу, що всі дужки у виразі мають бути парними. Або:

$\sqrt[3]{x} = x^{1/3}$ = exp(1/3∗ln(x)).

3. Обчислити $\sin^2 x$ => sqr(sin(x)). Порівняйте з $\sin x^2$ => sin(sqr(x)).

Не можна писати sin*x або sin x, після імені функції може слідувати лише аргумент у круглих дужках.

4. Обчислити k=tg(t). Оскільки функції тангенса у Pascal немає, пишемо k:=sin(t)/cos(t);
5. У записі виразів не можна пропускати знак *, як це часто робиться в математиці.

   b^2-4ac -> sqr(b)-4*a*c.

5.1.5. Константи

Ще раз детальніше познайомимося з константами в Pascal.

Константою називають величину, значення якої не змінюється у процесі виконання програми.

Числові константи використовуються для запису чисел. Розрізняють такі їх види.

1. Цілочисельні (цілі) константи: записуються із знаком + або -, або без знаку, за звичайними арифметичними правилами: -10 +5 6.

2. Дійсні числа можуть записуватися в одній з двох форм:

1) **звичайний запис:** 2.5, -3.14 2. - зверніть увагу, що ціла частина відділяється від дробу символом **крапки**;
2) **експоненціальна** ("наукова") форма: у цьому записі дійсне число представляється у вигляді $m*10^p$, де m – **мантиса** (лат. – надбавка) або основа числа, $0.1 \le |m| < 1$, p – **порядок** числа, це цілочисельна константа.

По іншому порядок числа називають ***експонентою***, а мантиссою називають дробову частину десяткового логарифма.

Справді, будь-яке дійсне число можна представити в експоненціальній формі: $-153.5 = -0.1535*10^3$, $99.005 = 0.99005*10^2$.

В усіх IBM-сумісних комп'ютерах дійсні числа зберігаються як сукупність мантиси та порядку, що дозволяє спростити операції над ними при використанні спеціальної арифметики, яка дозволяє окремо обробляти мантиси та порядки. Для програмного запису числа в експоненціальній формі замість "помножити на 10 в ступені" використовується позначення E або e (латинська):-

153.5 → $-0.1535*10^3$, -0.1535E3 або -1.535E02

99.005 → $0.99005*10^2$, 0.99005E+2 або 9.9005e+01

Без вживання спеціальних заходів, програма на Pascal виводитиме на екран та принтер дійсні числа саме у такій формі. Крім того, така форма зручна для запису дуже маленьких та дуже великих чисел:

10^{30}, 1e30, -10^{20}, -1E20, 10^{-30}, 1E-30.

Оскільки розмір пам'яті, що відводиться під мантису та порядок, обмежений, то **дійсні числа завжди представляються у пам'яті комп'ютера з деякою похибкою (погрішністю)**. Наприклад, простий дійсний дріб 2/3 дає у десятковому вигляді 0,666666... і, незалежно від розміру пам'яті, що виділяється для зберігання числа, неможливо зберігати *всі* його знаки в дробовій частині. Однією з типових проблем програмування є обчислення можливих похибок при роботі з дійсними числами.

Окрім числових констант існують інші їх види:

1) **логічні** константи призначені для перевірки істинності або помилковості деяких умов у програмі і можуть приймати лише одне **з двох значень**: службове слово **true** позначає істину, а **false** – хибу;
2) **символьні** константи можуть набувати значення будь-якого друкованого символу і записуються як символ, оточений *апострофами* ('одинарні лапки'):
   'Y' 'я' ' '

В останньому випадку значення символьної константи дорівнює символу пробілу. Якщо потрібно записати сам символ апострофа як символьну константу, всередині зовнішніх апострофів він подвоюється: ''''

До символьних також відносяться константи виду #X, де X – числове значення від 0 до 255 включно, що є десятковим *ASCII-кодом* символу. Таблиці ASCII-кодів, що використовуються операційними системами DOS та Windows, наведені у табл. 6.3. Наприклад, значення #65 відповідатиме коду символу 'A' латиницею.

3. Строкові константи – це будь-які послідовності символів, оточених апострофами. Як правило, строкові константи служать для запису запрошень до введення даних, що обробляються програмою, виведення діагностичних повідомлень і т.і.:

‘Введіть значення X:’, ‘відповідь=‘.

Якщо в строковій константі необхідно записати сам символ апострофа, це робиться так само, як для символьних констант.

4. Іменовані константи (в літературі зустрічається ще одне тлумачення – **типізовані константи**) описуються в розділі описів програми оператором такого вигляду:

const Імя1 = Значення1;
Імя2 = Значення2;
. . .
Імяn=Значеннян;

Тут ключове слово const показує початок розділу описів іменованих констант. Зрозуміло, що часто зручніше звертатися до константи по імені, ніж кожного разу переписувати її числове або строкове значення. Приклад розділу констант:

```
Const   e = 2.7182818285;
Lang = ‘Turbo Pascal 7.1’;
```

Тут описана числова константа ***e,*** яка є значенням основи натурального логарифму, і строкова константа з іменем lang, що містить рядок ‘Turbo Pascal 7.1’.

Кожне ім’я, що дається програмістом, має бути **унікальним** в межах однієї програми. Якщо ми включимо цей розділ у свою програму, то вже не зможемо створити в ній інших об’єктів з іменами ***e*** та *lang*.

5.1.6. Старшинство типів виразу

Тип виразу визначається старшим з типів операндів, що входять в нього (тобто стандартних функцій, змінних, констант). Старшинство типів ми можемо визначити за таблицею 5.6, у першому рядку якої – наймолодший тип. Приклад:

```
var       i,j: integer;
f:  real;
. . .
i+4*j     –  цілий тип виразу, можна записати результат у цілу змінну,
f+i*0.5 –  дійсний, результат пишеться в дійсну змінну.
```

Операція ділення / у Pascal **завжди** дає дійсне число. Для ділення цілих чисел з цілим результатом (залишок відкидається) використовуйте **div**, для взяття залишку від ділення двох цілих – **mod**.

Оскільки будь-які дані у пам’яті комп’ютера зберігаються у числовій формі та двійковій системі числення, окрім імені змінній обов’язково слід

присвоїти такий самий **тип**, який визначає **діапазон значень**, що приймаються змінною, та **спосіб її обробки** машиною. Пояснимо сказане на прикладі.

Латинська велика літера ‘А’ має десятковий код 65, або 01000001 у двійковому поданні. Без додаткової інформації про *тип* даних, що зберігаються в деякому елементі пам’яті, комп’ютеру було б неможливо вирішити, що саме вдають із себе ці дані – число 65, код символу ‘А’ або щось ще інше.

У будь-якій мові програмування, у тому числі і у Pascal, існує стандартний набір типів, до яких може бути віднесена та чи інша сукупність елементів пам’яті. Інформацію про тип даних Pascal зручно звести у таблицю. Рядки цієї таблиці впорядкуємо за старшинством **типів**, від "наймолодшого", який вимагає найменше число байт для подання, яке, відповідно, представляє найменший діапазон можливих значень, до "найстаршого", що представляє найбільший діапазон значень. У таблиці 5.6 представлені не всі можливі, а лише основні типи даних Pascal.

Таблиця 5.6. Таблиця старшинства типів даних

Ключове слово Pascal	**Назва і опис типу**	**Об’єм пам’яті, байт**	**Діапазон можливих значень**
Boolean	Логічний: зберігає одну логічну змінну	1	true та false
Char	Символьний: зберігає код одного символу з набору ASCII-кодов	1	від 0 до 255 включно (28=256)
Integer	Цілочисельний	2	$\pm 2^{15}$
Word	Цілочисельний без знаку	2	2^{16} – діапазон удвічі більше, оскільки 16-й біт не зайнятий під знак числа
Longint	Довге ціле: для представлення великих цілочисельних значень	4	$\pm 2^{31}$
Real	Дійсне число з точністю вистави до 11-12 знаку в дробовій частині	6	~ $2.9*10^{-39}$ – $1.7*10^{38}$
Double	Дійсне число з точністю вистави до 15-16 знаку в дробовій частині	8	~ $5*10^{-324}$ – 1.7*10308
String	Послідовність символів типу Char завдовжки від 1 до 255	2-256 (дані рядки + 1 байт для зберігання її довжини)	Будь-які рядки тексту, що складаються з друкованих символів

Цілочисельні та символьні типи узагальнено називають **порядковими**, підкреслюючи цим, що дані типи мають кінцевий набір значень, які можуть бути впорядковані або перераховані. Нагадаємо, що дійсні значення зберігаються у пам’яті комп’ютера інакше, ніж цілі – а саме, як сукупність знаку, мантиси та порядку.

Зрозуміло, що задача правильного вибору типів даних цілком покладається на програміста. Наприклад, якщо деякий лічильник у Вашій програмі може набувати цілочисельних значень від 1 до 100000, неправильно було б описувати його як змінну типу Integer – адже 2^{15}=32768, і при досягненні лічильником цієї величини станеться *скидання* його значення, яке дорівнюватиме -32768. Розумним у даному випадку був би опис лічильника як змінної типу Longint.

5.2. Структуровані типи

Будь-який із структурованих типів (а у TURBO-PASCAL їх чотири: масиви, записи, множини та файли) характеризується множинністю створюючих цей тип елементів, тобто змінна або константа структурованого типу завжди має декілька компонентів. Кожен компонент, у свою чергу, може належати структурованому типу, що дозволяє говорити про можливу вкладеність типів. У TURBO-PASCAL припускається довільна глибина вкладеності типів, проте сумарна довжина будь-якого з них у внутрішньому поданні не повинна перевищувати 5520 байт.

З метою сумісності із стандартним Pascal у TURBO-PASCAL дозволяється перед описом структурованого типу ставити зарезервоване слово PACKED, приписуюче компілятору, по можливості, економити пам’ять, яка відводиться під об’єкти структурованого типу; але компілятор фактично ігнорує цю вказівку: «пакування» даних у TURBO-PASCAL здійснюється автоматично скрізь, де це тільки можливо.

5.2.1. Одновимірні та багатовимірні масиви

Масивом називається скінченна послідовність змінних одного типу, які мають однакове ім'я та різняться порядковим номером.

Необхідність у використанні масивів при вирішенні завдань виникає кожного разу, коли доводиться мати справу з великим, але кінцевим числом однотипних (однакових) даних. У цьому випадку всі дані можна об’єднати за певним порядком проходження у єдину структуру, якій дається одне ім’я. Це дозволяє у компактному та одноманітному вигляді записати дії над окремими даними як над елементами масиву (використовуючи узагальнене позначення елементу масиву).

Особливістю алгоритмів обробки масивів є те, що всі або більшість (або якась кількість) елементів обробляються однаково (за одним алгоритмом). Тому, розв’язуючи задачу, доводиться в тому чи іншому порядку проглядати (перебирати за допомогою циклу) елементи масивів. Елементи масивів можна легко впорядкувати та забезпечити доступ до будь-якого з них простою вказівкою його імені та порядкового номера (номера комірки), який називається ***індексом*.** Щоб здійснити перебір, треба визначити правило зміни індексу елементів (для багатовимірних масивів – індексів).

Одновимірні масиви. Масив називається одновимірним, якщо для задання місцеположення елемента в масиві необхідно вказати значення лише одного індексу.

Масиви у TURBO-PASCAL багато у чому схожі з аналогічними типами даних в інших мовах програмування. Відмінна особливість масивів полягає у тому, що всі їх компоненти – сутність дані одного типу (можливо, структурованого).

Опис типу одновимірного масиву (вектора) задається таким чином:

<ім'я типу> = ARRAY [<сп.инд.типов>] OF <тип>

Тут

<ім'я типу> – правильний ідентифікатор;

ARRAY, OF – зарезервовані слова (масив, читається "э'рэй");

<сп.інд.типов> – список з одного або декількох індексних типів, які розділені комами; квадратні дужки, що оточують список, – вимога синтаксису;

<тип> – будь-який тип TURBO-PASCAL.

Зазвичай при визначенні індексів використовується тип-діапазон, в якому задаються межі зміни індексів. Межі індексів завжди вказуються через два символи «.». Як звернутися до елементів цього масиву? Для цього необхідно вказати індекс. Наприклад, M[3], M[5], M[i], M[i+j].

Як індексні типи у TURBO-PASCAL можна використовувати будь-які порядкові типи, окрім LONGINT та типів-діапазонів з базовим типом LONGINT.

Оскільки тип «масив» відноситься до стандартних типів мови Pascal, то визначити змінну як масив можна і безпосередньо при описі цією змінною, без попереднього опису типу масиву, наприклад: **var a,b : array [1..10] of Real**;

Межі зміни індексів повинні бути сталими величинами, а не змінними, інакше невідомо буде, скільки місця необхідно відвести в пам'яті під такий масив. Дуже зручно задавати розмір масиву за допомогою констант. Надалі, якщо знадобиться поміняти розмір масиву, достатньо це зробити тільки в розділі const, не чіпаючи всю програму. Наприклад, можна розмір масиву задати таким чином:

```
const Size=50;
var A: array[1..Size] of real;
```

Знайти найбільше число серед елементів масиву можна за допомогою такої програми:

```
program FindMaximumInArray;
var a: array[1..10] of real;
    i,max: integer;
begin
  for i:=1 to 10 do begin
        write('Введіть елемент за номером ',i,' -> ');
```

```
        readln(a[i]);
    end;
    max:=a[1];
    for i:=2 to 10 do
        if a[i]>max then max:=a[i];
    writeln('Максимум дорівнює ',max);
    readln;
end.
```

У якості типу елементів масиву можна використовувати усі типи, які відомі нам на даний момент (до них належать усі числові та символьні типи).

Нумерувати елементи масивів можна не тільки від одиниці, але й від будь-якого цілого числа. Взагалі індексом масиву може бути будь-який порядковий тип, тобто такий, який у пам'яті машини представляється цілим числом. Єдине обмеження полягає у тому, що розмір масиву не повинен перевищувати 64 Кб. Розглянемо деякі приклади оголошення масивів.

```
var   Numbers: array [0..1000] of integer;
      Names: array [1..10] of string;
      CountAll: array[char] of integer;
      Count: array['a'..'z'] of integer;
```

У наступному прикладі показано для чого може знадобитися останній тип.

```
{ Підрахунок кількості різних літер у рядку.}

program CountLetters;
var   s: string;
      count: array['a'..'z'] of byte;
      ch: char;
      i: byte;
begin
      write('Введіть рядок > ');
      readln(s);
      for i:=1 to length(s) do
          if (s[i] >= 'a') and(s[i] <= 'z') then inc(count[s[i]]);
      writeln('Кількість різних букв в рядку: ');
      for ch := 'a' to 'z' do
          if count[ch] <> 0 then
              writeln(ch,': ',count[ch]);
      readln;
end.
```

Глибина вкладеності структурованих типів взагалі та, зокрема, масивів – довільна, тому кількість елементів у списку індексних типів (розмірність масиву) не обмежена, проте сумарна довжина внутрішнього представлення будь-якого масиву, як вже було зазначено, не може бути більшою 65520 байт.

Багатовимірні масиви. При необхідності можна нумерувати масиви не одним індексом, а двома та більше. Багато задач пов'язано з обробкою багатовимірних масивів даних. Найбільш поширені при цьому *двовимірні масиви*. У математиці двовимірні масиви представляються матрицею, тобто прямокутною таблицею:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \text{K} & a_{1,m} \\ \text{K} & \text{K} & \text{K} & \text{K} \\ a_{n,1} & a_{n,2} & \text{K} & a_{n,m} \end{pmatrix}$$

або, у скороченому записі

$$A = \{a_{i,j}\}_{n \times m},$$

де n – кількість рядків матриці, m – кількість стовпчиків, i, j – індекси (номери) поточного рядка та стовпчика, на перетині яких знаходиться елемент $a_{i,j}$.

Як і інші об'єкти даних, матриця описується у розділі var. Від вектора її опис відрізняється тим, що у квадратних дужках перераховуються два оператори діапазону, які вказують, відповідно, нумерацію рядків та стовпчиків матриці:

var <ІмяМатриці>: array [n1..n2, m1..m2] of Тип;

Тут n1..n2 – діапазон значень номера рядка, n1 і n2 – цілочисельні константи;

m1..m2 – діапазон значень номера стовпчика, значення m1 і m2 також цілочисельні.

Як і вектори, матриці можуть бути утворені з елементів будь-якого типу даних, що існують у мові.

Під матрицю виділяється область пам'яті розмірністю n*m*k байт, де k – розмірність у байтах одного елементу. В оперативній пам'яті матриця зберігається по рядках.

Наприклад, для матриці A, яка описана оператором виду

var A:array [1..5,1..4] of real; виділяється 20 елементів пам'яті по 6 байт, причому у наступній за елементом A[1,5] комірці зберігається значення елементу A[2,1].

Звернення до окремих елементів матриці здійснюється за допомогою змінної з двома індексами, наприклад:

$a_{i,j}$	a[i,j]
$a_{2,1}$	a[2,1]
$a_{2n,k}$	a[2*n,k]

Перший індекс, як і в математиці, завжди вказує на номер рядка, а другий – на номер стовпчика.

Оскільки адресація пам'яті у будь-якому випадку лінійна, слід розуміти матрицю як зручний для програміста структурний тип даних. В окремих випадках використання матриці може бути замінене використанням вектора з

тією самою кількістю елементів: отже, матриці A[n,m] завжди може бути поставлений у відповідність вектор b розмірністю n*m, а звернення до елементу A[i,j] при нумерації рядків та стовпчиків з одиниці може бути замінене на звернення до елементу b[(i-1)*m+(j-1)].

Подібно до того, як будь-яка послідовна обробка вектора виконується у циклі for, обробка матриці виконується у подвійному циклі for:

```
for i:=1 to n do
    for j:=1 to m do
  {Обробка елементу  A[i,j]  по рядках}
```

Згідно з правилом виконання кратних циклів змінна j міняється у цьому подвійному циклі швидше за i. Таким чином, обробка усіх елементів матриці буде виконана по рядках. Для послідовної обробки всіх елементів матриці по стовпчиках досить поміняти місцями цикли по i та j:

```
for j:=1 to m do
    for i:=1 to n do
  {Обробка елементу A[i,j]  по стовпчиках}
```

Наведемо приклади використання подвійного циклу for для введення та виведення елементів матриці. Нехай матриця C розмірністю 4*2 (як ми пам'ятаємо, це означає, що в матриці 4 *рядки* та 2 *стовчика*) описана оператором

```
var c:array [1..4,1..2] of real;
```

У програмі передбачено також 2 цілочисельних лічильника для рядків та стовпчиків матриці:

```
var i,j:integer;
```

У цьому випадку типове введення матриці з клавіатури могло б виглядати таким чином:

```
writeln ('Введіть матрицю C розмірністю 4*2');
for i:=1 to 4 do
    for j:=1 to 2 do read (c[i,j]);
```

Інколи зручніше друкувати окреме запрошення до введення кожного елементу:

```
writeln ('Введіть матрицю C розмірністю 4*2');
  for i:=1 to 4 do
      for j:=1 to 2 do begin
          write ('C[',i,',',j']=');
          readln (c[i,j]);
      end;
```

Наприклад, як запрошення до введення елементу C[1,1] на екрані буде надруковано:

```
C[1,1]=
```

Оператор readln використовується, оскільки елементи вводяться поодинці.

Як і для векторів, можливі альтернативні способи введення – опис матриці у розділі констант та формування її елементів з випадкових чисел. Наведемо приклад опису матриці констант розмірністю 2*3:

const d:array [1..2,1..3] of integer = ((1,2,3), (4,5,6));

Як видно з прикладу, для опису матриці констант створюється список її рядків, кожен рядок, у свою чергу, є списком значень елементів.

Приклади описів багатовимірних масивів:

```
Var     Matrix:  array[1..4,1..3] of  real;
        Cube3D:  array[1..5,1..5,1..5] of  integer;
```

У пам'яті ПК елементи масиву слідують один за одним так, що при переході від молодших адрес до старших найшвидше міняється найправіший індекс масиву. Якщо, наприклад

```
var     a : array[1. .2,1. .2] of Byte;
begin
a [1,1]:=1;
a [2,1]:=2;
a [l, 2]:=3;
a [2,2]:=4;
end.
```

то у пам'яті послідовно один за одним будуть розташовані байти із значеннями 1,3,2,4.

У TURBO-PASCAL можна **одним оператором присвоєння передати всі елементи одного масиву іншому масиву того ж типу**, наприклад:

```
var     a, b: array [1..5] of Single;
begin
a := b;
end.
```

Після цього присвоєння усі п'ять елементів масиву А набудуть тих же значень, що й у масиві В. Але над масивами не визначені операції відношення. Не можна, наприклад, записати if a = b then ...

Порівняти два масиви можна тільки поелементно, наприклад:

```
var     a,b: array [1..5] of Single;
eq:  Boolean;
i:    Byte;
begin
eq := True;
for i := 1 to 5 do
        if a[i] <> b[i] then
        eq := False;
        if eq then
.....
end.
```

Наведемо ще декілька програм на використання масивів та циклів.

Приклад 5.3.

{Обчислення полінома P(x) ступеня n за схемою Горнера}

Program Polinom;

```
Var
        A: array [0..30] of Real;
        S, x: Real;
i, n : Integer;
 Begin
   Writeln('Обчислення полінома P(x) ступеня n за схемою Горнера');
   Write('Введіть ступінь полінома n (< 30) = ');
   Readln(n);
   Write('Введіть значення x = ');
   Readln(x);
   { Введення у масив коефіцієнтів полінома: a0, a1 ., an }
   For i:=0 to n do
   Begin
     Writeln('Введіть значення коефіцієнтів:');
     Write('a',i,' = ');
     Readln(A[i]);
   End;
   S := A[n];
   For i:=1 to n do  begin
       S := S*x + A[n-i];
   End;
   Writeln('Значення P(x)= ',S);
   ReadKey;
 end.
```

Приклад 5.4.

{ Програма зчитує послідовність бюлетенів, номер кандидата вказаний числом }

{ Результати голосування роздруковуються }

```
program Ballot;
Const   Stop = -1;          { Кінець введення }
        Candidat = 5;             { Число кандидатів }
Var     Votes: array[1..Candidat] of Integer;
        Count: Integer;
        Vvod : Integer;
Begin
  { Кожен починає з нульовим числом голосів }
  For Count := 1 to Candidat do
      Votes[Count]:= 0;
  WriteLn('Кожен бюлетень представлений номером кандидата між 1 та ',Candidat:1);
  WriteLn('Вводьте бюлетені, завершуючи список значенням ',Stop:2);
  Read(Vvod);
  While VVod <> Stop do
    Begin
      If (Vvod < 1) or (Vvod > Candidat) then
        WriteLn('бюлетень, закодований під N ',Vvod', не враховується')
```

```
        else
          Votes[Vvod]:= Votes[Vvod]+1;
        Read(Vvod)
      end;
  { Роздруківка результатів }
  WriteLn;                    {Порожній рядок}
  WriteLn('Результати голосування такі:');
  For Count := 1 to Candidat do
    WriteLn('Кандидат N',Count:2', отримав ',Votes[Count],' голосів');
  ReadLn;     {Для затримки результатів програми на екрані}
End.
```

Приклад 5.5.

```
{ Програма зчитує текст, що закінчується стоп-символом @, і визначає }
{ частоту появи букв }
program LetterFreq;
Var     Letter: array['A'..'Z'] of Integer;   {Частоти}
        Symbol: Char;                 {Вхідна літера}
        Index : Char;                 {Індекс циклу}
Begin
  { Читання тексту }
  Read(Symbol);
  While Symbol <> '@' do
    Begin
        If (Symbol >= 'A') and (Symbol <= 'Z') then
          Letter[Symbol]:= Letter[Symbol]+ 1;
  Read(Symbol)
    end;
   { Роздруківка результатів }
   WriteLn('Літера'',  Частота');
   For Index := 'A' to 'Z' do
if Letter[Index] <> 0 then
          WriteLn('  ',Index,Letter[Index]:9);
  ReadLn;
  ReadLn
End.
```

Приклад 5.6.

```
{ Дана послідовність з n дійсних чисел. }
{Впорядкувати її елементи за збільшенням методом сортування вибором }
{ Відшукується максимальний елемент та переноситься у кінець масиву; }
{ потім цей метод застосовується до всіх елементів, окрім останнього }
{(він вже знаходиться на своєму остаточному місці), і так далі }
Program SortCase;
Const n=10;
Type mas = Array[1..n] Of Real;
Var    x : Mas;
```

```
   i,j,m : Integer;
       r : Real;
Begin
   {введення елементів масиву}
   For i:=1 To n Do Readln(x[i]);
   For j:=n Downto 2 Do
      Begin {пошук m-номера max x[1..j]}
         m:=1;
         For i:=2 To j Do If x[i]>x[m] Then m:=i;
         {перестановка елементів x[m] і x[j]}
         r:=x[j]; x[j]:=x[m]; x[m]:=r
      End;
   {виведення елементів впорядкованого масиву}
   For i:=1 To n Do Writeln(x[i])
End.
```

5.2.2. Записи

Запис – це структура даних, що складається з фіксованої кількості компонентів, які називаються полями запису. Як і масиви, записи є структурами прямого доступу, проте, на відміну від масивів, можуть зберігати елементи (поля), що відносяться до різних типів даних. Щоб можна було посилатися на той чи інший компонент запису, поля іменуються. Структура оголошення типу запису така:

<ім'я типу> = RECORD <список полів> END

Тут <**ім'я типу**> – правильний ідентифікатор;

RECORD, END – зарезервовані слова (запис, кінець); <**список полів**> – список полів; все це є послідовністю розділів записи, між якими ставиться крапка з комою.

Кожен розділ запису складається з одного або декількох ідентифікаторів полів, відокремленних один від одного комами. За ідентифікатором (ідентифікаторами) ставиться двокрапка і опис типу поля (полів), наприклад:

```
type     BirthDay = record
   day, month : Byte;
   year :          Word;
end;
var      a,b : Birthday;
```

У цьому прикладі тип BirthDay (день народження) є запис з полями DAY, MONTH і YEAR (день, місяць і рік); змінні A і B містять записи типу BirthDay.

Як і в масиві, значення змінних типу запису можна присвоювати іншим змінним того ж типу, наприклад

```
a := b;
```

До кожного з компонентів запису можна отримати доступ, якщо використовувати складове ім’я, тобто вказати ім’я змінної, потім **крапку** та ім’я поля:

```
a.day :=  27;
b.year := 1939;
```

Поля можуть бути будь-якого типу (окрім файлу), у тому числі і типу запис

```
 Var c : Record
       name : String;
       bd : BirthDay;
 End;
 .....
 If  Pers.bd.year = 1984 Then begin
Pers.name := ‘Іванов’;
Pers.bd.month := 12;
 End;
```

Імена полів повинні бути унікальними в межах того запису, де вони оголошені, проте, якщо записи містять поля-записи, то імена можуть повторюватися на різних рівнях вкладеності.

Відмітимо, що у даному прикладі постійно доводиться звертатися до полів однієї і тієї ж змінної типу запис, і, отже, постійно писати її ім’я. Є можливість позбавитися від цієї незручності. У TURBO-PASCAL є **оператор приєднання** (**with**), який дозволяє один раз вказати, яким записом ми користуємося і надалі писати лише імена полів. Цей оператор виглядає так:

with <ім’я_запиу> **do** <оператор>;

Найчастіше як оператор використовується складний оператор. Наприклад:

```
with Pers do begin
   name := ‘Іванов’;
  bd.month := 12;
   ...
end;
```

Зазвичай запис містить сукупність різнотипних атрибутів, що відносяться до одного об'єкту. Наприклад, анкетні відомості про студента вузу можна представити у вигляді інформаційної структури, яка називається однорівневим деревом. У Pascal ця інформація може зберігатися в одній змінної типу Record (запис). Задати тип і описати відповідну змінну можна таким чином:

```
Type Anketal = Record
                  FIO  :      String[50];
                  Pol  :      Char;
                  Dat  :      String[16];
                  Adres  :    String[50];
                  Curs  :     1..5;
                  Grup        1. .10;
                  Stip :      Real;
           End;
```

```
Var Student : Anketal;
```

Поля запису можуть мати будь-який тип, зокрема, самі можуть бути записами. Така можливість використовується у випадку, якщо потрібно представити багаторівневе дерево (більше одного рівня). Наприклад, ті ж відомості про студентів можна відобразити дворівневим деревом. Така організація даних дозволить, наприклад, робити вибірки інформації по року народження або по місту, де живуть студенти. У цьому випадку опис відповідного запису буде мати наступний вигляд:

```
Type Anketa2 = Record
                  FIO: String[50];
                  Pol: Char;
                  Dat: Record
                          God: Integer;
                          Mes: String[10];
                          Den: 1..31
                       End;
                  Adres: Record
                            Gorod: String[20];
                            UIDomKv: String[30];
                         End;
                  Curs: 1..5;
                  Grup: 1..10;
                  Stip: Real
               End;
Var Student: Anketa2;
```

Поля такого запису, що знаходяться на другому рівні, ідентифікуються потрійним складовим ім'ям, наприклад, Student.Dat.God.

Для повного розуміння можливостей та правил роботи із записами розглянемо програму по обробці бази даних. База даних у нашому випадку - це дані про власників автомобілів. У кожному рядку текстового файлу BazaAvto.txt знаходяться дані для окремого власника: номер автомобіля; тип (марка) автомобіля; прізвище, ім’я, по батькові; адреса власника. У кінці кожного рядка при наборі такого файлу слід натискати клавішу Enter.

При наборі текстового файлу слід строго дотримуватись наступних правил:

1. Рядки не повинні розділятися пропусками, тобто їх слід набирати «впритул» один до одного. Кількість знакомісць у кожному рядку повинна строго відповідати довжині рядка, який замовлений при його описі, бракуючі місця необхідно заповнювати пробілами справа – у кінці фактичної довжини рядка. У цьому випадку усі однойменні поля-характеристики розташовуватимуться в окремих колонках.

2. Числа відділяються одне від одного пробілами. На кожне число одного і того ж поля (для різних рядків) слід виділити однакову кількість знакомісць. Зайві знакомісця необхідно заповнювати нулями ліворуч.

Приклад 5.7. Записати у текстовий файл BazaAvto.txt відомості про автомобілі: номер; тип (марка) автомобіля; прізвище, ім'я, по батькові; адреса власника. Програма повинна вивести у файл Rez_Avto.txt відомості про автомобілі, номери яких містять три задані цифри у будь-якому порядку.

```
Program Avto;
Uses    Crt;
Const   Chislo = 50; {Максимальне число номерів}
        Baza = 'BazaAvto.txt';
        Resyltat = 'Rez_Avto.txt';
        C1= 3; {Цифри номерів автомобілів для пошуку}
        C2= 5;
        C3= 7;
Type    n_int = 0..Chislo;
  Adres_zap = Record
        ulica: String[11];
        n_dom: String[7];
        n_kv: Integer;
  End; {Adres_zap}
  Avtom_zap = Record
        nom_avto: String[10];
        tip_avto: String[8];
        vladelec:  String[32];
        adres: Adres_zap;
  End; {Avtom_zap}
  Mas_Avtom = Array[n_int] of Avtom_zap;
Var     Chislo_Avtom: n_int;
  Avtom: Mas_Avtom;
  Baza_f, Rezyltat_f: Text;
Procedure Read_Avto;
{По рядках зчитуються з файлу Baza_f у масив записів Avtom}
{дані про номери автомобілів, підраховується їх кількість Chislo_Avtom}
Var     i: n_int;  {Поточний номер рядка}
Begin
   Assign(Baza_f,Baza);
   Reset(Baza_f); i:= 0;
   while (not EOF(Baza_f) or (i= Chislo)) do begin
     i:= i + 1;
     with Avtom[i], adres do
     ReadLn(Baza_f, nom_avto, tip_avto, Vladelec, ulica, n_dom, n_kv);
   end; {while}
   Chislo_Avtom := i;
```

```
  Close(Baza_f);
End; { Read_Avto }
Procedure Poisk_Avto;
Var      i: n_int;
  C1_Str, C2_Str, C3_Str: String[1];
  Poisk: Boolean;
Begin
  Assign(Rezyltat_f, Resyltat);
  Rewrite(Rezyltat_f);
  Write(Rezyltat_f, 'Відомості про автомобілі, номера яких');
  Writeln(Rezyltat_f, ' містять числа ',C1, ';',C2,';',C3,':');
  Writeln(Rezyltat_f);
  Str(C1,C1_Str); {Перетворення числа у рядок}
  Str(C2,C2_Str);
  Str(C3,C3_Str);
  Poisk:= False;  {початковий стан прапорця}
  for i:= 1 to   Chislo_Avtom do
    with Avtom[i] do
    if (Pos(C1_Str,nom_avto) > 0) and
        (Pos(C2_Str,nom_avto)> 0) and
        (Pos(C3_Str,nom_avto)> 0) then begin
      Poisk:= True; {Результат пошуку успішний}
      Writeln(Rezyltat_f, ' ', nom_avto, ', ', tip_avto, ', ', Vladelec);
      with adres do
        Writeln (Rezyltat_f, ' адрес: ', ulica, ' д. ', n_dom, ', кв. ', n_kv);
        Writeln(Rezyltat_f);
  end; {then}
  if Poisk = False then
     Writeln(Rezyltat_f,'В номерах бази даних шуканих номерів немає');
  Close(Rezyltat_f);
End; { Poisk_Avto }
BEGIN        {Головна програма}
  ClrScr;
  Read_Avto;
  Poisk_Avto;
  Writeln('Результати пошуку знаходяться у файлі Rez_Avto.txt');
  ReadLn; {Для затримки результату на екрані дисплею}
END. {Avto}
```

Приклад набору файлу BazaAvto.txt:

```
28-37  ЧАБ CITROEN    Антоненко  Антон  Антонович  Гагарина   24/2  0041
35-27  ЧФА МОСКВИЧ    Иванов     Иван   Иванович   Шевченко   8     0005
72-36  ЧАП BENTLEY    Петренко   Петр   Петрович   Смелянская       60051
25-73  ЧАТ  AUDI       Сидоренков Сидор  Сидорович  Громова    29/3  0112
```

5.2.3. Множини

Множинний тип даних нагадує тип даних, які перераховуються. У той же час, множина – набір елементів, які не зорганізовані у порядку дотримання. У математиці множина – будь-яка сукупність елементів довільної природи. Поняття множини у програмуванні значно вужче за математичне поняття.

Множини у Pascal – це набори однотипних логічно пов'язаних один з одним об'єктів. Характер зв'язків між об'єктами лише враховується програмістом та ніяк не контролюється TURBO-PASCAL. Кількість елементів, яка входять у множину, може мінятися у межах від 0 до 256 (множина, яка не містить елементів, називається порожньою). Саме непостійністю кількості своїх елементів множини відрізняються від масивів та записів.

Дві множини вважаються еквівалентними тоді і тільки тоді, коли всі їх елементи однакові, причому порядок розташування елементів у множині не має значення. Якщо всі елементи однієї множини входять також і в іншу, то говорять про включення першої множини у другу. Порожня множина включається в будь-яку іншу.

Приклад визначення та завдання множин:

```
  type  digitChar = set of '0'..'9';
       digit = set of 0. .9;
var    s1,s2,s3 :  digitChar;
       s4,s5,s6 :  digit;
begin
       s1 := ['1','2','3'];
       s2 := ['3','2','1'];
       s3 := ['2','3'];
       s4 := [0..3,6];
       s5 := [4,5];
       s6 := [3..9];
end.
```

У цьому прикладі множини S1 та S2 еквівалентні, а множина S3 включена у S2 та S1, але не еквівалентна їм.

Опис типу множини має вигляд:

<ім'я типу> = SET OF <баз.тип>

Тут <ім'я типу> – правильний ідентифікатор; SET, OF – зарезервовані слова; <баз.тип> – базовий тип елементів множини, що може використовувати будь-який порядковий тип, окрім WORD, INTEGER, LONGINT.

Для завдання множини використовується так званий конструктор множини: список специфікацій елементів множини, які відокремлюються один від одного комами; список оточується квадратними дужками (див. попередній

приклад). Специфікаціями елементів можуть бути константи або вирази базового типу, а також тип-діапазон того ж базового типу.

Над множинами визначені наступні операції:

***** перетин множин; результат містить елементи, загальні для обох множин; наприклад, S4*S6 містить [3,6], S4*S5 – порожня множина (див. вище);

+ об'єднання множин; результат містить елементи першої множини, доповнені елементами, яких бракує у другій множині:
S4+S5 містить [0,1,2,3,4,5,6];
S5+S6 містить [3,4,5,6,7,8,9];

- різниця множин; результат містить елементи з першої множини, які не належать другій:
S6-S5 містить [3,6,7,8,9];
S4-S5 містить [0,1,2,3,6];

= перевірка еквівалентності; повертає TRUE, якщо обидві множини еквівалентні;

<> перевірка нееквівалентності; повертає TRUE, якщо обидві множини нееквівалентні;

<= перевірка входження; повертає TRUE, якщо перша множина включена у другу;

>= перевірка входження; повертає TRUE, якщо друга множина включена у першу;

IN перевірка приналежності; у цій бінарній операції перший елемент – вираз, а другий -множина того самого типу; повертає TRUE, якщо вираз має значення, що належить множині:
3 in s6 повертає TRUE;
2*2 in s1 повертає FALSE.

Додатково до цих операцій можна використовувати дві процедури.

1) **INCLUDE** – додає новий елемент у множину. Звернення до процедури:
INCLUDE (S,I)

Тут S – множина, що складається з елементів базового типу TSetBase;
I – елемент типу TSetBase, який необхідно включити у множину.

2) **EXCLUDE –** виключає елемент з множини. Звернення:
EXCLUDE(S,I)

Параметри звернення – такі самі, як й у процедури INCLUDE.

На відміну від операцій + та -, що реалізовують аналогічні дії над двома множинами, процедури оптимізовані для роботи з одиночними елементами множини і тому відрізняються високою швидкістю виконання.

У прикладі 5.8, який ілюструє прийоми роботи з множинами, реалізується алгоритм виділення з першої сотні натуральних чисел усіх простих чисел. У його основі лежить прийом, відомий під назвою «**решето Ератосфена**». Відповідно до цього алгоритму спочатку формується множина BEGINSET, що складається з усіх цілих чисел у діапазоні від 2 до N. У множину PRIMERSET

(воно міститиме прості числа, які треба знайти) додається 1. Потім циклічно повторюються такі дії:

- взяти з BEGINSET перше вхідне у нього число NEXT та помістити його у PRIMERSET;
- вилучити з BEGINSET число NEXT та всі інші числа, які кратні йому, тобто 2*NEXT, 3*NEXT і так далі.

Цикл повторюється доти, доки безліч BEGINSET не стане порожньою.

Цю програму не можна використовувати для довільного N, оскільки у будь-якій множині не може бути більше ніж 256 елементів.

Приклад 5.8.

```
Program Primer_numbers_detect;
{Виділення усіх простих чисел з перших N цілих}
const N = 255;                  {Кількість елементів початкової множини}
type  SetOfNumber = set of 1..N;
var   n1, next, i : Integer;          {Допоміжні змінні}
      BeginSet,                 {Початкова множина}
      PrimerSet : SetOfNumber;  {Безліч простих чисел}
begin
      BeginSet := [2..N];       {Створюємо початкову множину}
      PrimerSet:= [1];  {Перше просте число}
      next:= 2;         {Наступне просте число}
      while BeginSet <> [] do {Початок основного циклу}
      begin
        n1 := next;  {n1-число, кратне черговому простому (next)}
      {Цикл видалення з початкової множини непростих чисел}
        while n1 <= N do
        begin
            Exclude(BeginSet, n1); {Вилучає елемент з набору}
            n1 := n1+next {Наступне кратне}
        end;      {Кінець циклу вилучення}
        Include(PrimerSet,next);
{Отримуємо наступне просте –  перше невикреслене з початкової множини}
        repeat
          inc(next)
        until (next in BeginSet) or (next < N)
      end;  {Кінець основного циклу}
      {Виводимо результат:}
      for i := 1 to N do
        if i in PrimerSet then Write(i:8);
      WriteLn;
      ReadLn
```

```
END.
```

Приклад 5.9.

{Організувати введення елементів трьох множин A, B, C. }

```
{Обчислити та надрукувати множини A + B, B * C, C – A}
Program CalcSet;
Type SetChar =  Set Of Char;
Var A, B, C : SetChar;
{Процедура введення множини}
Procedure EnterSet(Var D: SetChar);
Var  ch: Char;
Begin
   Writeln('Введіть елементи множини, крапка –  кінець введення');
   Read(ch);
   D:=[];
   While ch <> '.' Do  Begin
       D := D + [ch];
       Read(ch)
    End;
End; {EnterSet}
{Процедура виведення множини}
Procedure PrintSet(D: SetChar);
Var   i: Byte;
Begin
    For  i := 0 To 255 Do
     If Chr(i)  in D Then
        Writeln(Chr(i));
        End; {PrintSet}
        Begin {Основна програма}
           Writeln('Введіть елементи множини A');
           EnterSet(A);
           Writeln('Введіть елементи множини B');
           EnterSet(B);
           Writeln('Введіть елементи множини C');
           EnterSet(C);
           PrintSet(A+B);
           PrintSet(B*C);
           PrintSet(C-A);
           Readln;
End.
```

У прикладі 5.9 ми "не витримали" і оформили частину тексту програми у вигляді процедури (Procedure EnterSet(Var D: SetChar) – процедура введення множини).

Слово PROCEDURE читається “про’сидже”, перекладається “процедура”. Ім’я процедури створюється за тими самими правилами, що й ім’я змінної. Все, що йде після імені, будемо називати тілом процедури. Докладніше про це ми розповімо у розділі 6.

5.2.4. Рядки

Продовжимо знайомство із строковим типом даних. Тип **STRING** (рядок) у TURBO-PASCAL широко застосовується для обробки текстів. Він багато у чому схожий на одновимірний масив символів ARRAY[0..N] OF CHAR, проте, на відміну від останнього, кількість символів у рядку-змінній може змінюватися від 0 до N, де N – максимальна кількість символів у рядку. Значення N визначається оголошенням типу STRING [N] і може бути будь-якою константою порядкового типу, але не більше 255. TURBO-PASCAL дозволяє не вказувати N, у цьому випадку довжина рядка приймається максимально можливою, а саме N=255 .

Рядок у TURBO-PASCAL трактується як низка символів. До будь-якого символу у рядку можна звернутися так само, як і до елементу одновимірного масиву ARRAY [0..N] OF CHAR, наприклад:

```
var     st : String;
begin
        if st[5]= ‘A’ then...
end.
```

Найперший байт в рядку має індекс 0 і містить поточну довжину рядка, перший значущий символ рядка займає другий байт і має індекс 1. Над довжиною рядка можна здійснювати необхідні дії і у такий спосіб змінювати довжину. Наприклад, вилучити з рядка всі пробіли можна таким чином:

```
var     st : String;
        i : Byte;
begin
        i := ord(st [0] );      {i –  поточна довжина рядка}
        while (i <> 0) and (st[i]= ‘ ‘) do
        begin
                dec(i);
                st[0]:= chr(i)
        end;
end.
```

Значення ORD(st[0]), тобто поточну довжину рядка, можна отримати і за допомогою функції **LENGTH**(st), наприклад:

```
while (Length(st) <>0) and (st[Length(st)]=‘ ‘) do
```

st[0]:= chr(Length(st)-1)

Над рядками визначені такі операції:

1. Операція **зчеплення** (+) застосовується для зчеплення кількох рядків в один.

Наприклад:

```
st := ‘a’ + ‘b’;
st := st + ‘c’;          {st містить "abc"}
```

Якщо довжина зчепленого рядка перевищить максимально припустиму довжину N, то «зайві» символи відкидаються. Наступна програма, наприклад, надрукує символ 1:

```
var     st: String [1] ;
begin
        St:=‘1’ + ‘23’;
        WriteLn(st)
end.
```

2. Операції **відношення** (=, <>, >, <, >=, <=) проводять порівняння двох рядків зліва направо до першого неспівпадаючого символу, і той рядок вважається більшим, в якій перший неспівпадаючий символ має більший номер в стандартній таблиці обміну інформацією (ASCII). Результат виконання операцій відношення над рядками завжди має логічний тип.

Наприклад, вираз ‘MS-DOS’ < ‘MS-Dos’ має значення True.

Якщо рядки мають різну довжину, але в загальній частині символи збігаються, вважається, що коротший рядок менший, ніж довший.

Рядки вважаються рівними, якщо вони збігаються по довжині і містять одні і ті ж символи на відповідних місцях у рядку.

Решта всіх дій над рядками та символами реалізується за допомогою описаних нижче стандартних процедур та функцій:

CONCAT(S1 [,S2 ..., SN]) – функція типу STRING; повертає рядок, що є зчепленням рядків-параметрів SI, S2 ..., SN.

COPY(ST, INDEX, COUNT) – функція типу STRING; копіює з рядка ST COUNT символів, починаючи з символу з номером INDEX.

DELETE (ST, INDEX, COUNT) – процедура; вилучає Count символів з рядка ST, починаючи з символу з номером INDEX.

INSERT (SUBST, ST, INDEX) – процедура; вставляє підрядок SUBST у рядок ST, починаючи з символу з номером INDEX.

LENGTH (ST) – функція типу INTEGER; повертає довжину рядка ST.

POS (SUBST, ST) – функція типу INTEGER; відшукує в рядку ST перше входження підрядка SUBST та повертає номер позиції, з якої вона починається; якщо підрядок не знайдений, повертається нуль.

STR (X: real; var S: string),

STR (X: integer; var S: string) – перетворює число X у рядок S.

VAL (S: string; var X: real; Error: integer),

VAL (S: string; var X: integer; var Error: integer) – перетворює рядок S у число X. Якщо перетворення вдалось, Error = 0, інакше Error = номеру першого неперетвореного символу.

Приклади:

```
var      x : Real;
y : Integer;
st, st1: String;
begin
st := concat('12','345');          {рядок st містить 12345}
st1 := copy(st,3,Length(st)-2);  {st1 містить 345}
insert('-',st1,2);                 {рядок st1 містить 3-45}
delete(st,pos('2',st),3);          {рядок st містить 15}
end.
```

Операції відношення =, <>, >, <, >=, <= виконуються над двома рядками посимвольно, зліва направо з урахуванням внутрішнього кодування символів (див. табл. 5.1). Якщо один рядок менше за інший за довжиною, символи короткого рядка, яких бракує, замінюються значенням CHR(0).

Наступні операції відношення дадуть значення TRUE:

'A' > '1'

'Turbo' <' Turbo Pascal'

'Pascal' >'Turbo Pascal'

Приклад 5.10.

```
{ Програма інвертує (перевертає) слово, що знаходиться в змінній Str1 }
program Invers;
Var          Str1 : String;
       Chr1: Char;       { Змінна для тимчасового зберігання символу, що переставляється }
       K     :  Integer;
begin
 Str1:= ' БАР';
 for k:= 1 to (Length(Str1) div 2) do begin
       Chr1 := Str1[k];
       Str1[k]:= Str1[Length(Str1)-k+1];
       Str1[Length(Str1)-k+1]:= Char1;
 End;
 WriteLn(Str1); {На екран буде виведено слово –  РАБ}
  ReadLn
End.
```

Приклад 5.11.

```
{ Програма інвертує слово двічі, тобто залишає його без зміни }
program Invers;
Var          Str1 : String;
```

```
      Chr1: Char;       { Змінна для тимчасового зберігання символу, що
переставляється }
      K     :  Integer;
begin
 Str1:= ‘ БАР’;
 for k:= 1 to Length(Str1) do begin
      Chr1 := Str1[k];
      Str1[k]:= Str1[Length(Str1)-k+1];
      Str1[Length(Str1)-k+1]:= Char1;
 End;
 WriteLn(Str1); {На екран буде виведено слово –  БАР}
  ReadLn
End.
```

Для того щоб ввести значення типу STRING, необхідно використовувати оператори READLN, а не READ. Більше того, за один раз може бути введений тільки один рядок.

Приклад 5.12.

```
{ Програма зчитує рядок слів, які розділені пропусками }
{ Потім слова друкуються у зворотному порядку }
program RWord104;
Var     OldLine, NewLine: String;  { Змінні для старого та нового рядка }
        Blank: Integer;
        Word: String;
Begin
  NewLine := ‘‘;
  WriteLn(‘Введіть рядок, слова якого повинні реверсувати:’);
  WriteLn;
  ReadLn(OldLine);
  OldLine := Concat(OldLine,’ ‘);    { Додати пробіл }
  While OldLine <> ‘‘ do
    Begin
      Blank := Pos(‘ ‘,OldLine);
      Word := Copy(OldLine,1,Blank);
      NewLine := Concat(Word,NewLine);
      Delete(OldLine,1,Blank)
    end;
  WriteLn;
  WriteLn(‘Ось новий рядок із зворотним порядком слів:’);
  WriteLn(NewLine);
  ReadLn
End.
```

Приклад 5.13.

{Програма зчитує символьний рядок та друкує його слова у порядку за абеткою }

```
program Alpha;
Const   LineSize = 30;
Var     Line: array[1..LineSize] of String;
      Count, I, K, Kol_s: Integer;
```

```
        Str: String;
    Begin
      { Читання рядка}
      WriteLn('Введіть рядок тексту, який треба впорядкувати');
      ReadLn(Str);
      WriteLn;
      Kol_s:=1;
      {Виділення та запис слів у масив}
      For I := 1 to Length(Str) do
          if Str[I] <> ' ' then
      Line[Kol_s] := Line[Kol_s] + Str[I]
         else
          if Line[Kol_s] <> '' then
                    Kol_s := Kol_s + 1;
      if Line[Kol_s] = '' then
         Kol_s:= Kol_s –  1;    {Кількість слів у реченні та у масиві}
    { Сортування і друк списку слів }
      For I := 1 to Kol_s do
      Begin
         K := 1;
         While Line[K] = '' do
            K := K + 1;
         For Count := 2+(K-1) to Kol_s do
       If (Line[K] <> '') and (Line[Count] < Line[K]) and (Line[Count] <> '')
then
         K := Count;
         If Line[K] <> '' then
            Write(Line[K],' ');
         Line[K] := ''
      end;
      ReadLn
    End.
```

Приклад 5.14.

{З клавіатури вводиться рядок, треба перетворити цей рядок так }
{ щоб у ньому залишилися лише букви. }

```
program Bukva;
const letters = ['a'..'z','A'..'Z''];
var      s, s1 : string;
         i : integer;
begin
WriteLn('Vvedite stroku: ');
  ReadLn(s);
  WriteLn;
  s1 := '';
  for i := 1 to length(s) do
              if s[i] in letters then
              s1 := s1 + s[i];
WriteLn(s1);
  ReadLn;
end.
```

Прклад 5.15.

```
{Фрагмент програми, яка вилучає зайві пропуски між словами}
{ . . . }
 p:=1;
 while p <> 0 do begin
     p:=pos (‘  ‘,s);          {два пробіли}
     if p>0 then delete (s,p,1);
 end;
 if s[1] = ‘  ‘  then delete (s,1,1);
 if s[length(s)] = ‘  ‘ then delete (s, length(s),1);
 writeln (s);
```

Приклад 5.16.

```
{Робота з рядком як  з масивом символів (рахуємо  пропуски у рядку)}
var                    s:string;
  k,i:integer;
begin
  writeln (‘Text?’);
  readln (s);
  k:=0;
  for i:=1 to length (s) do
       if s[i]=‘ ‘ then k:=k+1;
  writeln (‘k=‘,k);
end.
```

5.3. Створюємо типи даних

Pascal надає можливість не тільки користуватися стандартними типами даних, але також іменувати їх по-іншому і навіть створювати свої типи.

Запис **TYPE** bukva = Char;

означає: ТИП bukva"дорівнює" (еквівалентний) типу Char

тобто ми просто придумали типу Char ще одну назву "bukva". Тепер все одно, як записати:

VAR a,b: Char; або VAR a,b: bukva;

Ще приклади:

```
TYPE      Vector      = array[1..10] of Integer;
          Matritsa = array[1..8] of Vector;
VAR       A,B: VECTOR;
          c  : matritsa;
          d  : array[1.. 8] of Vector;
```

Тут ми створили два нові типи з іменами Vector та matritsa. Очевидно, що змінні *c* та *d* описані однаково. Зверніть увагу, що замість **TYPE** matritsa = **array**[1.. 8] **of** Vector можна записати **TYPE** matritsa = **array**[1.. 8] **of** **array**[1..10] **of** Integer

або **TYPE** matritsa = **array**[1..8,1..10] **of** Integer .

Навіщо потрібні нові типи? Ось дві з декількох причин. Одна – наочність і зручність. Інша – чисто граматична – Pascal дозволяє в певних конструкціях записувати лише імена типів, а не їх визначення. Наприклад, коли ми будемо

вивчати процедури з параметрами, ми дізнаємося, що писати PROCEDURE par(a: array[1..10] of Integer); неправильно, писати PROCEDURE par(a: Vector); – правильно.

Контрольні запитання для самоперевірки

1. Які дані відносяться до даних простого типу?
2. Які функції можна застосувати для всіх даних, що перераховуються?
3. Які існують цілі типи даних?
4. Який порядок вкладеності цілих типів даних?
5. Які функції застосовуються до даних цілого типу?
6. Які функції застосовуються до даних символьного типу?
7. Як задаються дані типу, що перераховується та з чого він складається?
8. Як задаються дані інтервального типу?
9. Які функції застосовуються до даних інтервального типу?
10. Які існують типи дійсних даних?
11. Як здійснити піднесення будь-якого числа до довільного ступеня?
12. Що таке іменовані константи?
13. Як визначається тип виразу?
14. Як оголошуються масиви даних?
15. З яких елементів можуть створюватись масиви даних?
16. Що являє собою запис та з чого він може складатись?
17. Що являють собою множини та які обмеження для них у мові PACAL?
18. Які операції дозволяються над множинами?
19. Чим відрізняються рядки даних від одновимірного масиву символів?
20. Які процедури та функції використовуються для обробки рядків?
21. Як створити довільний тип даних?

6. Підпрограми. Процедури та функції

Вже з перших занять ми вам рекомендуємо при розробці програм строго дотримуватися правил хорошого стилю програмування, який дозволяє створювати легко зрозумілі програми. Одним з методів хорошого стилю програмування є також застосування підпрограм, які розробляються програмістом на додаток до вже існуючих стандартних підпрограм. Написання будь-якої великої програми неможливе без розбиття задачі на простіші ***підзадачі***, які ми можемо розв'язувати незалежно, так і без повторного використання раніше написаних фрагментів. Розв'язати ці найважливіші завдання дозволяє механізм *підпрограм*, який присутний у будь-якій мові програмування, зокрема, і у Pascal.

Підпрограмою називають незалежну частину програми, призначену для розв'язку деякої підзадачі. Підпрограма взаємодіє з основною програмою через механізм ***параметрів*** – так називають вхідні та вихідні дані, з якими працює підпрограма. Одного разу написана підпрограма виконана з різними значеннями параметрів, може розв'язувати деякий клас задач.

Використання підпрограм у чомусь схоже на розрахунки з використанням математичних або фізичних формул. Так, маючи загальні формули розв'язку квадратного рівняння, ми можемо підставити замість коефіцієнтів *a, b* і *c* будь-які числові значення і розв'язати будь-яке конкретне рівняння. Аналогічно, ми могли б написати підпрограму з ***вхідними*** параметрами *a, b, c* та ***вихідними*** параметрами *x1, x2* (знайдені корені рівняння), а потім, використовуючи потрібне число разів цю підпрограму (одноразове використання підпрограми називається її ***викликом***), розв'язати будь-яку кількість квадратних рівнянь.

Отже, використання підпрограм дозволяє розв'язати такі задачі:

- зменшення розмірів коду та економія пам'яті за рахунок можливості неодноразового виклику однієї і тієї ж підпрограми в межах однієї програми;
- краща структуризація програми за рахунок розбиття задачі на простіші підзадачі;
- ефективне повторне використання один раз написаного коду.

У Pascal існує два види підпрограм, які ми вивчимо по цій темі, – ***процедури*** та ***функції***. У програмі може бути довільна кількість як функцій, так і процедур.

Як наголошувалося у розділі 2, процедури та функції є відносно самостійними фрагментами підпрограм, які оформлені особливим чином та мають свої імена. Згадка цього імені у тексті програми називається викликом процедури (функції). Відмінність функції від процедури полягає у тому, що результатом виконання операторів, які створюють тіло функції, завжди є деяке єдине значення або покажчик, тому звернення до функції можна використовувати у відповідних виразах разом із змінними та константами. Домовимося далі називати процедуру або функцію загальним іменем «**підпрограма**». Підпрограми є інструментом, за допомогою якого будь-яка

програма може бути розбита на низку до певної міри незалежних одна від одної частин. Таке розбиття необхідне з двох причин.

По-перше, це засіб економії пам'яті: кожна підпрограма існує у програмі в єдиному примірнику, тоді як звертатися до неї можна багато разів з різних точок програми. При виклику підпрограми активізується послідовність операторів, що її складають, а за допомогою параметрів, які передаються підпрограмі, потрібним чином модифікується алгоритм, який реалізовується у ній.

Друга причина полягає у застосуванні методики нисхідного проектування програм. У цьому випадку алгоритм подається у вигляді послідовності викликів великих підпрограм, які реалізовують більш менш самостійні смислові частини алгоритму. Підпрограми у свою чергу можуть розбиватися на менші підпрограми нижнього рівня і так далі (рис. 7.1). Послідовна структуризація програми продовжується доти, доки алгоритми, які реалізовуються за допомогою підпрограм, не стануть настільки простими, щоб їх можна було легко запрограмувати.

6.1. Локалізація імен

Нагадаємо, що виклик підпрограми здійснюється простим використанням імені процедури в операторі виклику процедури або імені функції у виразі. У точці виклику підпрограми управління передається першій її команді. Виконання підпрограми завершується діями повернення управління у точку виклику – тобто наступному оператору, безпосередньо за оператором виклику в основній програмі, який називається точкою повернення.

При використанні розширеного синтаксису TURBO-PASCAL (див. нижче) функції можна викликати так само, як і процедури. Як відомо, будь-яке ім'я у програмі повинно бути обов'язково описано, перш ніж воно з'явиться серед операторів виконання. Не робиться виключення і стосовно підпрограм: кожну свою процедуру та функцію програмістові необхідно описати у розділі описів.

Описати підпрограму – це означає вказати її **заголовок** та **тіло**. У заголовку оголошуються ім'я підпрограми та формальні параметри у вигляді списку, якщо вони є. Для функції, крім того, вказується тип результату, що повертається нею. За заголовком слідує тіло підпрограми, яке, подібно до програми, складається з розділу описів та розділу операторів виконання. У розділі описів підпрограми можуть зустрітися описи підпрограм нижчого рівня, а в них – описи інших підпрограм і так далі

Ось яку ієрархію описів отримаємо, наприклад, для програми, структура якої зображена на рис.6.1 (для простоти вважається, що усі підпрограми являють собою процедури без параметрів):

Рис. 6.1. Приклад структури програми

```
Program ...;
Var розділ_описів_1;
Procedure A;
Procedure A1;
begin
      . . . .
end {A1};
Procedure A2;
Begin
      . . . .
end {A2};
begin {A}
      . . . .
end {A};
Procedure B;
Procedure B1;
begin {B1};
end
Procedure B2 ;
Procedure B21;
Var раздел_описів_2;
Begin
 {Тіло головної програми}
End.
```

Як видно з опису, кожна підпрограма має тіло, яке складається з операторів. Подібно до тіла циклу, тіло підпрограми оточується операторними дужками `begin ... end;`. Зверніть увагу, що в лістингу два розділи описів. Перший з них розташований до підпрограм, другий – після них перед тілом головної програми. Дані, які описані у першому розділі, – ***глобальні***, вони доступні всім частинам програми, які розташовані нижче за її текстом. Дані другого розділу описів доступні лише головній програмі, оскільки описані

безпосередньо перед нею. Загальне правило дуже просте: підпрограми "бачать" усі глобальні змінні, які описані вище за їх тіла. Аналогічно, без вживання спеціальних заходів підпрограма "бачить" і може викликати будь-яку іншу підпрограму, яка розташована вище за неї по тексту програми. Тут друга підпрограма може викликати першу, але не навпаки. Головна програма, як правило, розташована останньою і може викликати всі підпрограми.

На практиці **не рекомендується** робити підпрограми залежними від глобальних даних, оскільки це знижує їх **переносимість** (можливість повторного використання). Зрозуміло, що будь-якого з розділів описів в конкретній програмі може і не бути. Більш того, оскільки підпрограма – окрема та, в ідеалі, незалежна, вона може містити власний розділ опису **локальних** змінних, що призначені лише для її потреб та невидимих з інших частин програми. Наприклад, для підпрограми розв'язку квадратного рівняння такою локальною змінною могла б бути змінна `d`, яка призначена для обчислення дискримінанта рівняння. Пояснимо цей важливий момент на прикладі:

```
Var i:integer;
{глобальна змінна – описана поза всіма підпрограмами}
Заголовок Підпрограми;
Var i:integer;
{локальна змінна – описана після заголовка підпрограми}
Begin
 {Тіло підпрограми}
End;
Begin
{Тіло головної програми}
End.
```

Описана перед тілом підпрограми локальна змінна `i` не має жодного відношення до однойменної змінної, описаної вище. На час виконання підпрограми локальна змінна `i` витісняє глобальну, роблячи значення останніх недоступним. Після завершення підпрограми значення локальної `i` буде загублено, а значення глобальної `i` ніяк від цього не зміниться.

Таким чином, підпрограма будь-якого рівня може мати безліч імен констант, змінних, типів і вкладених в неї підпрограм нижчого рівня. Вважається, що всі імена, які описані всередині підпрограми, **локалізуються** у ній. Тобто, вони як би «невидимі» ззовні підпрограми. Таким чином, з боку операторів, що використовують звернення до підпрограми, вона трактується як «чорна скриня», у якій реалізується той чи інший алгоритм. Усі деталі цієї реалізації приховані від очей користувача підпрограми і тому є недоступними для нього. Наприклад, у розглянутому вище прикладі з основної програми можна звернутися до процедур A і B, але не можна викликати жодну з вкладених в них процедур A1, A2, B1 і т.і.

Ці угоди стосуються не тільки імен підпрограм, але й взагалі до будь-яких імен, які в них оголошені, – типів, констант, змінних та міток. Всі імена в

межах підпрограми, в якій вони оголошені, повинні бути унікальними, і не можуть співпадати з ім’ям самої підпрограми.

При вході у підпрограму нижчого рівня стають доступними не тільки оголошені у ній імена, але і зберігається доступ до всіх імен верхнього рівня. Іншими словами, будь-яка підпрограма як би оточена напівпрозорими стінками: зовні підпрограми ми не бачимо, що знаходиться всередині, але, потрапивши у підпрограму, можемо спостерігати все, що робиться зовні. Так, наприклад, з підпрограми B21 ми можемо викликати підпрограму A, використовувати імена, що оголошені в основній програмі та підпрограмах B та B2, і навіть звернутися до них. Будь-яка підпрограма може, нарешті, викликати саму себе – такий спосіб виклику називається рекурсією.

Нехай маємо такий опис:

```
Program ..;
var     V1 : ... ;
Procedure A;
var     V2 :...;
. . . .
end {A};
Procedure B;
var     V3 :...;
Procedure Bl;
var     V4 :...;
Procedure B11;
var V5;
. . . .
```

З процедури B11 доступні всі п’ять змінних V1...,V5, з процедури B1 доступні змінні V1..., V4, з основної програми – тільки V1.

При взаємодії підпрограм одного рівня ієрархії набуває чинності основне правило TURBO-PASCAL: **будь-яка підпрограма перед її використанням повинна бути описана**. Тому з підпрограми B можна викликати підпрограму A, але з A викликати B неможливо (точніше, така можливість з’являється тільки з використанням випереджаючого опису, див. п.7.6.) Продовжуючи образне порівняння, підпрограму можна сподобити скрині з непрозорими стінками і дном та напівпрозорим дахом: з підпрограми можна дивитися тільки «вгору» і не можна «додолу», тобто підпрограмі доступні тільки ті об’єкти верхнього рівня, які описані до опису даної підпрограми. Ці об’єкти називаються глобальними по відношенню до такої підпрограми.

На відміну від стандартного Pascal в TURBO-PASCAL припускається довільна кількість опису констант, змінних, типів, міток та підпрограм. Наприклад, розділ VAR опису змінних може з’являтися у межах розділу описів однієї і тієї ж підпрограми багато разів та перемежатися з оголошеннями інших об’єктів та підпрограм. Для TURBO-PASCAL абсолютно байдужим є порядок проходження та кількість розділів VAR, CONST, TYPE, LABEL, але при визначенні області дії цих описів слід пам’ятати, що імена, які описані нижче за текстом програми, є недоступними з раніше описаних підпрограм, наприклад:

```
var     V1 : ...;
```

```
Procedure S;
var     V2 : ...;
end {S};
var     V3 :...;
```

З процедури S можна звернутися до змінних V1 та V2, але **не можна використовувати V3, оскільки опис V3 слідує у програмі за описом процедури S**. Імена, які локалізовані у підпрограмі, можуть співпадати з раніше оголошеними глобальними іменами. У цьому випадку вважається, що локальне ім'я «**затуляє**» глобальне та робить його недоступним, наприклад:

```
var     i : Integer;
Procedure Print;
var     i : Integer;
begin
        writeln(i)
end {Print};
begin
        i := 1;
        Print;
end.
```

Що надрукує ця програма? Все, що завгодно: значення внутрішньої змінної і при вході у процедуру Print не визначено, хоча однойменна глобальна змінна має значення 1. Локальна змінна «закриє» глобальну, і на екран буде виведено довільне значення, яке міститься у неініційованій внутрішній змінній. Якщо прибрати опис

```
var       i : Integer;
```

з процедури Print, то на екран буде виведено значення глобальної змінної і, тобто 1. Таким чином, глобальні та локальні змінні з однаковими іменами – це різні змінні.

6.2. Опис процедури (функції)

Опис підпрограми складається із заголовка та тіла підпрограми.

Заголовок процедури має вигляд:

PROCEDURE <ім'я> [(<сп. ф. п . >)];

Заголовок функції:

FUNCTION <ім'я> [(<сп.ф.п.>)] : <тип>;

Тут <ім'я> – ім'я підпрограми (правильний ідентифікатор);

<сп.ф.п.> – список формальних параметрів;

<тип> – тип результату, який повертається функцією.

Одразу за заголовком підпрограми (функції) може слідувати одна із стандартних директив FORWARD. Директива FORWARD використовується при випереджаючому описі для повідомлення компілятору, що опис

підпрограми слідує десь далі у тексті програми (але у межах поточного програмного модуля).

Список формальних параметрів є необов'язковим і може бути відсутнім. Якщо ж він є, то у ньому повинні бути перераховані імена формальних параметрів та їх типи, наприклад:

Procedure SB(A: Real; B: Integer; C: Char);

Формальні параметри потрібні тільки для опису тіла підпрограми, а конкретні значення параметрів, які ми вказуватимемо у момент виклику підпрограми, називаються **фактичними параметрами**. При виконанні операторів підпрограми формальні параметри як би тимчасово замінюються на фактичні.

Як бачимо з прикладу, параметри у списку відділяються один від одного крапками з комою. Декілька послідовних підряд однотипних параметрів можна об'єднувати у підсписки, розділяючи їх комами, наприклад, замість

Function F(A: Real; B: Real): Real;

можна написати простіше:

Function F(A, B: Real): Real;

Оператори тіла підпрограми розглядають список формальних параметрів як своєрідне розширення розділу описів: усі змінні з цього списку можуть використовуватися у будь-яких виразах всередині підпрограми. У такий спосіб здійснюється налаштування алгоритму підпрограми на конкретне завдання.

Розглянемо наступний приклад. У мові немає операції піднесення до ступеня, проте за допомогою вбудованих функцій LN(X) та EXP(X) нескладно реалізувати нову функцію з ім'ям, наприклад, POWER, що здійснює піднесення будь-якого дійсного числа до будь-якого дійсного ступеня. У програмі (приклад 6.1) вводиться пара чисел X та Y та виводиться на екран дисплея результат піднесення X спочатку у ступінь +Y, а потім – у ступінь -Y.

Приклад 6.1.

```
var   x,y:Real;
Function Power (a, b : Real):Real;
begin {Power}
      if a = 0 then  Power := 0;
      if b = 0 then  Power := 1;
      If a > 0 then begin
        If b > 0 then
            Power := exp(b * ln (a))
        else
            Power := 1/exp(b * ln(a));
      end;
      else begin
        If b > 0 then
            Power := -exp(b * ln(abs(a)))
        else
            Power := -1/exp(b * ln(abs(a)));
```

```
        end;
end {Power} ;
begin {main}
      writeln('Введите два числа: ');
      readln(x,y);
      writeln (Power (x,y) :12:10, Power (x, -y) : 15 : 10)
      readln;
end {main}.
```

Нам вже відомо як викликати функцію з програми (наприклад sqrt). Опис функції дуже схожий на опис процедури, але є дві відмінності:

1) після імені функції та списку параметрів (якщо вони є) через двокрапку записується тип значення функції (можливі не лише числові типи, але і логічні, строкові, символьні);

2) серед операторів функції найважливішими є оператори надання значення функції (у нашому випадку це оператори **Power := exp(b * ln(a)), Power := 1/exp(b * ln(abs(a))), Power := 1** та **Power := 0**).

Для виклику функції POWER ми просто вказали її як параметр при зверненні до вбудованої процедури WRITELN. Параметри X та Y у момент звернення до функції – це фактичні параметри. Вони підставляються замість формальних параметрів A та B у заголовку функції POWER, і потім над ними виконуються необхідні дії. Отриманий результат присвоюється ідентифікатору функції – саме він і буде повернений як значення функції при виході з неї. У програмі функція POWER викликається двічі – спочатку з параметрами X та Y, а потім X та -Y, тому будуть отримані два різні результати.

Механізм заміни формальних параметрів на фактичні дозволяє потрібним чином побудувати алгоритм, що реалізований у підпрограмі. стежити за тим, щоб кількість та тип формальних параметрів строго відповідали кількості та типам фактичних параметрів у момент звернення до підпрограми. Сенс фактичних параметрів, що використовуються, залежить від того, в якому порядку вони перераховані при виклику підпрограми. У прикладі 7.1 перший по порядку фактичний параметр підноситиметься до ступеня, що задається другим параметром, а не навпаки. Користувач повинен сам стежити за правильним порядком перерахування фактичних параметрів у разі звернення до підпрограми.

Будь-який з формальних параметрів підпрограми може бути або **параметром-значенням**, або **параметром-змінною**, або, нарешті, **параметром-константою**.

У попередньому прикладі параметри A і B визначені як **параметри-значення**. Якщо параметри визначаються як **параметри-змінні**, то перед ними необхідно ставити зарезервоване слово VAR, а якщо це параметри-константи, то слово CONST, наприклад:

Procedure MyProcedure(var a: Real; b: Real; const C: String);

Тут A – параметр-змінна, B -параметр-значення, а C – параметр-константа.

Визначення формального параметра тим чи іншим способом істотно, в основному, тільки для програми, яка викликається: якщо формальний параметр оголошений як параметр-змінна, то при виклику підпрограми йому повинен відповідати фактичний параметр у вигляді змінної потрібного типу; якщо формальний параметр оголошений як параметр-значення або параметр-константа, то при виклику йому може відповідати довільний вираз. Контроль над неухильним дотриманням цього правила здійснюється компілятором TURBO-PASCAL. Якби для попереднього прикладу був використаний такий заголовок функції:

Function Power(var a, b : Real) : Real;

то при другому зверненні до функції компілятор вказав би на невідповідність типу фактичних та формальних параметрів (параметр b є вираз, тоді як відповідний йому формальний параметр описаний як параметр-змінна).

Існує ще одна обставина, яку слід враховувати при виборі виду формальних параметрів. Якщо був оголошений формальний параметр як параметр-значення, то при звернені до процедури (функції) здійснюється копіювання фактичного параметра у тимчасову пам'ять. Якщо цим параметром буде масив великої розмірності, то істотні витрати часу та пам'яті на копіювання при багаторазових зверненнях до підпрограми можна мінімізувати, оголосивши цей параметр параметром-константою. Параметр-константа не копіюється у тимчасову область пам'яті, що скорочує витрати часу на виклик підпрограми, проте **будь-які його зміни у тілі підпрограми неможливі** – за цим суворо стежить компілятор.

В усіх прикладах, де необхідне **розкладання цілого числа на цифри,** зручно застосовувати операції: MOD та DIV. Наприклад, якщо дано тризначне число "N" (X1X2X3), то цифри: X1X2X3, складові цього числа, визначаються блоком операторів:

X3:= N mod 10; N:= N div 10; X2:= N mod 10; N:= N div 10; X1:= N;

Приклад 6.2.

```
{Програма виявлення чисел Армстронга у діапазоні від10 до 9999}
program Chislo_Armstronga;
{ Число є числом Армстронга з n цифр, якщо сума цих цифр в n-ій ступені }
{ дорівнює початковому числу. Наприклад, для 153: 153 =1^3 + 5^3 + 3^3 = 1+125+27 }
{ Для числа з 4-х цифр – 1634: 1634 =1^4 + 6^4 + 3^4 + 4^4 = 1+1296+71+256 }
uses Crt; {Опис модуля Pascal для використання функції очищення екрану – ClrScr}
var
    chislo, b, n: longint;
    sum: real;
{Підрахунок кількості цифр в числі}
Procedure Kol_Cifr(var b, n: longint);
    begin
```

```
            while b > 10 do
            begin
               b := b div 10;
               inc(n);
            end;
        end;
{ Функція підрахунку b^n }
Function Stepen(var b: longint):real;
        var    a: longint;
        begin
            a := b mod 10;
            b := b div 10;
            if (a <> 0 ) then
              Stepen := exp(n*ln(a));
        end;
begin
        ClrScr;       { Очистка экрану  }
        writeln('Chisla Armstronga в діапазоні від 10 до 9999:');
        for chislo := 10 to 9999 do begin
            sum := 0;
            n   :=1;
            b   :=chislo;
            Kol_Cifr(b, n); {Підрахунок цифр у числі}
            b := chislo;
            while b >= 10 do
               sum:= sum + Stepen(b);
            if (b <> 0) then
              sum := sum + Stepen(b);
            b := trunc(sum);
            if chislo = b then
              writeLn(chislo:4,' –  число Армстронга');
        end;
        readln;
end.
```

Тут за допомогою оператора Uses викликається опис модуля Pascal Crt для використання функції очищення екрану – ClrScr.

Наведемо ще приклади використання функції, яка визначена програмістом.

Приклад 6.3.

Визначити площу трикутника за заданими координатами трьох його вершин на площині.

Нехай x1, y1, x2, y2, x3, y3 – координати вершин. За формулою Герона площа трикутника рівна Sqrt(p*(p-A)*(p-B)*(p-C)), де p = (A+B+C)/2 – напівпериметр трикутника, A = Довжина(x1,y1,x2,y2), B = Довжина(x2,y2,x3,y3), C = Довжина(x3,y3,x1,y1) – довжини сторін трикутника, а Довжина(x,y,z,u) = Sqrt($(z-x)^2+(u-y)^2$).

З використанням функцій це розв'язок запишемо так:

```
program  Geron;
   var X1,Y1,X2,Y2,X3,Y3,A,B,C,P : real;
   Function Length (X,Y,Z,U : real):real;
      begin Length := Sqrt(Sqr(Z-X) + Sqr(U-Y)) end;
begin
   read(X1,Y1,X2,Y2,X3,Y3);
   A := Length  (X1,Y1,X2,Y2);
   B := Length (X2,Y2,X3,Y3);
   C := Length (X3,Y3,X1,Y1);
   P := (A+B+C)/2;
   writeln ('Площа дорівнює ',Sqrt(P*(P-A)*(P-B)*(P-C)))
end.
```

Приклад 6.4.

```
{ Програма перетворення рядка символів рядка у заголовні літери }
{ Функція використовує  локальні та глобальні ідентифікатори }
program Uper_Char;
Var          Line:  String;
       Count: Integer;
Function Upper(OneChar: Char): Char;
{ Ця функція отримує одну літеру, якщо вона є малою літерою }
{ то повертається прописний еквівалент цієї літери }
Var          UpperCase: String[36];
       lowerCase: String[36];
       TempChar : Char;
       Letter: Integer;
Begin
  UpperCase := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  LowerCase := 'abcdefghijklmnopqrstuvwxyz';
  Letter := Pos(OneChar,LowerCase);
  If Letter = 0 then
     Upper := OneChar
  else
     Upper := UpperCase[Letter]
 End;  {функції Upper}
```

```
Begin       {основна програма }
  WriteLn('Введіть рядок для перетворення до заголовних букв:');
  ReadLn(Line);
  For Count := 1 to Length(Line) do
     Line[Count]:= Upper(Line[Count]);
  WriteLn('Перетворений рядок:');
  Write(Line);
  ReadLn
End.
```

6.3. Рекурсія та випереджаючий опис

Рекурсія – це такий спосіб організації обчислювального процесу, при якому функція по ходу виконання складових її операторів звертається сама до себе (***пряма рекурсія***), тобто обчислюється значення функції через значення цієї самої функції від інших елементів. Якщо ж функція викликає іншу функцію, яка потім викликає першу, то в цьому випадку має місце непряма – ***опосередкована рекурсія***.

Важливо відзначити, що, коли функція викликає сама себе, виконується нова копія цієї функції. При цьому локальні змінні у другій версії незалежні від локальних змінних у першій і не можуть безпосередньо впливати одна на одну.

У рекурсивному визначенні обов'язково має бути присутнім обмеження, тобто *гранична умова*, при виході на яке подальша ініціація рекурсивних звернень припиняється.

Класичним прикладом рекурсії є визначення факторіала. З одного боку, факторіал визначається у вигляді: n! = 1*2*3*...*n, а з іншого боку, його можна визначити наступним чином: n! = {1, якщо n <= 1; (n-1)!*n, якщо n > 1}.

Граничною в даному випадку є умова n <= 1.

Розглянемо програму обчислення факторіалу (приклад 6.5). Програма вводить з клавіатури ціле число N і виводить на екран значення N!, яке обчислюється за допомогою рекурсивної функції FACT. Для виходу з програми необхідно або ввести достатньо велике ціле число, щоб викликати переповнення при множенні чисел з плаваючою крапкою, або натиснути CTRL-Z та Enter.

При виконанні правильно організованої рекурсивної підпрограми здійснюється багаторазовий перехід від деякого поточного рівня організації алгоритму до нижнього рівня послідовно доти, доки, нарешті, не буде отримано тривіальне рішення поставленої задачі. У прикладі 6.5 рішення при N = 0 або 1 тривіальне і використовується для зупинки рекурсії.

Приклад 6.5.

```
{ Програма обчислення факторіалу }
Program Factorial;
```

```
{$S+} {Включаємо контроль переповнювання стеку}
var    n: Integer;
Function Fact(n: Integer): Longint;  {Рекурсивна функція, яка обчислює n! }
begin {Fact}
        if n < 0 then
              WriteLn ('Помилка у завданні N')
       else
              if (n = 0) or (n=1)  then
                    Fact := 1
              else Fact := Fact(n-l)*n
end {Fact} ;
begin {main}
       repeat
              ReadLn (n) ;
              WriteLn (n,'! = ',Fact(n))
       until EOF
end. {main}
```

Пояснимо роботу програми. На час виконання допоміжного алгоритму основний алгоритм (main) призупиняється. При виклику нової копії рекурсивного алгоритму знову виділяється місце для всіх змінних, що оголошуються у ньому, причому змінні інших копій будуть недоступні. При вилученні копії рекурсивного алгоритму з пам'яті вилучаються й усі його змінні. Активізується попередня копія рекурсивного алгоритму, стають доступними її змінні. Нехай необхідно обчислити 4!.

Основний алгоритм: вводиться n = 4, виклик fact(4). Основний алгоритм припиняється, викликається і працює fact(4) : 4<>0 і 4<>1, тому fact:=fact(3)*4. Робота функції припиняється, викликається і працює fact(3) : 3<>0 і 3<>1, тому fact:=fact(2)*3. В даний момент в пам'яті комп'ютера дві копії функції fact. Викликається і працює fact(2) : 2<>0 і 2<>1, тому fact:=fact(l)*2. У пам'яті комп'ютера вже три копії функції fact і викликається четверта. Викликається і працює fact(l) : 1=1, тому fact(l)=l. Робота цієї функції завершена, продовжує роботу fact(2) – fact(2) :=facl(l)*2=l*2=2. Робота цієї функції також завершена, і продовжує роботу функція fact(3). fact(3) :=fact(2)*3=2*3=6.Завершується робота і цій функції, і продовжує роботу функція fact(4). fact(4) :=fact(3)*4=6*4=24. Після цього управління передається в основну програму та друкується відповідь: "4! = 24".

Приклад 6.6.

Дано натуральне число N. Визначити число Y знаків у факторіалі від N.

Для цього завдання можна використовувати формулу Стірлінга для наближеного обчислення факторіалу: $N! := sqrt(2*PI*n)*(n/e)^n$;

Візьмемо десятковий логарифм від правої частини рівності. Після нескладних перетворень отримуємо: A := ln(sqrt(2*PI*n)) + n*ln(n) – n;

Припустимо, що треба знайти кількість цифр цілого Y. Визначити n-1 можна шляхом логарифмування із залишенням цілої частини логарифма. Наприклад, для 123 це буде log123=2,0899, т.е. 2 + 1 = 3). Тому Y = trunc(ln(A)/ln(10))+1.

Отже ми знаходимо A – це ln(n!), де n! обчислено за формулою Стірлінга. Потім

Y := trunc(A/ln(10))+1;

```
program Zn_Fact_1;
var A : Double;
   N, Y : Longint;
begin
     N := 5;
     A := ln(sqrt(2*PI*N)) + N*ln(N) –  N;
     Y := trunc(A/ln(10))+1;
     WriteLn('N= ',N);
     WriteLn('Znakov= ',Y);
     ReadLn
end.
```

Другий варіант з використанням функції Fac з прикладу 6.6.

```
program Zn_Fact_2;
var     N: Longint;
Function Fac(n: Integer): Longint;  { Рекурсивна функція, обчислююча n! }
begin {Fac}
       if n < 0 then
              WriteLn ('Ошибка в задании N')
       else
              if n = 0 then
                     Fac := 1
              else Fac := n * Fac(n-l)
end {Fac} ;
begin
     N := 5;
     WriteLn('N= ',N);
     N := Fac(N);
     WriteLn('N!= ',N);
     N := trunc(ln(N)/ln(10))+1;
     WriteLn('Znakov= ',N);
     ReadLn
end.
```

Розглянемо ще один приклад рекурсивної програми, яка раніше була розглянута на визначення числа: являється число паліндромом чи ні, але не для числа, а для рядка. Також будемо рішати цю задачу, використовуючи рекурсивну функцію.

Приклад 6.7. Потрібно визначити, чи є заданій рядок паліндромом, тобто читається однаково зліва направо і справа наліво. Граничне умова: рядок є паліндромом, якщо він порожній або складається з одного символу.

```
Program Palindrom;
{Рекурсивна функція}
Function Pal(S: String): Boolean;
Begin
        If Length(S) <= 1 Then
           Pal := True
        Else Pal := (S[1] = S[Length(S)]) And
             Pal(Copy(S, 2, Length(S) - 2));
End;
Var S: String;
{Основна програма}
Begin
        Write('Введіть рядок: ');
        ReadLn(S);
        If Pal(S)
          Then WriteLn('Рядок являється паліндромом')
          Else WriteLn('Рядок не являється паліндромом')
End.
```

Рекурсивна форма організації алгоритму зазвичай виглядає витонченіше за ітераційну і дає компактніший текст програми. Але виконується, як правило, повільніше і може викликати переповнення стеку (при кожному вході у підпрограму її локальні змінні розміщуються особливим чином організованої області пам’яті, яка називається програмним стеком). Якщо суворо слідувати правила, згідно з яким кожен ідентифікатор перед його вживанням має бути описаний, то таку програмну конструкцію використовувати не можна. Для того щоб такого роду виклики стали можливі, вводиться випереджаючий опис:

```
Procedure B(j : Byte); forward;
Procedure A(i : Byte);
begin
.....
У (i) ;
.....
```

```
end ;
Procedure B;
begin
.....
A(j);
.....
end;
```

Як бачимо, випереджаючий опис полягає у тому, що оголошується лише заголовок процедури B, а її тіло замінюється стандартною директивою FORWARD. Тепер у процедурі A можна використовувати звернення до процедури B, адже вона вже описана, точніше, відомі її формальні параметри, і компілятор може правильним чином зорганізувати її виклик. Зверніть увагу: тіло процедури B починається заголовком, у якому вже не вказуються описані раніше формальні параметри.

Породження все нових копій рекурсивної підпрограми до виходу на граничну умову називається ***рекурсивним спуском***. Максимальна кількість копій рекурсивної підпрограми, якіи одночасно знаходяться в пам'яті комп'ютера, називається ***глибиною рекурсії***. Завершення роботи рекурсивних підпрограм, аж до найпершої, що ініціювала рекурсивні виклики, називається ***рекурсивним підйомом***.

Розглянемо приклад, в якому показано, що не завжди можна застосовувати рекурсію просто “в лоб” (brute force).

Досить часто на олімпіадах зустрічаються завдання, які провокують до вживання алгоритмів перебору. Але простий підрахунок числа варіантів переконує в неефективності такого підходу. Для розв’язку таких завдань використовується метод (теорія) ***динамічного програмування***, який вперше був використаний у 1940-х роках Р. Беллманом. Суттність його у спрощеному вигляді полягає у тому, що для виявлення розв’язку поставленої задачі вирішується схожа (або схожі), але простіша. При цьому здійснюється перехід до ще простіших і так далі, поки не доходять до тривіальної.

Приклад 6.8. Числа Фібоначчі.

Обчислити N чисел у послідовності Фібоначчі (1, 1, 2, 3, 5, 8, .), в якій перші два члени дорівнюють одиниці, а всі інші є сумою двох попередніх. Нехай N менше 100.

Найочевидніший спосіб розв’язку задачі полягає у написанні рекурсивної функції приблизно такого вигляду:

```
Function F(X:integer):longint;
Begin
   if (X=1) or (X=2) then F:=1 else F := F(X-1)+ F(X-2)
end;
```

При цьому на шостому-сьомому десятку програма “підвісить” найшвидший комп’ютер. Спробуємо розібратися, чому так відбувається?

Для обчислення, наприклад, F(20) ми спершу обчислюємо F(19) та F(18). Причому F(18) ми обчислюємо “кожного разу”, “забуваючи”, що вже обчислили його, коли обчислювали F(19).

Тобто, наша основна помилка у тому, що значення функції при тому самому значенні аргументу обчислюється багато разів. Якщо виключити повторний обрахунок, то функція стане помітно ефективнішою. Для цього доводиться завести масив, у якому зберігаються значення нашої функції:

```
Var D : Array [1..100] of LongInt;
```

Тут спрацьовує **золотий закон програмування** – *виграючи у швидкості, програємо у пам’яті*. Спершу масив заповнюється значеннями, які свідомо не можуть бути значеннями нашої функції (найчастіше, це “нуль”). При спробі обчислити якесь значення, програма з’ясовує, чи не обчислювалося воно раніше, і якщо так, то бере готовий результат.

Функція набирає наступного вигляду (не вірте, будь ласка, книгам, що стверджують, що шукати числа Фібоначчі рекурсивно не можна в принципі – можна, якщо відсікання робити з розумом):

```
Function F(X : integer): LongInt;
Begin
  if D[X]= 0 then
       if (X=1) or (X=2)
       then D[X]:= 1
       else D[X]:= F(x-1)+ F(x-2);
  F := D[X]
End;
```

Цей підхід у динамічному програмуванні (метод розв’язку завдань з оптимальною підструктурою та підзадачами, що перекриваються, який набагато ефективний, ніж розв’язання “у лоб”) називається підходом “зверху вниз”. Він запам’ятовує розв’язані задачі, але черговість розв’язання задач все одно контролює рекурсія.

На цьому вже можна зупинитися, але можна ще більше спростити розв’язання, прибравши рекурсію взагалі. Для цього необхідно змінити спадаючу логіку міркування на висхідну (відповідно навпаки). У цьому завданні такий перехід очевидний і описується простим циклом:

```
D[1]:= 1; D[2]:= 1;
For i := 3 to X do
        D[i]:= D[i-1]+ D[i-2];
```

Тут використаний підхід “знизу догори”. Найчастіше такий спосіб рази у три швидший. Проте у ряді випадків такий метод призводить до необхідності розв’язати більшу кількість підзадач, ніж при рекурсії.

Дуже часто для його написання доводиться використовувати як проміжний результат низхідну форму, а інколи безрекурсивна (ітеративна) форма виявляється надзвичайно складною і малозрозумілою.

Загальна порада така: використовуйте і тестуйте рекурсивну форму, а перероблення займайтеся, якщо ваша програма перевищує відведений їй час на “великих” тестах.

Контрольні запитання для самоперевірки

1. Які існують види підпрограм та чим вони відрізняються?
2. Який порядок доступу до змінних у підпрограмах?
3. З яких частин складається опис процедури та функції?
4. Які існують форми передачі параметрів у процедури та функції?
5. Що являє собою рекурсія?
6. Які існують види рекурсії?
7. Як здійснюється рекурсивний обрахунок?
8. Що є випереджаючим описом процедури (функції) та для чого він потрібний?
9. У чому полягає підхід динамічного програмування?

7. Чи є ви хорошим програмістом?

Прийшов час перевірити вас на предмет того, як ви засвоїли пройдений матеріал у сенсі написання хороших програм, чи виробили ви певний стиль при написанні програм, а також дати декілька корисних порад (прийомів або методів), які використовують досвідчені програмісти, щоб отримати надійні, ефективні, зручні для застосування та зрозумілі програми. Давайте коротко розглянемо, чим відрізняється професійний підхід до створення програм від аматорського?

По-перше, аматор (новачок) не приділяє належної уваги етапу проектування ПЗ, а одразу приступає до кодування "з голови". Це неминуче призводить до появи безлічі помилок та витрат часу на перекомпонування програми.

По-друге, аматор пише "для себе" і при цьому, як правило, не піклується про оформлення тексту програми. Іменам об'єктів присвоюються випадкові і односкладові імена, не використовуються відступи та коментарі. Таку програму важко читати сторонній людині, а згодом – і авторові. Нарешті, результат роботи аматора використовується один раз, для іншого завдання доводиться або все писати знов, або корожити текст наявної програми. Це призводить до засмічення каталогів розробника, появи безлічі незв'язаних версій однієї і тієї ж програми, втрати часу на пошуки та даремне редагування.

Наше завдання полягає не лише у тому, аби вивчити синтаксис деякої мови програмування, але і засвоїти в якійсь мірі професійний підхід. Як показує практика, аматори неохоче розлучаються зі своїми звичками. Але шлях цей необхідний.

7.1. На кого слід орієнтуватися при розробці програми

Коли ви пишіте лист або статтю, то, ймовірно, прагнете будувати виклад просто і відповідно до правил граматики і синтаксису, щоб досягти легкості сприйняття написаного. Іншими словами, гарний стиль викладу допомагає нам швидко досягти розуміння.

Одна з фаз розробки програми пов'язана з її написанням. Крім програміста-розробника більшість програм переглядаються та використовуються іншими людьми. Отже, програміст, який пише заплутано, не дотримується дисципліни програмування, схожий на автора витіюватих фраз. При написанні програми слід враховувати думку трьох осіб. Перш за все, думка людини, яка буде переглядати та використовувати вашу програму. Користувач програми бажає, щоб інформація представлялася у зрозумілій та легко засвоюваній формі.

Другою особою, думку якої необхідно враховувати, є програміст, який захоче ознайомитися з вашою програмою. Це може виникнути з кількох причин, наприклад для того, щоб використовувати у своїй програмі алгоритм,

який розроблений у вашій програмі, або щоб адаптувати вашу програму до своїх вимог. У будь-якому випадку авторові, який написав зрозумілу та добре створену програму, будуть вдячні.

Сам автор є третьою особою, яка повинна братися до уваги при розробці програми. Програміст повинен мати можливість прочитати, принаймні, власну програму, наприклад, повернувшись з відпустки. Тому слід із самого початку привчатися до хорошого стилю програмування.

Цілі, що відмічені вище, можуть показатися очевидними, проте у реальному житті вони не завжди легко досяжні. Основне завдання, яке полягає у написанні хорошої програми, вимагає для свого розв'язку серйозного, продуманого ставлення. При розробці програми часто виникає спокуса привнести ясність в жертву незначному підвищенню ефективності. На щастя, мова Pascal володіє гарними засобами для проектування зрозумілих програм. Проте, слід пригадати "аксіому" програмування: "**Немає, не було і, швидше за все, не буде мови програмування, яка, хоча б у щонайменший мірі, гарантувала нас від написання поганих програм**". Найкращі засоби у невмілих руках можуть дати ефект, прямо протилежний очікуваному. Звідси стає зрозумілим, яке важливе значення має стиль написання програм (або культура програмування).

Під стилем, у даному випадку, мається на увазі набір прийомів або методів програмування, які використовують досвідчені програмісти, щоб отримати правильні, ефективні, зручні для застосування та легко зрозумілі програми. Тому запропоновані тут "стандарти" стилю є результатом здорового глузду та досвіду програмістів, а не правилами, які зафіксовані раз та назавжди.

7.2. Що рекомендується і що не рекомендується гарним програмістам

Поради, що слід робити і чого слід уникати, не мають на меті поставити програміста у жорсткі рамки. Ці правила допоможуть вам мінімізувати витрати часу при написанні досконалих програм. Не дивлячись на те, що деякі з цих правил (порад) можуть здатися вам непотрібними та нудними, але дотримання їх у процесі створення складної програми значно полегшує і, головне, прискорює її відлагодження.

1. При розв'язку будь-якої задачі насамперед ви *обираєте (розробляєте) алгоритм*, а потім – часом вже у процесі кодування – деталізуєте його. Не є таємницею, що багато кроків алгоритму можуть стати зрозумілими лише тоді, коли велика частина програми вже написана так, щоб легко було вносити зміни до її тексту, уточнення та поправки. На крайній випадок, ви будете вимушені повністю відмовитися від обраного алгоритму та перейти до реалізації іншого, більш зрозумілого. При цьому вся виконана робота по написанню тексту програми виявиться марною.

2. Створення будь-якої програми (за обраним алгоритмом розв'язання) треба починати з опису змінних. Зрозуміло, що неможливо одразу ж передбачити всі змінні, які можуть знадобитись у ході створення

програми, проте найважливіші, – ті, заради яких уся робота і починається, – мають бути описані з самого початку.

3. Правильний вибір імен змінних – це легкість для читання програми. Кращим способом передати сенс змінної або символьної константи є присвоєння їй імені, яке відображає призначення за сенсом змінної або константи. У подальшій роботі це позбавить автора програми або її користувача від необхідності кожного разу напружувати пам’ять для з’ясування, що ж цей ідентифікатор означає. У той же час обране ім’я не має бути дуже довгим. Списки ідентифікаторів у блоках опису слід також впорядковувати – це полегшує пошук в них потрібних елементів.

Імена змінних повинні бути обрані так, щоб найкращим чином визначати ті величини, які вони відображають. Наприклад, в операторі X := Y + Z імена змінних обрані невдало, оскільки зовсім не використовується мнемоніка. Такий запис оператора

PRICE := COST + PROFIT; набагато краще.

Використовуйте імена з відповідною мнемонікою. Мнемоніка – це мистецтво запам'ятовування, засноване на законах асоціацій. Не слід використовувати слова, в яких зазвичай робляться орфографічні помилки; імена, що розрізняються тільки однією буквою; слова, що мають більше одного очевидного скорочення; ключові слова мови програмування.

Використовуйте розподільник (звичайно це “_” – підкреслення) у складній мнемоніці для змінних, що полегшує читання імен змінних:

COST_DETAIL

Проте не варто заводити дуже довгі (більше 6-7 символів) імена, оскільки це загальмує написання програми. Варто використовувати скорочення, короткі синоніми, російські (українські) слова (за сенсом та прочитанням, проте латинські за написанням).

Імена, які розрізняються лише одним символом, слід вважати вкрай невдалими. Проте в усьому потрібно знати міру і, зрозуміло, шість допоміжних змінних, що відображають шість послідовних станів одного і того ж процесу, варто назвати a1, a2 ... a6. Тим самим ви підкреслите їх єдине походження.

При виборі імен змінних варто також користуватися інтуїтивно знайомими вам умовностями. Наприклад, *i, j* та *k* зазвичай служать допоміжними лічильниками, або керуючими змінними в операторах циклу, *m* та *n* найчастіше зберігають розмірність, а *x* та *y* – координати. Краще не відступати від звичних вам скорочень та позначок, навіть якщо вони і не є загальноприйнятими.

4. Імена функціям та процедурам також потрібно давати смислові, причому тут обмеження на довжину набагато м’якші: до 10-15 символів. Навряд чи ваша програма звертатиметься до процедур та функцій настільки ж часто, як до змінних. Тому тут на перший план виходить наповнення імен підпрограм поясненням виконуваних ними операцій. Краще не полінуватися і набрати два-три рази, наприклад **Raschet_zatrat**, ніж плутатися під час відлагодження між **Raschet_z** і **Raschet_p**. Найзручніше, якщо імена підпрограм відповідають виконуваним ними крокам алгоритму тієї чи іншої

міри деталізації: у цьому випадку легко виявити помилки процесу кодування та розробки (неправильний порядок операцій, передчасна зупинка тощо).

5. Не вводьте зайвих змінних. Розділ опису повинен містити у собі лише ті змінні, які потім зустрічатимуться у тексті програми. Ця вимога викликана міркуваннями економії, причому не стільки місця у пам'яті, скільки ваших зусиль по виявленню незрозумілих помилок.

6. Не рекомендується використовувати одну і ту саму змінну для кількох цілей. Описавши змінну, можна піддатися спокусі отримати додаткові вигоди від її використання для нових цілей в іншій частині програми, наприклад, для економії пам'яті. Чи слід користуватися такою економією? Звичайно ж ні.

7. Прокоментуйте усі змінні. Розуміння даних – ключ до розуміння програми. Кожна пропозиція, що оголошує деяку змінну, повинна супроводжуватися коментарем, який пояснює сенс цієї змінної.

8. Уникайте імен змінних, які схожі на зарезервовані ключові слова мови програмування.

9. Робіть коментарів більше, ніж це вважається за необхідне. Хороші коментарі написати непросто. Оскільки їхня мета – полегшити розуміння програми, – вони повинні бути так само добре продумані, як і кодування програми. Якщо ви зазнаєте труднощів при складанні коментарів для опису того, що ви робите, то швидше за усе ви "не усвідомлюєте, що створюєте".

Не слід вважати, що якщо ви є автором програми, то жодних труднощів щодо її перегляду під час відлагодження у вас не виникне. Неможливо вивчити програму, яка безперервно змінюється настільки, щоб орієнтуватися у ній "із заплющеними очима", тому не скупіться на короткі, але ділові коментарі.

Вступні коментарі (на самому початку програми) зазвичай містять: номер та ім'я програми (модуля); прізвище автора; дату, номер версії; призначення програми; перелік основних алгоритмів з посиланнями на джерела; імена підпрограм, що викликаються програмою; опис введення-виведення.

Кожна підпрограма повинна також починатися з коментарів, які пояснюють, що вона робить. Поясненнями треба супроводжувати ті частини програми, які важко зрозуміти без коментарів. Вважається, що програма на мовах високого рівня легко читається і як би себе документує. Часто логіка одного програміста – автора програми – не очевидна для іншого.

10. Пропуск рядків. Пропуск рядків – часто недостатньо оцінюваний метод поліпшення наочності програм. Як у природній мові ми користуємося пропуском рядків для відокремлення параграфів, так само у програмі необхідно розділяти її окремі фрагменти. Пропуском одного рядка можна відокремлювати кожну групу логічно пов'язаних операторів – це полегшує пошук окремих частин у програмі.

11. Не рекомендується розміщувати декілька операторів в одному рядку. У мові Pascal припускається розміщення на одному рядку декількох операторів. Проте слід уникати такої практики, оскільки це ускладнює розуміння структури програми. Звичайно, припустимі і деякі відхилення.

Наприклад, декілька коротких схожих операторів цілком можна записати в один рядок, розділяючи їх хоча би одним пробілом:

```
a:=0;  b:=0;  c:=0;
```

12. Ставте пропуски для покращення перегляду програм. Пропуски слід ставити скрізь, де це призводить до покращення перегляду програм. Можна написати такий оператор: If Count<Lines then

Але наступний оператор переглядати значно легше: If Count < Lines then

13. Перенесення. У багатьох мовах програмування припускається перенесення оператора в інший рядок. Якщо оператор не поміщається на поточному рядку, починайте його з наступного рядка. Складні оператори також краще розміщувати на кількох послідовних рядках. Тоді при *покроковому прогоні* легко буде локалізувати помилку. Якщо ви все ж таки хочете перенести частину оператора на наступний рядок, робіть перенесення після знаку операції. Ось випадок недбалого перенесення:

```
A: = B – C
   –  (D + 2);
```

Набагато краще перенести так:

```
A: = B – C –
    (D + 2);
```

У другому прикладі знак мінус на першому рядку, повідомляє читача про те, що оператор продовжується на наступному рядку. Крім того, якщо другий рядок випадково буде викинутий, то в першому прикладі це пройде безслідно, тоді як у другому буде виявлена синтаксична помилка.

14. Одного оператора у рядку достатньо. Мова Pascal припускає розміщення кількох операторів в одному рядку. Наприклад

```
X := A*3;  if  A < B then  Begin B := cos(C);  A := X + 1 End;
```

Проте запис кількох операторів на одному рядку незручний з двох причин. По-перше, важко переглядати програму. По-друге, це заважає використанню таких засобів, як ділення на параграфи. Краще розміщувати кожний оператор на окремому рядку.

```
X := A*3;
if  A < B then Begin
     B := cos(C);
     A := X + 1
End;
```

Ще одна з причин розміщення операторів на окремому рядку полягає у тому, що у повідомленнях про синтаксичні помилки завжди вказується номер рядка.

15. Дужки обходяться дешевше, ніж помилки. Правильне використання дужок істотно покращує легкість для читання програми. Оскільки послідовність виконання операцій визначається їх пріоритетами, то програміст часто може обмежитися невеликою кількістю дужок, але при цьому порушується читання і корегування програм. Це ілюструє такий приклад :

Мала кількість дужок *Використання додаткових дужок*

A*B*C/(D*E*F) (A*B*C)/(D*E*F)

Основне правило таке: в сумнівних випадках ставте дужки. Це не тільки робить програму зрозумілішою, але і запобігає помилкам.

16. Рекомендується використовувати щаблеву форму запису програми. У даному посібнику ми в усіх прикладах програм використовуємо зсув так, щоб легко розпізнавати кожен рівень вкладення. Відступи не впливають на логіку програми, але суттєво покращують її наочність, прояснюють її логіку і полегшують її читання. Правильно зроблені відступи виявляють структуру програми.

Текст програми без відступів:

```
If (I <= 1) then FACTORIAL := 1 else begin FACTORIAL := 1;
For J := 2 until I do FACTORIAL := FACTORIAL * J
End;
```

Текст програми з відступами:

```
If (I <= 1) then
    FACTORIAL := 1
else
    begin
        FACTORIAL := 1;
        For J := 2 until I do
            FACTORIAL := FACTORIAL * J
    End;
```

Умовний оператор, цикли – типовий випадок використання відступів. Відступи при вкладених умовних операторах та операторах циклу допомагають визначити початок та кінець цих операторів.

Продовження описів та операторів на нові рядки також треба зрушувати праворуч. Але по можливості слід уникати довгих рядків.

Виняток з цього правила становлять лише оператори налагоджувального друку, які потім будуть вилучені з остаточного варіанту програми. Їх краще за усе починати з першої позиції рядка, незважаючи на відступи поточного блоку. Так вони сильніше виділятимуться.

17. Бажано, аби кожен оператор *if* мав не лише *then*-, але і *else*-*гілку*, навіть якщо вона залишиться порожньою. Це дозволить вам уникнути двозначних структур, таких як

```
if ...
   then if ...
           then ...
   else ...
```

яка насправді сприймається компілятором абсолютно інакше:

```
if ...
   then if ...
           then ...
           else ...
```

Такі помилки найбільш небезпечні при видаленні з тексту вже правильно працюючої програми *налагоджувальних операторів*. Наприклад, якщо повністю вилучити з конструкції

```
if ...
  then if ...
         then ...
         else <один налагоджувальний оператор>
  else ...
```

нібито непотрібний четвертий рядок, то зберегти працездатність програма не зможе. Добре ще, якщо вам пощастить і результат або взагалі не буде виданий, або виявиться настільки неправдоподібним, що не відмітити цього дивацтва буде неможливо. Інакше вийде нібито правильна програма, яка насправді не є правильною.

Порятунком від такої прикрості можуть послужити і додаткові операторні дужки – потрібно лише стежити, аби вони не дуже засмічували текст блоку:

```
if ...
 then begin if ...
        then ...
        else <один налагоджувальний оператор>
      end
 else ...
```

Якщо тепер ми вилучимо той рядок, жодної шкоди програмі це не нанесе.

18. Рекомендується писати спонукаючі повідомлення зрозумілою мовою. Повідомлення, які програма використовує для запрошення введення різної інформації, називаються ***спонукаючими повідомленнями***. Слід уважно підходити до складання спонукаючих повідомлень, як, втім, і будь-якого виведення для зменшення імовірності їх неправильного тлумачення.

19. Не змінюйте значення параметра циклу у тілі циклу. Зміна параметра циклу всередині циклу ускладнює і розуміння циклу, і доказом цього може стати, що він може стати “нескінченним”.

20. Користуйтеся змінними-константами. Не використовуйте у програмі числових величин, якщо вони використовуються кілька разів, краще користуйтеся змінними-константами:

```
LENGTH_CIRCLE := 2 * 3.14159 * RADIUS;
```

Краще

```
Const   PI = 3.1415926;
Var     RADIUS :  Real;
          . . .
LENGTH_CIRCLE := 2 * PI * RADIUS;
```

21. Мікроефективність. До якнайгірших порушень хорошого стилю програмування відносяться численні "поліпшення" програми задля досягнення ефективності, особливо за рахунок різних трюків. Це не тільки ускладнює читання програми, але і знижує її надійність. Звідси і назва "Мікроефективність".

Ті програмісти, які займалися продуктивністю систем, знають, що ефективність досягається в результаті ретельного аналізу структур даних та алгоритмів, але не за рахунок “тривиальностей” типу: заміна 3*X на X+ X + X (у деяких випадках це зробить оптимізуючий компілятор).

Не оптимізуйте передчасно. Якщо програма відлагоджена і працює правильно можна зайнятися її ефективністю. Як сказав Брайан Керниган, спочатку змусьте програму працювати, потім змусьте програму працювати швидко. Спочатку пошук сумнівних ділянок програми, а потім оптимізація на макроефективному рівні. **Набагато простіше зробити коректну програму швидкою, ніж швидку – коректною!**

22. Ефективні алгоритми. У деяких випадках для оптимізації критичних ділянок програми можна зайнятися ефективністю. Наведемо такий простенький приклад. Ви вже знаєте, що в Pascal відсутня можливість піднесення до ступеня, не враховуючи квадрата. Тому для отримання a^{20} потрібно помножити a*a*a*a...*a 19 разів. Але якщо врахувати, що результат множення можна зберегти в проміжній змінній, відповідь можна знайти за 5 дій.

```
Var A, B: real;
Begin
   Write(‘Введіть число’);
   Readln(A);
   B:=a*a;      {отримуємо A в 2}
   B:=B*B;     {отримуємо A в 4}
   B:=a*b;      {отримуємо A в 5}
   B:=B*B;     {отримуємо A в 10}
   B:=B*B;     {отримуємо A в 20}
   Writeln(‘A в 20 степени=‘,B:0:4);
   Readln
End.
```

Замість вказівки ширини поля для виведення числа в операторі Writeln(‘A в 20 ступені=‘,B:0:4) краще ставити 0, тоді компілятор відведе стільки позицій під цілу частину, скільки насправді потрібно, а для дробової частини ми порахували можливим залишити 4 позиції.

23. Простота – запорука успіху. Іноді пишуть складні програми з метою довести, що без автора цих програм обійтися не можливо. Дотримуйтеся в програмуванні принципу KISS (Keep It Simple, Sidney - Роби це простіше, Сідней). Простота операторів допомагає виявити помилку у програмі як при синтаксичному контролі, так і у процесі її виконання. Якщо компілятор повідомив, що виявлена синтаксична помилка у 20-у рядку, то при записі простого оператора виявити її легко. Але якщо оператор дуже довгий, або у цьому рядку знаходиться декілька операторів, то пощук помилки стає нелегкою справою. Але не слід вдаватися до іншої крайності. Так, наприклад, цей оператор

ROOT1 := (-B + SQRT(B*B – 4*A*C))/(2*A); легший для сприйняття, ніж група операторів

```
BB := B*B;
A1 := A*C;
A2 := 4*A1;
B1 := BB – A2;
B2 := SQRT(B1);
BOTTOM := 2*A;
TOP := -B + B2;
ROOT1 := TOP/BOTTOM;
```

Розглянемо ще один приклад – **обмін значеннями.** Необхідно поміняти значеннями вміст змінних A та B, тобто написати програму, схожу на цю:

A := B;

B := A;

Але ця програма працювати не буде (у обох змінних буде значення B).

Тепер спробуємо знайти правильне рішення. Позначимо початкове значення A за A1, B за B1. Тоді необхідно, щоб після закінчення роботи програми A дорівнювало B1, а B – A1.

Найпростішим, наочнішим та найбільш природним способом є алгоритм обміну за допомогою допоміжної змінної (temp) такого ж типу, як A та B.

```
Var a, b, temp: integer;
Begin
   Write(‘введіть два числа ‘);
   Readln(A,B);
   temp := B;
   B := A;
   A := temp;
   Writeln(‘A=‘,A, ‘B=‘,B);
   Readln;
End.
```

А зараз розглянемо один з прикладів програми, дивлячись на який не одразу можна здогадатися, що саме робить ця програма. Такі приклади даються, як приклади для розвитку мислення програміста, **але їх ні в якому разі не можна застосовувати при написанні практичних, простих, наочних програм**.

Хай у змінних A та B є якісь значення, позначимо їх A1 і B1 відповідно (A=A1; B=B1).

1) Занесемо у змінну A результат додавання A та B (A := A + B):
   A = A1 + B1; B = B1;
2) Занесемо у змінну B різницю A і B (B := A – B):
   оскільки A = A1 + B1; то B=(A1+B1) – B = A1;
3) Занесемо у змінну A різницю A та B (A := A – B):
   A = B1; B = A1;

А зараз запишемо код програми, в якій ми “заощадили пам’ять", відведену раніше під змінну temp, але втратили в наочності та в ефективності (додалися зайві арифметичні операції):

```
Var a, b: integer;
Begin
   Write(‘введіть два числа ‘);
   Readln(A,B);
   A:=A+B;
   B:=A-B;
   A:=A-B;
   Writeln(‘A=‘, A, ‘B=‘,B);
   Readln;
End.
```

Відмітимо, що така програма у деяких випадках може працювати неправильно, якщо A і B мають тип даних real. Також можлива неприємна ситуація і для цілочисельних змінних, якщо вони набувають дуже великих значень (при додаванні можливе **переповнення**). Але, жодного переповнення не станеться, якщо замінити операцію арифметичного додавання на побітову операцію XOR (додавання за модулем 2). Можете у цьому переконатися.

```
var       a,b: integer;
begin
     write(‘Введите a и b:  ‘);      readln(a,b);
     a := a xor b;       {порозрядне виключаючє ИЛИ,}
     b := a xor b;       {якщо його виконати другий раз,)
     a := a xor b;       {то знову набудемо первинного значення}
     writeln(‘a=‘,a,’ b=‘,b);
     readln;
end.
```

24. Не забувайте в доброзичливій манері, але без зайвого багатослів'я, запросити користувача ввести вхідні дані і вказати порядок їх введення. У деяких випадках може знадобитися вказати тип величин.

25. Намагайтеся не використовувати одну й ту ж змінну для зберігання різних значень. Це заплутає як Вас, так і людей, які будуть працювати з Вашим кодом надалі.

26. Не потрібно використовувати значення параметра циклу після його завершення. У багатьох випадках змінна параметра циклу – локальна і після завершення циклу пам'ять, що виділяється під неї, буде звільнена і в ній може міститися сміття. У деяких випадках «розумний» компілятор може видати помилку.

27. Уникайте у циклах обчислень, які не залежать від їх параметрів! Вираз типу sin(Pi/n), поміщений у цикл, де n не міняється, виглядає безглуздо. Адже кожне обчислення синуса (так само, як й інших стандартних функцій) – це достатньо витратне розкладання у ряд Маклорена, яке виконується машиною. У подібних випадках такі константі вирази треба виносити за межі

циклу, присвоювати їхнє значення тимчасовим змінним та використовувати такі змінні у циклі.

28. Передавайте значення підпрограмам переважно *за адресою*, а не за значенням. Для матричних та векторних даних намагайтеся робити це завжди.

29. Не робіть підпрограми залежними від глобальних даних. Недотримання цього правила істотно зменшить імовірність повторного використання коду вами або іншим розробником.

30. Програмування згори-донизу. У процесі розробки алгоритму та програми слід починати з найзагальнішої моделі розв'язання, поступово уточнюючи її до рівня окремого блоку, і потім детально опрацьовувати кожен блок. Детальніше метод програмування згори-донизу ми розглядатимемо в окремому розділі.

31. Один об'єкт – одна задача. Концентруйтеся одночасно лише на одній проблемі. Кожен об'єкт (змінна, функція, простір імен) повинні розв'язувати одне, точно поставлене завдання. Із зростанням об'єктів збільшується область їх відповідальності, але вони не повинні відхилятися від свого призначення.

Об'єкти з різнорідними функціями зазвичай важкі для проектування та реалізації. Переважно використовуйте короткі функції з чітко вказаним єдиним призначенням для розв'язку одного конкретного завдання.

Уникайте глибокої вкладеності. Надмірно вкладені блоки коду частенько перешкоджають реалізації принципу "одна функція – одна задача", і зазвичай ця проблема розв'язується кращим розподілом завдання на окремі частини.

32. Не покладайтеся на надійність техніки, робіть час від часу збереження написаного. Просте натиснення клавіші **F2** через 5-10 хвилин може врятувати від багатьох неприємностей в разі "зависання" завдання, не кажучи вже про раптове відключення електроенергії. Копіюйте свій код принаймні на три носіях, що знаходяться фізично в різних місцях – на випадок псування носія, пожежі, пограбування тощо.

33. Одна голова добре, а дві – краще. Регулярно переглядайте код всією командою. Чим більше очей – тим вище якість коду. Покажіть ваш код іншим та ознайомтеся з їх кодом – це принесе користь усім. Регулярне рецензування коду іншими членами команди приносить свої плоди:

- підвищення якості коду при доброзичливій критиці іншими;
- поліпшення якості дизайну та реалізації шляхом обміну ідеями;
- швидке навчання нових членів команди;
- підвищення рівня довіри, професійної гордості та співпраці у команді.

34. Регулярно читайте статті про сучасні засоби розробки, застосовуйте усі новинки у своїй області програмування, і, **головне – застосовуйте їх.** Інформації в області програмування в Інтернеті – море! Підручники, документація, статті – усе доступно і, при тому, безкоштовно.

35. Дисциплінованість та стресостійкість. Дедлайни (від англ. Deadline – крайній термін (дата та/або час), до якого має бути виконане завдання) – невід'ємна частина роботи програміста. Якщо Ви нервуєте, то навіть маленька

складність починає здаватися величезною проблемою. А головне, можна впритул не помічати простого рішення. Спокій, тільки спокій.

36. Ніколи не опускайте руки! Практично ніколи, нічого і ні у кого не виходить зробити з першого разу. Пам'ятайте, що якщо Вам важко, Ви на правильному шляху! Досягти мети можна тільки переступивши через свої лінощі та тягу до комфорту. Але якщо Ви зможете зробити це, Вам відплатиться багаторазово!

7.3. Використання коментарів для пояснень

Одним із засобів досягнення легкості розуміння програми є включення у програму коментарів. Коментар – це текст, який вставляється у програму для пояснення певних аспектів кодування. Коментарі призначені для легшого розуміння програми людиною, яка з нею знайомиться. Розглянемо приклад програми з коментарями.

Приклад 7.1.

{ Обчислити дохід за терміновим вкладом з урахуванням складного} {річного відсотка }

```
program Procent;
Var          Summa: Real;      {Сума грошей}
       Rost:  Real;        {Щорічне зростання прибутку}
       Time:  Integer;     {Кількість років}
Begin
  {Запит величини основного капіталу, відсотка та терміну}
  WriteLn('Яка величина основного капіталу?: ');
  Read(Summa);
  WriteLn('Яка частка приросту у відсотках?: ');
  Read(Rost);
  Rost := Rost/100;         {Перетворення % у десяткове значення}
  WriteLn('За скільки років виплачуються відсотки?: ');
  ReadLn(Time);
  { Обчислення остаточного результату }
  While Time > 0 do
    Begin
      Summa := Summa*(1+Rost);
      Time  := Time -  1
    end;
  WriteLn('Остаточна сума грошей = ', Summa:10:2);
  ReadLn
End.
```

Приклад 7.2.

{Дано натуральне число N. Визначити, чи є воно досконалим. }

```
{Досконале число N дорівнює сумі усіх своїх дільників, що не перевершують
само N.}
{Наприклад, число 6 = 1+2+3}
Var  n,i,sum:word;
Begin
    Write('Введіть натуральне число: ');
    Readln(n);
    Sum:=0;
    for i:=1 to n div 2 do
        if n mod i = 0 then
            sum:=sum+i;
            if sum=n then
        writeln('число ',n', досконале')
           else writeln('число ',n', не досконале' );
         Readln
      End.
```

7.4. Створення дружнього інтерфейсу

Призначений для користувача ***інтерфейс*** **–** це забезпечення взаємодії програми та людини. Гарним вважається **дружній інтерфейс** – той, який зручний не програмістові, а користувачеві. Тепер, коли у нас вже є всі інструменти, які дозволяють створювати гарні *інтерфейси* до програм, прийшов час поговорити про них детальніше також у вигляді порад та побажань.

1. Створюйте у своїх програмах заставку. Насамперед, одразу після запуску ваша програма повинна повідомити користувача, яку саме задачу вона збирається розв'язувати. Причому інформація про завдання, що розв'язується, має бути вичерпною. Це особливо важливо, якщо у постановці задачі є серйозні обмеження, про які одразу ж треба повідомити користувача.

Інформаційна частина інтерфейсу, яка з'являється на екрані одразу після запуску програми, називається ***заставкою***. Заставка може містити:

- назву програми;
- пояснення (коротку або детальну інформацію про завдання, що розв'язується);
- інформацію про автора програми;
- номер версії програми і таке інше.

Заставка може складатися як з окремого екрану, який зникає після натиснення довільної клавіші (його змінює робоча область програми), так і лише з одного рядка, який залишається на екрані до кінця роботи програми (або доки її не витіснить об'ємне виведення). Наприклад:

Сортування масиву (до 100 елементів) методом вставки.

Знаходження найкоротшого шляху у зв'язному графі.

2. Введення інформації. Користувач може вводити інформацію двома способами: **вільним введенням** або **вибором** із наданих можливостей.

Вільне введення інформації може бути потрібно, наприклад, при запиті імені файлу, що зберігає будь-які об'ємні дані. Можна також просити користувача ввести своє ім'я або декілька невеликих чисел. Нагадаємо, що введення великих об'ємів інформації (наприклад, таблиць або матриць) бажано організовувати через файли. Інакше у багато разів зростає ймовірність помилки і, як наслідок, необхідність програмувати додаткові блоки, що дозволяють ці помилки виправляти.

3. Використовуйте запрошення для введення. Кожного разу, коли програма чекає вільного введення від користувача, вона повинна повідомляти про це, виводячи на екран запрошення ***для введення.*** Запрошення вигляду "Введіть x:" неможливо вважати задовільним, оскільки воно не містить жодної інформації про очікувані дані. Гарне запрошення повинно повідомляти користувача, що саме від нього хочуть отримати у даний момент: тип, формат і розмір даних, що вводяться. Наприклад:

Введіть координати центру кола (два цілі числа -100 <= x, y <= 100):

4. Передбачайте перевірку будь-якої введеної користувачем інформації. При вільному введенні користувач, власно кажучи, може вводити що завгодно і зовсім не обов'язково інформацію в очікуваному програмою форматі. А у мові Pascal, як ми вже знаємо, неприпустимі невідповідності типів даних. Наприклад, якщо не відключений контроль введення/виведення, то спроба ввести букву "O", коли очікується цифра "0", призведе до аварійної зупинки програми. Ще складніше буває розібратися з форматом дійсних чисел (часто замість десяткової крапки користувачі ставлять звичну для нас кому) тощо.

Таким чином, для надійності роботи програми необхідно передбачити перевірки будь-якої введеної користувачем інформації. Такий контроль отримав у середовищі програмістів не дуже ввічливу назву "***Захист від дурня***" (fool proof). При правильній організації цього захисту ваша програма "не зламається" навіть у тому випадку, якщо замість введення даних користувач просто сяде на клавіатуру.

Якщо з будь-яких причин введення все-таки вийшло помилковим, повідомлення про це повинно з'явитися негайно, разом з пропозицією ввести інформацію знов. Крім того, навіть у разі правильного введення корисно одразу ж надрукувати значення, яке отримала змінна.

5. Виведення інформації. Кожного разу, коли програма приступає до виконання об'ємного блоку робіт, вона повинна повідомляти про це користувачеві, а ще краще, якщо на екрані буде знаходитись який-небудь індикатор **процесу**.

Програма також повинна повідомляти про результати своєї роботи. Проте "голе" виведення є неприйнятним: результати обов'язково повинні супроводжуватись вичерпними поясненнями. Далі, окрім остаточного результату, треба виводити і проміжні результати – після закінчення обробки

кожного логічно самостійного крупного блоку. Якщо ж робота програми виявляється перерваною через якісь помилки, повідомлення про її причини повинно з’явитися на екрані.

7.5. Підведемо підсумки

Резюмуючи, можна зауважити, що усі великі програми на світі написані тільки за рахунок одного – правильної структуризації коду. І хороший програміст – це не той, хто "знає мови", а той, хто вміє створювати легко зрозумілий та модифікований код, максимально використовуючи стандартні засоби мови розробки.

Написання красивого та ефективного програмного коду – ціле мистецтво, багато у чому, на жаль, забуте у зв’язку з вибуховим ростом потужності обчислювальних пристроїв, що викликав зниження вимог до якості алгоритмів та програм. Цей невеликий перелік корисних порад (прийомів та методів), які використовують досвідчені програмісти, щоб отримати надійні, ефективні, зручні для застосування та зрозумілі програми, не може замінити вивчення спеціалізованих дисциплін на кшталт "Технології програмування", але успадкування викладених тут прийомів дозволить програмістам-початківцям звикати не просто "вирішувати завдання", а робити це по можливості елегантнішим та економічнішим, з точки зору обчислювальних витрат, способом.

У підсумку хотілось би привести висловлювання Бьерна Страуструпа у його интервью. На питання: чи **можете ви щось порадити програмістам-початківцям?** Він відповів:

Вивчайте основи програмування: алгоритми, архітектуру машин, структури і т.і. Не копіюйте сліпо підходи і методи з одного додатку в інший.

Ви завжди повинні знати, що ви робите, бути упевненими, що ваша програма працює, і твердо знати, чому вона працює. Не думайте, що ви можете передбачити, якою буде індустрія програмування через 5 років і чим саме доведеться займатися вам, тому вчитеся загальнішим і кориснішим прийомам і підходам. Намагайтеся писати код, який краще, код, який більше відповідає вашим принципам програмування.

Працюйте так, щоб програмування більшою мірою було професійною діяльністю, а не низькорівневим «хакерством» (програмування – це і ремесло, але не лише ремесло). Вчитеся на класиці в області розробки і за допомогою кращих книг, не потрібно покладатися на "how to" і документацію в онлайні – вона недостатньо глибоко порушує питання програмування.

Контрольні запитання для самоперевірки

1. Як правильно обрати потрібний алгоритм?
2. Як краще визначити імена змінних, констант та функцій і процедур?
3. Для чого потрібні коментарі?

4. У чому полягає структуризація тексту програми?
5. Які існують прийоми для уникнення помилок?
6. Які існують прийоми підвищення ефективності програми?
7. У чому полягає розробка дружнього інтерфейсу?

8. Файли

До цього ми розглядали введення інформації з клавіатури та виведення її на екран. Проте процес введення з консолі дуже трудомісткий, а результат виведення на консоль – недовговічний. На щастя існує зручніший спосіб записувати, зберігати, пересилати та при необхідності зчитувати інформацію з постійної (зовнішньої) пам’ яті комп’ютера. Для цього застосовуються фізичні файли. Вибір між консоллю та файлами вам доведеться робити кожного разу, коли ви станете писати чергову програму.

8.1. Поняття файлу та змінної файлового типу

З файлами ви вже багаторазово зустрічалися. В них ви зберігали тексти програм, з їх допомогою ви переносили програми з одного комп’ютера на інший. Переклад українською мовою слова *file* – зберігати інформацію, а також картотека, досьє – усі ці значення правильно характеризують і нове значення цього слова.

Під **файлом** розуміється або іменована область зовнішньої пам’яті ПК (жорсткого диску, гнучкої дискети, електронного «віртуального» диску), або логічний пристрій – потенційне джерело або приймач інформації. У першому випадку файл – це впорядкований набір записів або інша сукупність даних у певному форматі, що зберігається у комп’ютерній системі під загальним ім’ям. Уся сукупність файлів ділиться на два великі класи – файли програм та файли даних. За типом даних, що зберігаються, файли діляться на текстові, графічні, двійкові, командні, файли БД, відео- та аудіо-файли.

Будь-який файл має три характерні особливості. По-перше, у нього є ім’я, що дає можливість програмі працювати одночасно з кількома файлами. По-друге, він містить компоненти одного типу. Типом компонентів може бути будь-який тип TURBO-PASCAL, окрім файлів. Іншими словами, не можна створити «файл файлів». По-третє, довжина новостворюваного файлу ніяким чином не обумовлюється при його оголошенні і обмежується лише ємкістю пристроїв зовнішньої пам’яті.

Робота з фізичними файлами та логічними файлами (клавіатура, екран, порт введення/вивення) на мові Pascal здійснюється через змінну файлового типу.

Файловий тип або **змінну файлового типу** можна задати одним з трьох способів, яку як і усі змінні, оголошують у розділі опису змінних *var*:

<ім’я> = FILE OF <тип>;
<ім’я> = TEXT;
<ім’я> = FILE;

Тут <ім’я> – ім’я файлового типу (правильний ідентифікатор);
FILE OF – зарезервовані слова;
TEXT – ім’я стандартного типу текстових файлів;
<тип> – будь-який тип TURBO-PASCAL, окрім файлів.

Наприклад:

```
type   product = record
                 name : String;
                 code : Word;
                 cost : comp
       end;
       text_80 = file of String [80];
var    f1 : file of char;
       f2 : text;
       f3 : file;
       f4 : text_80;
       f5 : file of product;
```

В залежності від способу оголошення можна виділити три види файлів:

- **типизовані файли** (задаються реченням FILE OF...);
- **текстові файли** (визначаються типом TEXT);
- **нетипизовані файли** (визначаються типом FILE).

У наших прикладах F1, F4 та F5 – типизовані файли, F2 – текстовий файл, F3 –нетипизований файл. Вид файлу, взагалі кажучи, визначає спосіб зберігання інформації у файлі. Проте у TURBO-PASCAL немає засобів контролю виду раніше створених файлів. При оголошенні вже існуючих файлів програміст повинен сам стежити за відповідністю виду оголошення характеру файлу.

Нетипізовані файли – це файли, підтримка яких здійснюється з максимально можливою швидкістю. Введення таких файлів у систему було викликано прагненням підвищити ефективність програм, які приймають участь в інтенсивному обміні із зовнішніми наборами даних.

Ці файли на відміну від типізованих та текстових файлів не мають строго певного типу. Нетипізований файл розглядається у Pascal як сукупність символів або байтів. Представлення *char* або *byte* не грає жодної ролі, важливий лише об’єм займаних даних. Таке представлення стирає відмінності між файлами незалежно від типу їх оголошення. На практиці це призведе до того, що будь-який файл, який підготовлений як текстовий або типізований, можна відкрити та розпочати роботу з ним, як з нетипізованим набором даних.

8.2. Доступ до файлів

Будь-якій програмі доступні два заздалегідь оголошених файли із стандартними файловими змінними, які пов’язані з консоллю за замовченням (тобто програмістові не треба робити жодних дій, щоб зв’язати консоль та ці файли): INPUT – для читання даних з клавіатури та OUTPUT – для виведення на екран. Процедура ReadLn(S) за замовченням працює як ReadLn(Input, S) та читає у змінну S інформацію з клавіатури. Процедура WriteLn(S) діє так само, як процедура WriteLn(Output, S), та виводить інформацію із змінної S на екран

монітора. Стандартний Pascal вимагає обов'язкової згадки цих файлів у заголовку програми, наприклад, таким чином:

PROGRAM NameOfProgram(input,output);

У TURBO-PASCAL це необов'язково, **ось чому заголовок програми можна опускати**.

Будь-які інші файли, а також логічні пристрої стають доступними програмі тільки після виконання особливої процедури відкриття файлу (логічного пристрою). Ця процедура полягає у з'язуванні раніше оголошеної файлової змінної з ім'ям існуючого або новостворюваного файлу, а також у вказівці напрямку обміну інформацією: читання з файлу або запис у нього.

Файлова змінна зв'язується з іменем файлу в результаті звернення до стандартної процедури ASSIGN (читається "э'сайн"):

ASSIGN (<ф.з.>, <ім'я файлу або л.п.>);

Тут <ф.з.> – файлова змінна (правильний ідентифікатор, який оголошений у програмі як змінна файлового типу);

<ім'я файлу або л.п.> – текстовий вираз, що містить ім'я файлу або логічний пристрій.

Якщо ім'я файлу задається у вигляді порожнього рядка, наприклад, **ASSIGN(f, '')**, то залежно від напряму обміну даними файлова змінна зв'язується із стандартним файлом INPUT або OUTPUT.

8.3. Імена файлів

Ім'я файлу – це будь-який вираз строкового типу, який будується за правилами визначення імен в MS-DOS (операційній системі ПК):

- ім'я містить до восьми дозволених символів; дозволені символи – це прописні та строкові латинські літери, цифри та символи:
  ! @ # $ % ^ & ( ) ' ~ – _
- ім'я починається з будь-якого дозволеного символу;
- за ім'ям може слідувати розширення – послідовність до трьох дозволених символів; розширення, якщо воно є, відділяється від імені крапкою. Перед ім'ям може вказуватися так званий шлях до файлу: ім'я диска та/або ім'я поточного каталогу та решта імен каталогів наступних рівнів.

Ім'я диска – це один із символів A...Z, після якого ставиться двокрапка. Імена
A: і B: належать до дискових накопичувачів на гнучких дискетах, імена C:, D: та наступні – до жорстких дисків. Якщо ім'я диску не вказане, то мається на увазі пристрій за замовченням – той, який був встановлений в операційній системі перед початком роботи програми.

За ім'ям диску може вказуватися ім'я каталогу, який містить інформацію про файл. Якщо імені каталогу передує зворотний слеш (\), то шлях до файлу починається з кореневого каталогу, якщо слеша немає – з поточного каталогу,

що встановлений у системі за замовченням. За ім'ям каталогу може слідувати одне або декілька імен каталогів нижнього рівня. Кожному з них повинен передувати зворотний слеш. Весь шлях до файлу відділяється від імені файлу зворотнім слешем. Максимальна довжина імені разом із шляхом – 79 символів, наприклад:

```
var     finp: text;
        fout: file of  String;
const  name = 'c:\dir\subdir\out.txt';
begin
        assign(finp,'123.dat');
        assign(fout,name);
end.
```

8.4. Логічні пристрої

Стандартні апаратні засоби ПК, такі як клавіатура, екран дисплея, пристрій друку (принтер) та комунікаційні канали введення-виведення, визначаються у TURBO-PASCAL спеціальними іменами, які називаються логічними пристроями. Усі вони у TURBO-PASCAL розглядаються як потенційні джерела або приймачі текстової інформації.

CON – логічне ім'я, яке визначає консоль – клавіатуру та екран дисплея, встановлює відмінність між цими фізичними пристроями за напрямом передачі даних: читання даних можливе тільки з клавіатури, а запис – тільки на екран. Таким чином, за допомогою логічного пристрою *CON* не можна, наприклад, зчитати дані з екрану ПК, хоча така апаратна можливість існує.

Введення з клавіатури буферизується: символи по мірі натиснення на клавіші розміщуються у спеціальний буфер для рядка символів, який передається програмі тільки після натиснення на клавішу Enter. Буферизація введення забезпечує можливість редагування рядка, що вводиться, стандартними засобами DOS. При введенні символів здійснюється їх exo-повтор на екрані ПК. У TURBO-PASCAL можна зчитати будь-який символ з клавіатури, у тому числі символ CR, що виробляється клавішею Enter, одразу після натиснення на цю клавішу без exo-повтора.

PRN – логічне ім'я принтера. Якщо до ПК підключено декілька принтерів, доступ до них здійснюється по логічних іменах *LPT1, LPT2* та *LPT3.* Імена *PRN* та *LPT1* спочатку – це синоніми. Засобами DOS можна присвоїти ім'я *PRN* будь-якому іншому логічному пристрою, який здатний приймати інформацію.

Стандартний бібліотечний модуль PRINTER, що входить в бібліотеку TURBO.TPL, оголошує ім'я файлової змінної LST та пов'язує його з логічним пристроєм LPT1. Підключається цей модуль за допомогою оператора Uses. Це дає можливість використовувати просте звернення до принтера. Наприклад, програма

```
Uses Printer;
begin
```

```
    WriteLn(LST, 'Hello World!')
end.
```

виведе на принтер фразу «Hello World!», а всі необхідні операції по відкриттю логічного пристрою виконає бібліотечний блок PRINTER.

AUX – логічне ім'я послідовного комунікаційного каналу, який зазвичай використовується для зв'язку ПК з іншими машинами. Комунікаційний канал може здійснювати як прийом, так і передачу даних, проте у програмі у кожен момент часу йому можна призначити тільки одну з цих функцій. Як правило, у складі ПК є два комунікаційні канали, яким даються імена логічних пристроїв *COM1* та *COM2*. Спочатку імена ***AUX*** та ***Com1*** **– синоніми**.

NUL **–** логічне ім'я «порожнього» пристрою. Це пристрій найчастіше використовується у режимі налагоджування і трактується як пристрій-приймач інформації необмеженої ємності. При зверненні до NUL як до джерела інформації видається ознака кінця файлу *EOF*. Пов'язування логічного пристрою з файловою змінною здійснюється процедурою *ASSIGN,* наприклад:

```
var   fi, fo : text;
begin
      assign(fi,'AUX');
      assign(fo,'LPT2');
end.
```

Якщо потрібно вивести інформацію не на екран, а на принтер, потрібно вказати як ім'я файлу спеціальний файл пристрою:

```
Assign(FIL,'PRN');
ReWrite(FIL);
```

З цієї миті вся інформація, що виводиться процедурами Write і WriteLn, поступатиме не на екран, а на принтер. Повернутися до введення та виведення інформації з консолі можна за допомогою процедур:

```
Assign(FIL,'CON');
ReWrite(FIL);
```

Іноді виникає необхідність в одній програмі виводити інформацію як на екран, так і на принтер. Виведення на екран виконується процедурою WriteLn(список_виведення). Для виведення на принтер використовується процедура WriteLn у форматі WriteLn(lst, список_виведення), де lst – ім'я файлу, яке пов'язане з принтером. Це ім'я оголошене у модулі Printer, який має бути підключений оператором Uses Printer.

TURBO-PASCAL ніколи не пов'язує імена логічних пристроїв з дисковими файлами, у цьому сенсі ці імена можна вважати зарезервованими. Іншими словами, не можна, наприклад, звернутися до дискового файлу з іменем PRN, оскільки TURBO-PASCAL завжди інтерпретує такий запит як звернення до принтера.

Наведемо приклад програми, яка видає результат своєї роботи на принтер.

Приклад 8.1.

```
{ Програма обчислення суми елементів введеного масиву }
program SUMMAS;
  uses PRINTER;
```

```
  var  MAS: ARRAY [1..10] OF REAL;
       I: INTEGER;
       SUM: REAL;
begin
  { Введення значень елементів масиву }
  for I := 1 to 10 do
    begin
      write('Введіть черговий елемент для обчислення суми [',I:2']');
      readln(MAS[I]);
    end;
  { Друк значень елементів масиву }
  writeln(LST,'Значения елементів масиву:');
  for I := 1 to 10 do
    writeln(LST,'MAS[',I:2']=',MAS[I]:3:2);
  writeln(LST);
  { Знаходження та друк суми елементів масиву }
  SUM := 0;
  for I := 1 to 10 do
    SUM := SUM + MAS[I];
  writeln(LST,'Сумма значень елементів масиву = ',SUM:6:2);
end.
```

8.5. Ініціалізація файлу

Ініціалізувати або відкрити файл – це означає вказати для цього файлу напрям передачі даних. У TURBO-PASCAL можна відкрити файл для читання, для запису інформації, а також для читання та запису одночасно.

Для читання файл ініціалізується за допомогою стандартної процедури RESET:

RESET (<ф.з.>);

Тут <ф.з.> – файлова змінна, яка пов'язана раніше процедурою ASSIGN з вже існуючим файлом або логічним пристроєм-приймачем інформації.

При виконанні цієї процедури дисковий файл або логічний пристрій готується до читання інформації. В результаті спеціальна змінна-покажчик, яка пов'язана з цим файлом, буде вказувати на початок файлу, тобто на компонент з порядковим номером 0.

Якщо робиться спроба ініціювати читання з неіснуючого файлу або з логічного пристрою PRN, виникає помилка періоду виконання, яка може бути повідомлена програмі ненульовим значенням вбудованої функції IORESULT типу WORD. Наприклад, наступний фрагмент програми дозволяє встановити, чи існує необхідний файл на диску:

```
var     f:  file of char;
begin
```

```
assign(f,'myfile.dat');
{$I-} {Відключаємо контроль помилок введення-виводу}
reset(f);
{$I+} {Включаємо контроль помилок введення-виводу}
if IOResult <> 0 then
        ..... {Файл не існує}
else
        ..... {Файл існує}
end.
```

У цьому фрагменті за допомогою директиви компілятора **{$I-}** відключається автоматичний контроль помилок введення-виведення. Якщо цього не зробити, то відсутність файлу призведе до аварійного завершення програми. Після відкриття файлу оператором **Reset(f)** покажчик встановлюється на початок файлу. Дані зчитуються з початку файла оператором Read або ReadLn**.** Повторне вживання оператора Reset(f); встановлює покажчик на початок файлу для зчитування даних, вміст файлу при цьому не змінюється.

У TURBO-PASCAL дозволяється звертатися до типізованих файлів, які відкриті процедурою RESET (тобто для читання інформації), за допомогою процедури REWRITE (тобто для запису інформації). Така можливість дозволяє легко оновлювати раніше створені файли, які є типізованими, та при необхідності розширювати їх. **Для текстових файлів, які відкриті процедурою RESET, не можна використовувати процедуру WRITE або WRITELN**.

Стандартна процедура REWRITE (<ф.з.>)

ініціює запис інформації у файл або у логічний пристрій, який був пов'язаний раніше із файловою змінною <ф.з.>. Процедурою REWRITE не можна ініціювати запис інформації у вже раніше існуючий дисковий файл: при виконанні цієї процедури старий файл знищується і ніяких повідомлень про це у програму не передається. Новий файл готується до прийому інформації і його покажчик приймає значення 0.

Стандартна процедура APPEND (<ф.з.>)

ініціює запис у раніше існуючий текстовий файл для його розширення, при цьому покажчик файла встановлюється у його кінець. Процедура APPEND застосовується тільки до текстових файлів, тобто їхня файлова змінна повинна мати тип TEXT. Процедурою APPEND не можна ініціювати запис у файл, що типізується або не типізується. Якщо текстовий файл раніше вже був відкритий за допомогою RESET або REWRITE, використання процедури APPEND призведе до закриття цього файлу та відкриття його знову, але вже для додавання записів.

8.6. Процедури та функції для роботи з файлами

Нижче описуються деякі процедури та функції, які можна використовувати з файлами.

Процедура Close. Закриває файл, проте зв'язок файлової змінної з іменем файлу, який був встановлений раніше процедурою ASSIGN, зберігається. Формат звернення:

CLOSE (<ф.з.>)

При створенні нового або розширенні старого файлу процедура забезпечує збереження у файлі усіх нових записів та реєстрацію файлу у каталозі. Функції процедури CLOSE (читається "клоуз") виконуються автоматично по відношенню до всіх відкритих файлів при нормальному завершенні програми. Оскільки зв'язок файлу з файловою змінною зберігається, файл можна повторно відкрити без додаткового використання процедури ASSIGN.

Процедура RENAME. Перейменовує файл. Формат звернення:

RENAME (<ф.з.>, <нове ім'я>)

Тут <нове ім'я> – строковий вираз, що містить нове ім'я файлу. Перед виконанням процедури необхідно закрити файл, якщо він раніше був відкритий процедурами RESET, REWRITE або APPEND.

Процедура ERASE. Знищує файл. Формат звернення:

ERASE (<ф.з.>)

Перед виконанням процедури необхідно закрити файл, якщо він раніше був відкритий процедурами RESET, REWRITE або APPEND.

Наступний фрагмент програми показує, як можна використовувати процедури RENAME та CLOSE при роботі з файлами. Припустимо, що потрібно відредагувати файл, ім'я якого містить змінна NAME. Перед редагуванням необхідно переконатися, що потрібний файл існує на диску, та перейменувати його – замінити розширення цього файлу на BAK (страхувальна копія). Якщо файл з таким розширенням вже існує, його треба знищити.

```
var   fi : text;      {Початковий файл}
      fo : text;      {Відредагований файл}
      name : String;
      name_bak: String;
      k, i: Word;
const bak = '.bak';
begin
      . . . .
      {Отримуємо в name.bak ім'я файлу з розширенням .BAK:}
      k := pos('.',name);
      if k = 0 then
      k := length(name)+1;
      name_bak := copy(name,1,k-1)+ bak;
      {Перевіряємо існування початкового файлу:}
      assign(fi,name);
      {$I-} ' reset(fi);
```

```
if lOResult <> 0 then
      halt;    {Завершуємо програму: файлу не існує}
close(fi);
{Перевіряємо існування .ВАК-файла:}
assign(fo,name_bak);
reset (fo);
{$I+}
if lOResult = 0 then
begin {Файл .ВАК існує:}
     close(fo); {закриваємо його}
    erase(fo) {та знищуємо}
end;
{Перевірки закінчені, підготовка до роботи:}
rename(fi,name_bak);
reset(fi);
assign(fo,name);
rewrite(fo);
. . . .
end.
```

Перевірка на існування файлу .ВАК у даному прикладі необхідна, оскільки звернення rename(fi,name_bak) викличе помилку у випадку, якщо такий файл існує.

8.7. Текстові файли

Текстові файли найширше використовуються у науково-технічних дослідженнях. Одна з основних переваг цих файлів полягає у тому, що їх вміст зручно переглядати, оскільки він представлений не у машинному коді, як типизовані або нетипизовані файли, а у кодах ASCII – у вигляді цифр, букв та інших символів клавіатури.

Текстові файли зв'язуються з файловими змінними, які належать стандартному типу TEXT. Текстові файли призначені для зберігання текстової інформації. Саме такого типу у файлах зберігаються, наприклад, тексти програм. Компоненти (записи) текстового файлу можуть мати змінну довжину, що суттєво впливає на характер роботи з ними.

Текстовий файл трактується у Pascal як сукупність рядків змінної довжини, які містять символи з набору ASCII. Доступ до рядка можливий лише послідовно, починаючи з першого символу. Припустимо, що необхідно прочитати з файлу, якому відровідає файлова змінна F_TXT четвертий його елемент у змінну *sim* цього ж типу – це можливо здійснити, лише прочитавши «даремно» три перші елементи у записі цього файлу:

```
. . .
Reset(F_TXT);     {Відкриває файл для читання та встановлює його покажчик }
                  {на першу компоненту}
For  i:= 1 to 4 do
```

```
    Read(F_TXT, sim);
Reset(F_TXT);     {Встановлює заново покажчик на першу компоненту файлу}
. . .
```

У змінній *sim* залишиться лише остання (четверта) прочитана компонента файлу.

При створенні текстового файлу (лише текстового файлу) у Turbo-редакторі (чи за допомогою іншого редактора) у кінці кожного рядка слід натискати клавішу Enter, цю ж клавішу необхідно натиснути також у кінці файлу. У цьому випадку у кінці запису рядка ставиться спеціальна ознака EOLN (End OfLiNe – кінець рядка), а у кінці всього файлу – ознака EOF (End Of File – кінець файлу). Ці коди можна помістити у файл також за допомогою набору символів #13#10 (EOLN) та одночасним натисненням клавіш Ctrl+Z (EOF). Ці ознаки можна протестувати однойменними логічними функціями (див. нижче). При формуванні текстових файлів використовуються такі системні фунції:

EOLN - послідовність код ASCII #13 (LF) і #10 (CR);

EOF - код #26 стандарту ASCII.

Наведемо приклад текстового файлу з вказівкою усіх невидимих (керуючих) символів, які можуть бути у ньому присутніми:

Це[20]**текстовий**[20]**файл,**[20]**який**[20]**зберігає**[20]**числа**:[13][10]
[13][10]
[20][20]**127**[20][20]**16.0**[13][10]
[20][20]**234**[20][20]**17.5**[13][10]
Кінец[20]**файлу**[13][10]
[26]

Управляючі символи, які позначені їх ASCII кодом, оточені квадратними дужками: [13] – повернення каретки, [10] – переведення на новий рядок, [20] – пропуск, [26] – ознака кінця файлу. При виведенні на екран цей файл виглядає таким чином:

Це текстовий файл, який зберігає числа:
127 16.0
234 17.5
Кінец файлу

Для доступу до записів застосовуються процедури READ, READLN, WRITE, WRITELN. Вони відрізняються можливістю звернення до них із змінним числом фактичних параметрів, в якості яких можуть використовуватися символи, рядки та числа. Першим параметром у будь-якій із перерахованих процедур може стояти файлова змінна. У цьому випадку здійснюється звернення до дискового файлу або логічного пристрою, який пов’язаний із змінною процедурою ASSIGN. Якщо файлова змінна не вказана, відбувається звернення до стандартних файлів INPUT та OUTPUT.

Процедура READ. Забезпечує введення символів, рядків та чисел. Формат звернення:

READ (<ф.з.>,<сп.введення>) або READ (<сп.введення>)

Тут <сп.введення> – список введення: послідовність з однієї або більше змінних типу CHAR, STRING, а також будь-якого цілого або дійсного типу.

При введенні змінних типу CHAR виконується читання одного символу з файлу та присвоєння зчитаного значення змінній. Якщо перед виконанням читання покажчик файлу досяг кінця чергового рядка, то результатом читання буде символ CR (ASCII код #13), а якщо досягнутий кінець файлу, – то символ EOF (код #26). При введенні з клавіатури символ CR вводиться при натисненні на клавішу Enter, а символ EOF – при одночасному натисненні клавіш CTRL та Z.

При введенні змінних типу STRING кількість прочитаних процедурою та розміщених у рядку символів дорівнює максимальній довжині рядка, якщо тільки раніше не зустрілися символи CR або EOF. У цьому випадку самі символи CR та EOF у рядок не поміщаються. Якщо кількість символів у вхідному потоці даних більша за максимальну довжину рядка, «зайві» символи до кінця рядка відкидаються, а нове звернення до READ повертає порожній рядок. Таким чином, процедура READ не в змозі прочитати послідовність рядків: перший рядок буде прочитаний нормально, а всі наступні виявляться порожніми. Для введення послідовності рядків треба використовувати процедуру READLN

При введенні числових змінних процедура READ спочатку виділяє підрядок у вхідному потоці за наступними правилами:

- усі попередні пропуски (пробіли), символи табуляції та маркери кінця рядка EOLN пропускаються;
- після виділення першого значущого символу, навпаки, будь-який з перерахованих символів або символ EOF служать ознакою кінця підрядка.

Виділений таким чином підрядок потім розглядається як символьне представлення числової константи відповідного типу та перетвориться у внутрішнє уявлення, а набуте значення присвоюється змінній. Якщо у підрядку був порушений необхідний формат представлення чисельної константи, виникає помилка введення-виведення. Якщо при введенні попередніх пробілів зустрівся символ EOF, змінна набуває значення 0. Відзначимо, що у TURBO-PASCAL не передбачено введення шістнадцяткових констант.

При використанні процедури READ стосовно стандартного файлу INPUT, тобто при введенні з клавіатури, символьні рядки запам’ятовуються у буфері, який передається процедурі тільки після натиснення на клавішу Enter. Це дозволяє редагувати дані при їх введенні. Для редагування використовуються такі клавіші:

- Backspace, CTRL-H: переведить курсор ліворуч та стирає символ зліва від курсору;
- переведення курсору праворуч – відновлює символ за символом попередній рядок введення;

– CTRL-Z Enter – завершує введення за процедурою READ; «зайві» символьні параметри, що залишилися, приймають значення CHR(26), рядки повертаються порожніми, а чисельні змінні залишаються без зміни.

Максимальна довжина буфера введення при роботі з клавіатурою складає 127 символів. Введення з клавіатури за процедурою READ супроводжується ехо-повтором символів, що вводяться, на екрані ПК.

Процедура READ чудово пристосована для введення чисел. При зверненні до неї для введення чергового цілого або дійсного числа процедура «перестрибує» маркери кінця рядків, тобто фактично весь файл розглядається нею як один довгий рядок, який містить текстове представлення чисел. У поєднанні з перевіркою кінця файлу функцією EOF процедура READ дозволяє організувати просте введення масивів даних, наприклад, таким чином:

```
const N = 1000; {Максимальна довжина введення}
var     f : text;
        m : array [1..N] of real;
        i : Integer;
begin
        assign(f, ‘prog.dat’);
        reset(f); i := 1;
        while not EOF(f) and (i <= N) do
        begin
                read(f,m[i]);
                inc(i)
        end;
        close(f);
end.
```

Процедура READLN. Забезпечує введення символів, рядків та чисел. Ця процедура ідентична процедурі READ за винятком того, що після зчитування останньої змінної частина рядка, що залишилася, до маркера EOLN пропускається, тому наступне звернення до READLN або READ починається з першого символу нового рядка. Крім того, цю процедуру можна викликати без параметра <сп.введення> (див. процедуру READ), що призведе до пропуску всіх символів поточного рядка аж до EOLN.

Якщо процедура використовується для читання з клавіатури, натиснення на клавішу Enter перетвориться у послідовність байтів CR + LF, а курсор буде вказувати на початок наступного рядка, тоді як у процедурі READ ехо-повтором клавіші Enter є символ CR і курсор поміщається на початок поточного рядка.

Процедура WRITE. Забезпечує виведення інформації у текстовий файл або передачу її на логічний пристрій. Формат звернення:

WRITE (<ф.з.>, <сп.виведення>) або WRITE (<сп.виведення >)

Тут <сп.виведення> – список виведення: послідовність з одного або більше виразів типу CHAR, STRING, BOOLEAN, а також будь-якого цілого або дійсного типу.

Файлова змінна <ф.з.>, якщо вона вказана, повинна бути заздалегідь описана як змінна типу TEXT та пов'язана з іменем файлу або логічним пристроєм процедурою ASSIGN. Якщо файлова змінна відсутня, мається на увазі виведення у стандартний файл OUTPUT, який зазвичай пов'язаний з екраном ПК.

Будь-який елемент списку виведення може мати форму

OutExpr [: MinWidth [: DecPlaces]]

Тут OUTEXPR – вираз, який виводиться;

MINWIDTH та DECPLACES – вирази типу WORD (квадратні дужки означають можливість відсутності укладених в них параметрів).

Підпараметр MINWIDTH, якщо він присутній, вказує мінімальну ширину поля, в яке має записуватись символьне подання значення OUTEXPR. Якщо символьне подання має меншу довжину, ніж MINWIDTH, воно буде доповнено зліва пробілами, якщо ж воно має велику довжину, то підпараметр MINWIDTH ігнорується і виводиться необхідне число символів.

Підпараметр DECPLACES задає кількість десяткових знаків у дробовій частині дійсного числа. Він може використовуватись тільки спільно з MINWIDTH і лише стосовно виразу одного з дійсних типів, що виводиться.

Якщо ширина поля виведення не вказана, відповідний параметр виводиться одразу за попереднім без якого-небудь їх розділення. Символи та рядки передаються файлу виведення без змін, але забезпечуються попередніми пробілами, якщо задана ширина поля виведення і ця ширина більше за потрібну для виведення.

При виведенні логічних виразів залежно від їх значення виводяться рядки TRUE або FALSE. (Введення логічних констант процедурами READ або READLN не передбачено).

Дійсні числа виводяться в експоненціальному форматі, якщо не вказаний підпараметр DECPLACES, інакше обирається формат представлення числа з фіксованою крапкою. Експоненціальний формат представляє дійсне число у вигляді

_s#.###############E*####,

де: _ – пропуск;

s – пропуск для додатного, а знак «-» для від'ємного чисел;

– десяткова цифра;

E – символ десяткової основи системи числення;

* – знак «+» або «-» залежно від знаку десяткового порядку числа.

Якщо підпараметр MINWIDTH опущений, приймається його значення за замовченням (23). Якщо MINWIDTH менше 10, вважається, що він дорівнює 10.

Якщо підпараметр DECPLACES дорівнює нулю, то ані дробова частина числа, ані десяткова крапка не виводяться. При від'ємному значенні DECPLACES цей параметр ігнорується, і число виводиться в експоненціальному форматі з урахуванням MINWIDTH. Якщо значення DECPLACES більше 18, приймається значення 18. Слід врахувати, що в разі вказівки підпараметра DECPLACES дійсне число завжди виводитиметься у форматі з фіксованою крапкою та необхідною кількістю знаків у дробовій частині, навіть якщо значення підпараметра MINWIDTH виявиться недостатнім для розміщення цілої частини: у цьому випадку значення MINWIDTH автоматично збільшується.

При виведенні на екран у випадку, коли довжина низки символів, що виводиться, перевищує ширину екрану, «зайві» символи переносяться на наступний екранний рядок. При заповненні екрану його вміст зсувається догори на один рядок.

Процедура WRITELN. Ця процедура повністю ідентична процедурі WRITE за винятком того, що рядок символів, який виводиться, завершується кодами CR і LF. При виклику WRITELN можна опускати параметр <сп.виведення>, у цьому випадку у файл передається маркер EOLN, що при виведенні на екран призведе до переведення курсору на початок наступного рядка.

Логічна функція EOLN. Функція повертає TRUE, якщо у вхідному текстовому файлі досягнутий маркер кінця рядка. А функція **EOF** – повертає TRUE, якщо досягнутий кінець файла. Формат звернення:

EOF(<ф.з.>)

EOLN(<ф.з.>)

Якщо параметр <ф.з.> (файлова змінна) опущений, функції перевіряють стандартний файл INPUT. Існує деяка відмінність у роботі функцій EOLN та EOF з дисковими файлами і логічними пристроями. Річ у тім, що для логічного пристрою неможливо передбачити, яким буде результат читання чергового символу. Тому при роботі з логічним пристроєм функція EOLN повертає TRUE, якщо останнім прочитаним з пристрою символом був EOLN або EOF, тоді як при читанні з диска TRUE повертається у випадку, якщо наступним прочитаним символом буде EOLN або EOF. Аналогічна відмінність спостерігається також у функції EOF: для логічного пристрою TRUE повертається у випадку, якщо останнім символом був EOF, а при читанні з диска – якщо наступним зчитуваним символом буде EOF. Іншими словами, функції тестують відповідні ознаки для логічного пристрою після чергового читання, а для файлу – перед читанням.

Нагадаємо ще такі функції при роботі з файлами:

Filesize (f) −повертає розмір файлу в символах,

Filepos(f) − повертає поточне місце покажчика при читанні у файлі,

Seek (f, N) – робить поточним символ N у файлі f.

8.8. Робота з текстовими файлами

При роботі з числовими даними необхідно враховувати, що числа у файлі повинні відділятися одне від одного хоча б одним пробілом. Отже, при записі числових даних у файл необхідно використовувати формат з достатньою кількістю позицій для виведення. Наведемо приклади операторів запису та читання даних.

```
Assign(f1,'File1.dan');          { призначити змінній f1  файл з ім'ям: File1. dan }
ReWrite(f1);                 { відкрити файл для запису в першій програмі }
Writeln(f1,'Значения "X","Y"' );          { почати запис }
 For i:= 1 to N do begin
     X:= 0.5*i;   Y:= Ln(X);          { приклад розрахунку значень змінних }
     write(f1, X:6:2, Y:10:4);        { записати дані у файл File1. dan }
     If i mod 5 = 0 then writeln(f1)          { записати символ #13 }
end;
Close (f1);                      { закрити файл у першій програмі }
Assign(f2,'File1.dan');          { призначити змінній f2,  файл з ім'ям: File1. dan }
Reset(f2);                       { відкрити файл для читання в другій програмі }
Readln(f2);                      { пропустити перший рядок }
For i:= 1 to N do begin
    read(f2, a[i], b[i]);        { зчитати дані у масиви "A" та "B" }
    If i mod 5 = 0 then readln(f2)           { зчитати символ #13 }
end;
Close (f2);                  { закрити файл у другій програмі }
```

При роботі із строковими даними необхідно вказувати довжину змінної типу String в операторі опису типів змінних. Інакше оператором Read(f, S) у строкову змінну "S" зчитається до 255 символів, а оператором Readln(f, S) зчитаються всі символи до #13, але не більше 255, причому пропуски у кінці рядка ігноруються. Наведемо приклад програми для зчитування строкових та числових даних з файлу та запису їх в інший файл.

```
var   c: char;   j, i:  word;
       s: array[1..10] of string[12];
       a: array[1..10, 1..6] of word;
       f1, f2: text;
BEGIN
 assign(f1, 'F1.txt');  reset(f1);
 assign(f2, 'F2.txt');  rewrite(f2);
 for i:= 1 to 10 do  begin
     read(f1, s[i]);                  { зчитування рядка }
     for j:= 1 to 6 do read(f1, a[i,j]);      { зчитування шести чисел }
     readln(f1)                       { зчитування символу кінця рядка }
  end;
 for c:= 'A' to 'Я' do                { цикл по перебору символів }
 for i:= 1 to 10 do
```

```
      if s[i,1] = c then begin
       write(f2,  s[i]);      { запис рядків в алфавітному порядку за першими символами }
       for j:= 1 to 6  do  write(f2, a[i,j]:2);      { запис шести чисел }
       writeln(F2)
      end;
  close(f1); close(f2);
END.
```

Тут вважається що у файлі F1.txt записані дані, які мають вигляд:

Іванов 5 4 4 5 4 3
Петров 4 5 3 4 3 4

і таке інше. Після зчитування даних у програмі відбувається їх сортування шляхом перебору та запис у файл F2.txt в алфавітному порядку тільки за першою літерою прізвища.

При зчитуванні даних з файлу невизначеної довжини можна використовувати функцію **EOF(f);** яка повертає ознаку кінця файлу, а саме: EOF(f) дорівнює True, якщо покажчик стоїть на ознаці кінця файлу (код #26), інакше EOF(f) дорівнює False. Наведемо приклад операторів для зчитування тексту з файлу FF1.t, кодування тексту та запису у файл FF2.t із збереженням коду #13.

```
assign(f1, 'FF1. t');  reset(f1);
assign(f2, 'FF2. t');  rewrite(f2);
while not EOF(f1) do begin read(f1,c);      {зчитуємо змінну типу Char }
 if c <> #13 then c:=pred(c); write(f2,c)    {кодуємо та виводимо на екран }
end;
```

Приклад 8.2.

```
{ Програма обчислення суми елементів введеного масиву }
program SUMMAS;
  uses PRINTER;
  var  MAS: ARRAY [1..10] OF REAL;
       I: INTEGER;
       SUM: REAL;
begin
  { Введення значень елементів масиву }
  for I := 1 to 10 do  begin
       write('Введіть черговий елемент для обчислення суми [',I:2']');
       readln(MAS[I]);
  end;
  { Перепризначення виведення Output-екрану у текстовий файл }
  Assign(Output,'c:\bp\progr\Vvod.txt');
  {$I-}  {Відключаємо автоконтроль}
  {Reset(Output);}
  Rewrite(Output);
  {$I+}
```

```
  { Друк значень елементів масиву }
  writeln('Значення елементів масиву:');
  for I := 1 to 10 do
        writeln('MAS[',I:2']=',MAS[I]:3:2);
   writeln;
  { Пошук та друк суми елементів масиву }
  SUM := 0;
  for I := 1 to 10 do
     SUM := SUM + MAS[I];
writeln('Сума значень елементів масиву = ',SUM:6:2);
end.
```

Для подальшого знайомства з текстовими файлами наведемо приклад програми, яка вирішує наступні завдання.

1. Створити файл початкових даних – масив з *N* цілих чисел.
2. Ввести у програму інформацію з файлу початкових даних, вивести її у файл з результатами.
3. Відсортувати масив за збільшенням елементів.
4. Вивести відсортований масив у файл з результатами.

Приклад 8.3.

```
Program TextFile;
Uses  Crt;
        Const  n= 5;                { Довжина масиву }
        k_isx = 'Isx_Dan.txt';    { Ім'я зовнішнього файлу з початковими даними }
        k_rez = 'Rez_Dan.txt';  { Ім'я зовнішнього файлу з результатами }
Type  Int_n = 1..n;
        Mas = Array[Int_n] of Integer;
        Var    i: Int_n;  R: Mas;
        F_Isx,F_Rez: Text;       { Файлові змінні (ФЗ)}
Procedure Chtenie;               { Читає із зовнішнього файлу в масив R}
Begin
   Assign(F_Isx, k_isx); { Здійснює зв'язок ФЗ  F_Isx із зовнішнім файлом }
   Reset(F_Isx);             { Відкриває файл F_Isx для читання }
   for i:= 1 to n do
        Read(F_Isx, R[i]);
           Close(F_Isx);    { Закриває файл з початковими даними }
End; {Chtenie}
Procedure Pechat(M: Mas);   { Друк даних }
Var     j: Int_n;
Begin
   for j:= 1 to n do
        Write(F_Rez, M[j]:4);
           WriteLn(F_Rez);
        End; {Pechat}
Procedure Sort(Var M: Mas);
```

```
Var   i, Candidat: Int_n;  x: Integer;
Begin
   for i:= 1 to n-1 do
       for Candidat:= i+1 to n do
          if M[i] > M[Candidat]  then  begin
               x:= M[i];
               M[i]:= M[Candidat];
               M[Candidat]:= x;
           end;  {then}
       end; {Sort}
BEGIN
       ClrScr;
       Chtenie;
       Assign(F_Rez,k_rez);    { Зв'язок ФЗ F_Rez із зовнішнім файлом }
       Rewrite(F_Rez);         { Відкриває зовнішній файл для запису }
       WriteLn(F_Rez, 'Початковий масив R:');
       Pechat(R);
       Close(F_Rez);           { Закриває файл F_Rez}
       Sort(R);
       Append(F_Rez);  { Відкриває файл для його розширення або дозапису }
       WriteLn(F_Rez, 'Масив R після сортировки');
       Pechat(R);
       Close(F_Rez);           { Закриває файл F_Rez}
       END. {Text_File}
```

У цій програмі процедура *Chtenie* здійснює читання або введення даних (одновимірного масиву) із заздалегідь підготовленого вами файлу *Isx_Dan.txt* (тобто вже існуючого) в глобальну змінну *R*. Для досягнення цієї мети спочатку процедура *Assign(F_Isx, k_Isx)* зв'язує заздалегідь оголошену файлову змінну *F_Isx* з ім'ям файлу за допомогою глобальної строкової константи *k_isx*. Файл *Isx_Dan.txt* повинен знаходиться в тій же папці, що і програма (тобто в поточній директорії). Після завершення роботи процедури *Assign(F_Isx, k_Isx)* програма під файловою змінною *F_Isx* завжди розумітиме файл *Isx*_Dan.txt.

Для читання інформації з файлу його необхідно відкрити або підготувати до читання. Відкриває файл для читання процедура *Reset(Flsx)*. В результаті її роботи спеціальна змінна-покажчик, пов'язана з цим файлом, вказуватиме на початок файлу – на нульову компоненту першого рядка. У текстових файлах нумерація символів рядка починається з одиниці, в нульових же компонентах кожного рядка зберігається інформація про довжину рядка.

Власне читання інформації (елементів масиву) здійснюється циклом *for* і процедурою *Read(F_Isx, R[i])*, в якій використовується два параметри. Другий параметр – це та глобальна змінна *R[i]*, куди буде записана інформація. Першим же параметром є файлова змінна *F_Isx*, що вказує, звідки ця інформація буде отримана. Якщо перший параметр опустити (як ми завжди поступали раніше), то дані в *R[i]* повинні прочитуватися з екрану дисплея.

Процедура *Close(F_Isx)* для роботи підпрограми *Chtenie* не потрібна. Її наявність обумовлена лише виконанням рекомендацій відносно хорошого стилю програмування, згідно з яким відкриті файли після їх обробки завжди слід закривати цією процедурою. Процедура *Close* закриває канал введення-виведення інформації із зовнішнього файлу, з яким пов'язана файлова змінна, проте зв'язок файлової змінної з ім'ям, встановлений раніше процедурою *Assign*, при цьому зберігається.

У тілі програми процедура *Assign(F_Rez, k_rez)* здійснює зв'язок *F_Rez* з файлом *Rez_Dan.pas* через строкову константу *k_rez*. Процедура ж *ReWrite* (переписати) відкриває файл *F_Rez* для запису. Якщо файл з таким ім'ям (*Rez_Dan.pas*) вже існує, то уся інформація в ньому буде знищена. При цьому покажчик файлу встановлюється на нульову компоненту. Якщо ж такий файл ще не існує, то він буде створений: в нашому випадку з'явиться поки що порожній файл з ім'ям *Rez_Dan.pas*. Далі в цей файл буде записаний рядок *Ishodnyi massyv R :*. Якщо опустити перший параметр процедури *WriteLn*, то вказаний рядок з'явиться не у файлі, а на екрані дисплея.

Процедура *Pechat(R)* з фактичним параметром *R* роздрукує масив *R* у файл *F_Rez*.

8.9. Робота з типизованими файлами

Компонентами таких файлів можуть бути як прості (integer, longint, real тощо), так і структуровані типи даних (масиви, записи та ін.). Розмір буфера для типизованих файлів встановлюється автоматично на основі розміру його компонент. Нумерація компонент файлу починається не з одиниці, як в текстових файлах, а з нуля. Прямий доступ до компонент типизованого файлу забезпечується процедурою Seek (to seek – шукати):

Seek(<ФЗ>, <N компоненти>);

Тут <ФЗ> - файлова змінна**,** <N компоненти> - константа типу longint, вона вказує номер компоненти. Ця процедура зміщує покажчик файлу до необхідної компоненти, її не можна застосовувати до текстових файлів.

Функція **FileSize(<ФЗ>): longint** повертає число компонент файлу.

Виклик процедури **Seek(<ФЗ>, FileSize(<ФП>))** розміщує покажчик у кінці файлу – після останньої його компоненти. Це виявляється можливим внаслідок того, що, як відзначалося вище, нумерація компонент типизованих файлів починається з нуля, тому функція **FileSize(<ФЗ>)** повертає номер компоненти, що стоїть за останньою компонентою файлу.

Процедура **Truncate(<ФЗ>)** вилучає частину файлу, починаючи з поточної позиції до його кінця.

Для типизованих і нетипизованих файлів обидві процедури Reset і Rewrite дозволяють здійснювати як читання з файлу, так і запис у файл. Такий режим цих процедур встановлюється за замовченням, він дозволяє легко оновлювати раніше створені файли і при необхідності розширювати їх. У цьому випадку значення змінної FileMode модуля System дорівнює 2. При FileMode = 1 для цих

процедур забезпечується режим «тільки запис», а при FileMode = 0 – режим «тільки читання». Змінити режим можна наданням потрібного значення змінній FileMode перед відкриттям файлу. Після цього процедури Reset і Rewrite відкриватимуть файли в заданому режимі. Проте слід мати на увазі, що режим процедури Rewrite управляється у такий спосіб тільки для вже існуючих файлів. Для новостворюваних файлів процедура Rewrite завжди включає режим «читання/запис» (FileMode= 2).

На практиці рідко виникає необхідність втручатися у стандартний порядок процедур. Рекомендується використовувати процедури для тих цілей, які підказуються їх назвами.

Після відкриття файлів введення і виведення в них даних здійснюється процедурами Write і Read, так само як і у разі текстових файлів.

В якості прикладу використання типизованих файлів приведемо програмне рішення наступної задачі.

Приклад 8.4. Програма записує у типизований файл квадрати чисел від 1 до 10. Потім, після введенні чисел від 1 до 10 вона повинна виводити значення їх квадратів з цього файлу.

```
Program Tip_File;
Uses Crt;
Const          L_Min= 1;  L_Max= 10;
Type  Int_N = L_Min..L_Max;
Var    N : Int_N;    {Число для введення}
       R: Real;      {Для зберігання квадратів чисел}
       File_kvadr: File of Real; {Файл для зберігання квадратів чисел}
       Sim: Char;    {Символьна змінна}
Procedure N_Kvadr;  {Створює файл, що містить квадрати десяти чисел типу
Real}
Begin
   Rewrite(File_kvadr);   {Створює новий порожній файл і готує його до запису}
   for N:= L_Min to L_Max do begin
       R := Sqr(N);
       Write(File_kvadr, R);      {Записує числа у файл в двійковому коді}
   end;  {for}
   Close(File_kvadr);
End; N_Kvadr
Procedure Vvod; {Считує з экрану числа в диапазоні L_Min - L_Max }
Begin                  {та видає їх квадрати на экран дисплея}
    Reset(File_kvadr);
    Repeat
       WriteLn('Введіть число від ', L_Min,  ' до ', L_Max);
         ReadLn(N);
       if (N>= L_Min) and (N<= L_Max) then begin
       {Поточна позиція файлу пересувається в задану позицію}
```

```
        Seek(File_kvadr, N - 1);     {Позиція N*N оскільки відлік починається з
      нуля}
        Read(File_kvadr, R);
        WriteLn('Квадрат ',N,' дорівнює ', R:4:0);
      end  {then}
      else  WriteLn('Введене число находитсья за діапазоном');
      WriteLn('Для продовження роботи програми натисніть клавішу Enter') ;
      Write('Для завершення роботи програми введіть s ');
      Read(Sim);
  Until Sim = 's';
  Close(File_kvadr);
End;  {Vvod}
BEGIN
      Assign(File_kvadr, 'Mas_Kv.txt');
      N_Kvadr;
      Reset(File_kvadr);
      Vvod;
      ReadLn;  {Для затримки результату на экрані дисплея}
      END.  { Tip_File }
```

У цій програмі процедура *Assign(File_kvadr, 'Mas_Kv*.txt') зв'язала файлову змінну *File_kvadr* з файлом *'Mas_Kv.txt* ', в якому зберігаються квадрати чисел типу real від 1 до 10 (у двійковому представленні). Цей файл створює підпрограма *N_Kvadr*. Вона відкриває новий файл, записує туди вказані числа, а потім закриває його. Процедура *Reset(File_kvadr)* відкриває цей же файл, але вже для читання. При цьому помітимо, що при закритті файлу в процедурі *N_Kvadr* зв'язок *File_kvadr* з файлом *Mas_Kv.txt* не уривається. Після процедури *Reset* виконується процедура *Vvod*, яка на введення числа від 1 до 10 видає його значення в квадраті з файлу *Mas_Kv.txt*.

Для виходу з вічного циклу необхідно натиснути клавішу *s*.

8.10. Робота з нетипизованими файлами

У Pascal, крім розглянутих файлів, існують також нетипизовані файли. Вони сумісні з усіма типами файлів і використовуються тоді, коли тип елементів файлу неважливий (наприклад, при копіюванні). Такі файли описуються наступним чином:

var имя_файла: **file**;

Наприклад, можливий такий опис:

var FileOneType: **file**;

Файл без типу представляється як послідовність елементів довільного типу, але обумовленого розміру. Це означає, що у файл можна записати значення будь-якої змінної, що має заданий розмір, а при читанні з такого файлу допускається довільна інтерпретація вмісту чергового елементу.

Відкриваються файли без типу тими ж операторами **Reset** і **Rewrite**, але в цьому випадку є другий параметр – розмір запису (елементу файла), заданий в байтах. Попередньо потрібно за допомогою оператора **Assign** зв'язати внутрішнє ім'я файлу із зовнішнім:

```
Assign(FileOneType, 'f.dat');   Reset(fileOneType, 1);
```

Другий параметр операторів Reset і Rewrite може бути опущений, що означає завдання розміру записи в 128 байт. Найбільша швидкість обміну даними забезпечується при довжині записи, кратній 512 байт (розмір сектора на диску).

Слід пам'ятати, що якщо загальний розмір файлу не кратний обраному розміру запису, то останній запис виявиться неповним, і файл може бути прочитаний не до кінця. Цього не буде, якщо задати розмір запису рівним одному байту.

Обмін даними при роботі з нетипизованими файлами здійснюється за участю робочого буферу, в якості якого використовується оголошена в програмі змінна. Її розмір повинен бути достатнім для розміщення даних, які читаються (записуються) за один сеанс читання (запису).

Перед читанням нетипизований файл повинен бути відкритий за допомогою процедури **Reset**, а саме читання здійснюється процедурою **BlockRead**.

```
BlockRead(ім'я_файлу, змінна_буфер, кількість_записів)
```

Третій параметр – це кількість записів (довжина буфера у байтах), читаних за один раз (тип **Word**).

При виконанні процедури BlockRead дані поміщаються в оперативну пам'ять, починаючи з першого байту змінної, вказаної в якості другого параметру процедури BlockRead. Тому **змінна_буфер** повинна мати розмір, що дорівнює добутку, кількості читаних за один раз записів (третій параметр) і розміру запису, заданого у процедурі Reset.

У процедурі BlockRead можливе завдання четвертого параметру (тип **Word**), в якому поміщається число фактично прочитаних записів.

Запис даних в нетипизований файл проводиться тільки після його відкриття за допомогою процедури **Rewrite**. Для запису даних використовується процедура **BlockWrite**, яка має ті ж три (або чотири) параметри, що і **BlockRead**. При цьому в **змінну_буфер** потрібно заздалегідь помістити записи. Кількість цих записів має збігатися зі значенням третього параметру процедури BlockWrite, а розмір – з другим параметром процедури Rewrite. У четвертому параметрі процедури BlockWrite (якщо він є) повертається кількість фактично поміщених у файл записів. Якщо на диску немає вільного місця, то після виконання процедури BlockWrite значення третього і четвертого параметрів будуть відрізнятися.

Якщо при читанні з диску виявиться, що розмір буферу буде менше зазначеного вище або при записі на диск недостатньо вільного місця, то за відсутності четвертого параметру в процедурах BlockRead і BlockWrite буде

зафіксована помилка. При наявності четвертого параметру помилку не буде згенеровано.

Наведемо приклад читання та запису файлу. Повний формат (чотири параметри) процедур наступний: **BLOCKREAD(ff, buff, numb, numblre)**, **BLOCKWRITE(ff, buff, numb, numblre)**.

Тут **ff** – файлова змінна, **buff** – буфер для введення/виведення, **numb** – довжина буфера у байтах, **numblre** – фактична довжина прочитаних байтів. Нижче наводяться приклади використання процедур щодо нетипизованих файлів.

Читання файлу:

```
const blklen=128;
var  f1: file;
       tab: array[0..255] of byte;
       IOReset: byte;
       buff: array[1..blklen] of char;
       blsz, numbl: Longint;
       ind,numblre:integer;
       . . .
      blsz:=1; numbl:= 128;
      assign (f1, 'ggg.pas');
      {SI-}
      reset (f1,1); {open file for read}
      if (IOResult <> 0) then
      begin
         writeln(‘file can’t be open’); readln;
         close(f1);
          halt;
      end;
       BlockRead(f1, buff, numbl, numblre);
        if (IOResult <> 0) then
        begin
            close(f1);
            writeln('file not found'); readln;
            halt;
        end;
        {SI+}
```

В даному прикладі деякий файл ggg.pas зв’язується з файловою змінною f1. Перед відкриттям файлу відключається система діагностики Pascal в разі помилки, а після відкриття аналізується помилка (IOResult). Якщо має місце помилка, то файл закривається і робота програми припиняється. Змінна numbl вказує на кількість фактично прочитаних байт. Якщо вона менша за визначну numbl, то маємо останній запис файлу.

У разі **запису** у нетипизовані файли аналогічно спочатку заповнюється буфер, а потім виконується запис у файл:

```
const blklen=128;
var  f1:file;
     tab: array[0..255] of byte;
     IOReset:byte;
     buff: array[1..blklen] of char;
     blsz, numbl: Longint; ind,numblre:integer;
        . . .
     blsz:=1; numbl:= 128;
     assign (f1, 'ggg.pas');
     {SI-}
     rewrite (f1,1); {open file for write}
     if (IOResult <> 0) then
       begin
          writeln('file can't be open'); readln;
          close(f1);
           halt;
     end;
{встановлення покажчика файла у потрібну позицію та}
{підготовка буферу для запису у файл}
         BlockWrite(f1, buff, numbl, numblre);
          if (IOResult <> 0) then
          begin
             close(f1);
             writeln('file not found'); readln;
             alt;
          end;
          {SI+}
```

Приклад 8.5. Робота програми копіювання файлу на прикладі з нетипизованими файлами:

```
const
  CountRead = 512; // кількість читаних записів
   // При довжині запису в 1 байт - розмір буфера

var
  file_in, file_out: file;
  name1, name2: string;
  num_read, num_write: word; // фактичнго зсчитані/записані записи
  buf: array[1..CountRead] of char; // буфер

begin
```

```
  Write('File 1: ');
  Readln(name1);
  Write('File 2: ');
  Readln(name2);
  Assign(file_in, name1);
{$I-}
  Reset(file_in,1);
{$I+}
 if IOResult <> 0 then begin
     WriteLn('Файл-оригінал не знайдений '); Halt
 end;
 Assign(file_out, name2);
 Rewrite(file_out, 1); // довжина запису 1 байт
 repeat
    BlockRead(file_in, buf, CountRead, num_read);
    BlockWrite(file_out, buf, CountRead, num_write);
 until (num_read = 0) or (num_write <> num_read);
 writeln('Copying is completed');
 if num_read <> num_write then
    writeln('Not enough space');
 close(file_out); close(file_in);
end.
```

Контрольні запитання для самоперевірки

1. Що являє собою файл?
2. Що являє собою файлова змінна та яке її призначення?
3. Які типи файлів обробляються у мові Pascal та як це визначається?
4. Як забезпечується доступ до файлів?
5. Що являють собою логічні файлові пристрої?
6. Що являє собою ініціалізація або відкриття файлу?
7. Як перевірити наявність файлу на диску?
8. Для чого призначена стандартна процедура REWRITE?
9. Для чого призначена стандартна процедура APPEND?
10. Яке призначення процедур CLOSE, RENAME, ERASE?
11. У чому полягають особливості використання функцій EOL та EOF у текстових файлах?
12. Як застосовуються процедури READ, READLN, WRITE та WRITELN?
13. Як прочитати нетипизований файл?
14. Як зробити запис у нетипизований файл?

9. Стандартні модулі

При роботі з мовою програмування ми реально працюємо з комплексом програм, які позволяють програмістам створювати власні програми, користуючись при цьому значною кількістю готових процедур і функцій, бібліотеками визначень типів, констант, змінних.

На даний момент ми знаємо багато стандартних процедур мови Pascal: Write, WriteLn, Read, ReadLn, Sound, Delay, NoSound і т.і. Пригадаємо тепер, що для того, щоб взагалі бути виконаною, програма повинна знаходитися в оперативній пам'яті комп'ютера. Pascal – це теж програма. Значить, для того, щоб ми могли працювати з Pascal, всі стандартні процедури Pascal теж повинні знаходитися в пам'яті? Проте, пам'ять – річ дефіцитна і її треба економити. Добре б у пам'ять завантажувалися не всі процедури, а тільки ті, які потрібні даному програмістові. Дійсно, в багатьох наведених в цьому посібнику програмах ми не використали звук, навіщо ж нам було мати в пам'яті процедури Sound і NoSound?

Реально так і зроблено. У Pascal визначено цілий ряд необов'язкових його програм, які називаються **стандартними модулями**. Спрощено кажучи, стандартний модуль є збіркою процедур і інших елементів, які завантажуються з диска у пам'ять тільки тоді, коли програміст їх спеціально під'єднає. Один з цих модулів називається **CRT** і крім усього іншого займається звуком. Назва CRT походить от *Cathode-Ray Tube* – електронно-променева трубка. Інший називається **Graph** і дає можливість програмістові працювати із зображеннями на екрані. Ці модулі містять описи стандартних констант, процедур і функцій, які використовуються при роботі з монітором у текстовому та графічному режимах. З деякими іншими стандартними модулями ми вже познайомилися.

Якщо програміст спеціально не попросив додаткових функцій, то в пам'яті виявляється тільки мінімальна частина Pascal, яка потрібна більш-менш всім.

А як же попросити? Дуже просто. Модулі підключаються до програми за допомогою оголошення їх імені в директиві компілятора **USES**. Наприклад, якщо вам потрібно працювати із звуком, ви пишете першим рядком своєї програми USES CRT, а якщо ви збираєтеся працювати із зображеннями, то – USES Graph. USES читається " 'юзез", перекладається "використовує". Позначати рядок USES CRT є наказом комп'ютеру "використовувати стандартний модуль CRT", для чого завантажити у пам'ять ті його процедури та інші елементи, які вимагає програма.

Модуль System, в якому визначені основні константи, типи, змінні та підпрограми Turbo Pascal, підключається до програми автоматично (за замовченням). Включення його імені в розділі USES призведе до помилки на стадії компіляції програми.

Отже, ми досить грубо намалювали ідею використання стандартних модулів. Якщо ж програміст бачить, що їх можливостей йому не вистачає, він може написати декілька процедур і створити з них свій власний модуль, придумати йому ім'я і користуватися потім так само, як і стандартними. Але про це ви довідаєтесь тоді, коли ви будете вивчати Pascal детальніше.

Ви дізналися про Pascal (зокрема про модулі) найголовніше і поширеніше. Проте це складає, приблизно, одну четверту частину всіх багатств Pascal, а може й менше. Залишилися приблизно 3/4 частини не викладені в даному посібнику, оскільки вони не так часто вживаються, або дуже складні для вводного курсу.

Контрольні запитання для самоперевірки

1. Що являють собою стандартні модулі?
2. Як стандартні модулі підключаються до програми користувача?
3. Яке призначення модуля GRAPH?
4. Яке призначення модуля CRT?
5. Яке призначення модуля SYSTEM?

10. Комп’ютер звучить

Якщо навіть у вашому комп’ютері немає звукової карти, все одно він може звучати. Подивимося, що примушує його зробити така програма:

```
Uses CRT;
Begin
    Sound(300)
End.
```

Якщо ми хочемо, щоб наш комп’ютер налаштувався на роботу зі звуком, ми повинні першим рядком програми написати Uses CRT. Ми про це вже згадували, але детально про те, що це означає, ми розповімо ще в розділі 8, а зараз не відволікатимемося.

Єдиний оператор програми Sound(300) (слово Sound звучить “‘саунд”, перекладається “звук”) наказує комп’ютеру включити рівний одноманітний звук частотою 300 коливань в секунду (герц). Для тих, хто не знає, пояснимо, що частота визначає висоту звуку. Sound(300) – це звук середньої висоти. Sound(6000) – це звук високий, тонкий, як комариний писк. Sound(40) – звук низький.

Отже, вся дія нашої програми полягає у тому, що включається звук. А що далі? Коли він вимикається? А ніколи! Програма, виконавшись миттєво, припиняє свою роботу, і ми залишаємося наодинці зі звуком. Намагаючись його припинити, ми виходимо з середовища Pascal – не допомагає. Загалом, звук продовжується весь той час, поки комп’ютер включений і ніяк не заважає комп’ютеру правильно працювати. Ми можемо запустити іншу програму – звук супроводжуватиме нас. Найпростіший спосіб позбавитися від звуку – перезапустити комп’ютер. Інший спосіб – виконати програму, в якій є оператор NoSound:

```
Uses CRT;
Begin
    NoSound(300)
End.
```

Оператор NoSound (звучить “ноу ‘саунд”, перекладається “немає звуку”), вимикає звук. Тепер розглянемо таку програму:

```
Uses CRT;
Begin
    Sound(300)
    Delay (2000);
    NoSound
End.
```

Тут ми бачимо новий для нас оператор Delay (2000). Він читається “ди’лей”, перекладається “відстрочення” або “пауза”. Його дія в тому, що він припиняє роботу програми на 2000 мілісекунд або, що те ж саме, на 2 секунди. Delay (1000) припиняє роботу програми на 1 секунду, Delay (500) – на півсекунди і так далі. Відмітимо, що насправді тривалість паузи сильно

залежить від швидкодії комп'ютера. І ще, не на всіх версіях Pascal працює цей оператор. Тому, зазвичай у такому разі оператор Delay замінюють, наприклад, наступною конструкцією:

```
for j:=1 to 32000 do  {Робимо затримку за допомогою двох порожніх циклів}
        for k:=1 to 1500 do;
```

Отже, оператор Sound (300) включає звук. Відразу після цього оператор Delay(2000) припиняє роботу програми на 2 сек. Але звук цей оператор не може вимкнути, комп'ютер продовжує звучати. Через 2 сек програма знову оживає і виконується оператор NoSound. Звук вимикається. Таким чином, результатом виконання цих трьох операторів буде звук частотою 300 Гц тривалістю 2 сек.

У операторах Sound і Delay замість чисел можна записувати цілочисельні змінні величини і вирази. Ось програма, що проводить серію звуків, що поступово підвищуються:

```
Uses  CRT;
Var hz: Integer;
Begin
   hz:= 60;
   while hz < 800 do begin
      Sound(hz);
      Delay(1000);
      hz:=hz+40
   end;
   NoSound
End.
```

Якщо вас цікавлять музичні ноти, то ось вам оператори Sound, що задають всі ноти третьої октави:

Нота до	Sound(523)
Нота до дієз	Sound(554)
Нота ре	Sound(587)
Нота ре дієз	Sound(622)
Нота мі	Sound(659)
Нота фа	Sound(698)
Нота фа дієз	Sound(740)
Нота соль	Sound(784)
Нота соль дієз	Sound(831)
Нота ля	Sound(880)
Нота ля дієз	Sound(932)
Нота сі	Sound(988)

Приклад 10.1. Спробуємо змоделювати сирену міліційної машини: звук вгору – звук вниз – звук вгору – звук вниз – і так кілька разів.

```
Program Sirena;
Uses  CRT;
Var hz, i: Integer;
Begin
    for i:=1 to 3 do begin        {Повторити три рази звук сирени}
       hz:= 60;
       while hz < 800 do begin {Звук вгору}
          Sound(hz);          Delay(1000);        hz:= hz+5
       end;
       while hz > 60 do begin   {Звук вниз}
          Sound(hz);    Delay(1000);    hz:=hz-5
       end;
    end;  {for}
    NoSound
End.
```

Контрольні запитання для самоперевірки

1. Яким чином використовується оператор Sound?
2. Як працює оператор Delay?
3. Як забезпечується застосування певного звуку у програмі на мові Pascal?
4. Як вимикається звук у програмі на мові Pascal?

11. Трохи про графіку

Модуль Graph читається “граф”, це скорочення слова “графіка”. Якщо ми напишемо першим рядком своєї програми USES Graph, то Pascal надасть нам у розпорядження цілу низку процедур та інших засобів, які дозволяють нам малювати на екрані різнокольорові крапки, відрізки прямих, дуги, зафарбовані та не зафарбовані кола, прямокутники, а також виконувати багато інших дій. Користуючись цими можливостями, ви дуже скоро напишете програми, які малюють прості картинки, та примушують зображення рухатися по екрану. За допомогою модуля CRT ми навчимося керувати цим рухом з клавіатури, а отже зможемо створювати свої власні комп’ютерні ігри.

11.1. Текстовий та графічний режими

Існують два режими (способи) роботи комп’ютера з монітором – текстовий та графічний. У будь-якому місці програми ви можете наказати комп’ютеру перемкнутися з одного режиму в інший.

Текстовий режим використовується для виведення на екран текстової та числової інформації. Працюючи у текстовому режимі, комп’ютер вважає екран розбитим на 25 рядків та 80 стовпців, іноді на іншу кількість. У кожній з клітинок, що створились, уміщується рівно одна буква або цифра або будь-який інший символ. Положення символу на екрані задається двома координатами – вертикальною X та горизонтальною Y. X – це номер позиції у рядку, а Y – номер рядка. Який саме символ знаходитиметься у клітинці, визначаєте ви у вашій програмі. Малювати та показувати картинки комп’ютер у текстовому режимі не може. Хоча можна видавати деяку **псевдографіку**.

Наприклад, для порівняння значень елементів масиву зручно будувати стовпчикові діаграми (гістограми). Наприклад, для виведення "N" значень додатних елементів масиву M у вигляді горизонтальної гістограми можна використовувати оператори:

```
k:= 30/M_max;       { k –  нормуючий масштабний коефіцієнт }
                    { M_max –  найбільший елемент масиву M }
for i:=1 to N do  begin
    writeln;        {перехід до нового стовпчика }
    Yg:=round(k*M[i]);    { Yg –  довжина стовпчика гістограми }
    for j:=1 to Yg do  write(#220);    { виведення символу  з кодом 220 }
end;
```

Додавши оператори виведення порядкового номера та значень M[i], отримуємо гістограму у вигляді:

Працюючи у **графічному режимі** комп'ютер вважає екран розбитим на безліч найдрібніших крапок (**пікселів**), кожен з них – набагато дрібніше за клітинку текстового режиму. Комп'ютер не вміє малювати на екрані монітора нічого, окрім цих крапок, що світяться. Проте, розташувавши декілька крапок, що світяться, у рядок впритул одна до одної, отримаємо лінію, а суцільно заповнивши ними деяку область екрану, отримаємо зображення фігури. Положення піксела також задається двома координатами – X та Y. Координата X збільшується зліва направо, а координата Y – зверху донизу. Кількість пікселів на екрані залежить від типу графічного адаптера і для поширеного адаптера VGA складає 640 x 480. Піксели розташовані на екрані стрункими рядками. Кожен піксел за вказівкою програми може бути погашеним або горіти заданим кольором (рис. 11.1).

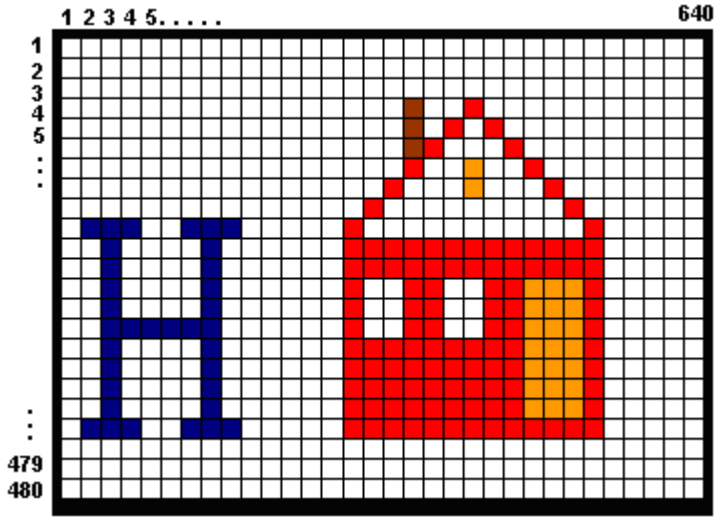


Рис. 11.1. Принцип створення зображення на екрані монітора

На рис. 11.1 ви бачите, що на екрані уміщається 640 стовпців і 480 рядків пікселів. Загальна кількість пікселів виходить рівною 640x480=307200. Мінімальний помітний розмір піксела залежить від якості монітора. Чим більше помітних пікселів уміщується на екрані, тим розмір цих пікселів менший – і тим тонше і правдоподібніше малюнки на екрані. У графічному режимі комп'ютер може малювати картинки та друкувати символи, причому якщо у текстовому режимі величина і форма всіх букв більш-менш однакова, то у графічному режимі ми можемо друкувати символи різної форми та розмірів.

Виходить, що графічний режим краще за текстовий? Загалом, так, звичайно. Але у графічного режиму є один недолік – він вимагає від комп'ютера значних зусиль і тому на малопотужних комп'ютерах часто працює дратівливо повільно.

В усіх написаних раніше програмах ми нічого не говорили про ці режими, ми мовчали про них. Якщо ми у програмі спеціально не просимо, то Pascal завжди обирає текстовий режим за замовченням.

11.2. Модуль CRT

Модуль CRT містить описи констант, процедур та функцій, які забезпечують управління текстовим режимом роботи монітора та звуковим генератором.

Процедури

ClrScr – очищає екран або вікно і поміщає курсор у верхній лівий кут.

Delay(D: Word) – припиняє роботу програми на вказане число D мілісекунд. Практично час затримки залежить від тактової частоти процесора.

GOTOXY(X, Y: Byte) – переміщує курсор в позицію X рядка Y екрану.

NoSound – вимикає джерело звуку.

Sound(F: Word) – запускає джерело звуку з частотою F (Гц).

TextBackGround(Color:Byte) – встановлює колір фону.

TextColor(Color: Byte) – встановлює колір символів.

Window(X1, Y1, X2, Y2: Byte) – визначає текстове вікно на екрані. X1, Y1 – координати лівого верхнього кута вікна, X2, Y2 – правого нижнього кута вікна.

Функції

KeyPressed: Boolean – аналізує натиснення клавіші. Результат TRUE, якщо на клавіатурі натиснута клавіша (окрім Alt, Ctrl тощо), інакше FALSE. Не затримує виконання програми.

ReadKey: Char – читає символ з клавіатури без ехо-повтора на екрані. Припиняє виконання програми до натиснення на будь-яку клавішу, окрім Alt, Ctrl тощо.

Модуль Graph містить константи, процедури та функції для управління графічним режимом роботи монітора.

11.3. Модуль GRAPH

Модуль Graph містить константи, процедури та функції для управління графічним режимом роботи монітора.

Константи кольору

```
Black      = 0;  {Чорний}          DarkGray     = 8;  {Темносірий}
Blue       = 1;  {Синій}           LightBlue    = 9;  {Яркосиній}
Green      = 2;  {Зелений}         LightGreen   = 10; {Яскравозелений}
Cyan       = 3;  {Блакитний}       LightCyan    = 11; {Яскравоголубий}
Red        = 4;  {Червоний}        LightRed     = 12; {Рожевий}
Magenta    = 5;  {Фіолетовий}      LightMagenta = 13; {Малиновий}
Brown      = 6;  {Коричневий}      Yellow       = 14; {Жовтий}
LightGray  = 7;  {Світлосірий}     White        = 15; {Білий}
```

Константи типів і товщини ліній

SolidLn = 0; {Суцільна}	DashedLn = 3; {Пунктирна}
DottedLn = 1; {Точкова}	NormWidth=1; {Нормальна товщина}
CenterLn = 2; {Штрихпунктирна}	ThickWidth = 3; {Потрійна товщина}

Константи шаблону штрихування

EmptyFill = 0;	{Заповнення кольором фону}
SolidFill = 1;	{Суцільне штрихування}
LineFill = 2;	{Горизонтальне штрихування}
LtSlashFill = 3;	{/// штрихування}
SlashFill = 4;	{/// штрихування товстими лініями}
BkSlashFill = 5;	{\\\ штрихування товстими лініями}
LtBkSlashFill = 6;	{\\\ штрихування}
HatchFill = 7;	{Заповнення прямою клітиною}
XHatchFill = 8;	{Заповнення косою клітиною}
InterleaveFill = 9;	{Заповнення частою сіткою}
WideDotFill = 10;	{Заповнення рідкими крапками}
CloseDotFill = 11;	{Заповнення частими крапками}
UserFill = 12.	{Тип задається користувачем}

Процедури

Arc(X, Y: Integer; U1, U2, R: Word) – будує дугу кола поточним кольором з поточними параметрами лінії. X, Y – координати центру дуги, U1 – кут до початкової точки дуги, що відлічується проти годинникової стрілки від горизонтальної вісі, яка направлена зліва направо, U2 – кут до кінцевої точки дуги, що відлічується так само, як U1, R – радіус дуги.

Bar(X1, Y1, X2, Y2: Integer) – будує прямокутник замальований поточним кольором з використанням поточного стилю (орнаменту, штрихування). X1, Y1, X2, Y2 – координати лівого верхнього і правого нижнего кутів прямокутника.

Bar3D(X1, Y1, X2, Y2: Integer; Glubina: Word; Top: Boolean) – будує паралелепіпед, використовуючи поточний стиль та колір. X1, Y1, X2, Y2 – координати лівого верхнього та правого нижнего кутів передньої грані; Glubina – ширина бічної грані (відлічується по горизонталі), Top – ознака включення верхньої грані (якщо True – верхня грань викреслюється, False – не викреслюється).

Circle(X, Y: Integer; R: Word) – Малює поточним кольором коло радіусу R з центром у точці (X,Y).

ClearDevice – Очищує графічний екран, замальовує його у колір фону.

ClearViewPort – Очищує виділене графічне вікно, замальовує його у колір фону.

CloseGraph – Закриває графічний режим, тобто звільняє пам'ять, яка розподілена під драйвери графіки та файли шрифтів, і відновлює текстовий режим роботи екрану.

Ellipse(X, Y: Integer; U1, U2, XR, YR: Word) – Малює дугу еліпса поточним кольором; X, Y – координати центру еліпса; U1, U2 – кути до початкової і кінцевої точок дуги еліпса (див. процедуру Arc); XR, YR – горизонтальна і вертикальна напіввісь еліпса.

FillEllipse(X, Y: Integer; XR, YR: Word) – малює заштрихований еліпс, використовуючи X,Y як центр та XR,YR як горизонтальну та вертикальну напіввісі еліпсу.

FillPoly(N: Word; Var PolyPoints) – Малює та штрихує багатокутник, N вершин, що містить, з координатами у PolyPoints.

InitGraph(Var Driver, Mode: Integer; Path: String) – Організовує перехід у графічний режим. Змінні Driver та Mode містять тип графічного драйверу та його режим роботи. Третій параметр визначає маршрут пошуку графічного драйвера. Якщо рядок порожній (тобто дорівнює ‘‘), вважається, що драйвер знаходиться у поточному каталозі.

Line(X1, Y1, X2, Y2: Integer) – Малює лінію від точки X1, Y1 до точки X2,Y2.

LineTo(X, Y: Integer) – Малює лінію від поточного покажчика до точки X,Y.

MoveTo(X, Y: Integer) – Зміщує поточний покажчик до точки X,Y.

OutTextXY(X, Y: Integer; TextString: String) – Виводить текст у задане місце екрану.

PieSlice(X, Y: Integer; U1, U2, Radius: Word) – Будує сектор кола замальований поточним штрихуванням та кольором заповнення. X, Y – координати центру сектора кола; U1 і U2 – початковий та кінцевий кути сектора, що відлічуються проти годинникової стрілки від горизонтальної вісі, яка спрямована праворуч; Radius – радіус сектора.

PutPixel(X, Y: Integer; Color: Word) – Виводить крапку кольором Color з координатами X, Y.

Rectangle(X1, Y1, X2, Y2) – Малює контур прямокутника, використовуючи поточний колір та тип лінії. X1, Y1 – координати лівого верхнього кута прямокутника, X2, Y2 – координати правого нижнього кута прямокутника.

Sector(X, Y: Integer; U1, U2, XR, YR: Word) – Малює та штрихує сектор еліпса радіусами XR, YR з центром в X, Y від початкового кута U1 до кінцевого кута U2.

SetBkColor(Color: Word) – Встановлює колір фону.

SetColor(Color: Word) – Встановлює основний колір, яким здійснюватиметься малювання.

SetFillStyle(Pattern, Color: Word) – Встановлює зразок штрихування і колір.

SetLineStyle(LineStile, Pattern, Thickness: Word) – Встановлює товщину та стиль лінії.

SetTextStyle(Font, Direction, CharSize: Word) – Встановлює поточний шрифт, напрямок (горизонтальний або вертикальний) та розмір тексту.

SetViewPort(X1, Y1, X2, Y2: Integer; ClipOn: Boolean) – Встановлює прямокутне вікно на графічному екрані. Параметр ClipOn визначає "відсікання" елементів зображення, які не вміщаються у вікні.

Функції

GetMaxX та GetMaxY – Повертають значення максимальних координат екрану в поточному режимі роботи, відповідно, по горизонталі та вертикалі.

GraphResult – Повертає значення GrOk, що відповідне коду 0, якщо всі графічні операції програми виконалися без помилок, або повертає числовий код помилки (від -1 до -14).

11.4. Перемикання між текстовим та графічним режимами

Якщо ви збираєтеся працювати із зображеннями, то повинні перемкнутися у графічний режим і для цього пишете першим рядком програми USES Graph. Але сам по собі цей рядок не є вказівкою комп'ютеру перемкнутися у графічний режим, хоч би тому, що знаходиться у розділі описів, а не у розділі операторів. Для перемикання у графічний режим (або, як то кажуть, для ініціалізації графічного режиму) служить стандартна процедура InitGraph. Для того щоб закрити графічний режим і знов перемкнутися у текстовий, призначена стандартна процедура CloseGraph.

Ось приклад програми, яка спочатку у текстовому режимі пише на екрані текст "Це текстовий режим", потім перемикається у графічний режим, малює коло, а потім знову перемикається у текстовий режим та виводить "Це знову текстовий режим":

```
USESGraph;
VAR      Driver: integer; { драйвер }
         Mode: integer;  { графічний режим }
         Path: string;          { місце розташування драйвера }
         ErrCode: integer; { результат ініціалізації граф. режиму }
BEGIN
         WriteLn('Текстовий режим');
         {Ініціалізували графічний режим:}
         ReadLn;
         Driver:=0;
         InitGraph(Driver, Mode  '<шлях до графічних драйверам>');
         {наприклад, драйвер, файл EGAVGA.BGI, знаходиться у каталозі
d:\bp\bgi }
         ErrCode := GraphResult;
          if ErrCode <> grOk then Halt(1);
         {Звернення до процедури малювання кола}
```

```
  Circle(120,100,40);
  ReadLn;
  {Закриваємо графічний режим}
  CloseGraph;
  WriteLn('Знову текстовий режим');
  ReadLn
END.
```

Перед використанням процедури InitGraph необхідно створити дві змінні величини типу Integer з довільними іменами (тут використані імена Driver та Mode). Значення цих змінних (параметри процедури) при зверненні до процедури InitGraph повинні бути записані всередині круглих дужок. Вам на перших порах абсолютно не обов'язково це знати, але пояснимо, що Driver означає тип вашого відеоадаптера (CGA, EGA, VGA або інший), а Mode означає номер графічного режиму. Якщо ви нічого не знаєте ані про те, ані про інше, сміливо пишіть Driver := 0 і Pascal сам визначить тип вашого відеоадаптера і встановить наймогутніший з допустимих графічний режимів.

Третій елемент – не що інше, як шлях до графічних драйверів Pascal. Ми використовували кутні дужки <>, щоб підкреслити, що у вашій програмі потрібно писати не ті чотири слова, що всередині кутових дужок, а те, на що вони вказують. Майже напевно для вас графічний драйвер представлений файлом egavga.bgi, який розташований у каталозі BGI. Якщо сам Pascal розташований у каталозі TP диску C, то рядок вашої програми виглядатиме таким чином:

InitGraph(Device, Mode, 'c:\TP\BGI');

Якщо ви його записали вірно, то можете спробувати запустити програму, і при правильному налаштуванні Pascal у вас все вийде.

Якщо Pascal при запуску графічного режиму скаржиться, що не може знайти модуль (File not found (GRAPH.TPU)), то перевірте, що записано у меню **Options → Directories ... → Unit directories**. Там повинен бути вказаний шлях до файлів стандартних модулів та, зокрема, до модуля GRAPH.TPU. Вони зазвичай розміщені у каталозі Units головного каталога Pascal.

Функція GraphResult оператора ErrCode:=GraphResult повертає код помилки при встановленні графічного режиму. Цей оператор можна не вставляти у програму, але краще це зробити про всяк випадок, дотримуючись певної культури програмування.

Якщо код помилки не дорівнює вбудованій змінній grOk (if ErrCode <> grOk then Halt(1)), то програма припиняє свою роботу оператором Halt(1).

Відмітимо, що при перемиканні режимів весь вміст екрану очищується.

11.5. Малюємо прості фігури

Для простоти будемо вважати, що графічний режим вашого монітора вважає екран розділеним на 640 пікселів завширшки та 480 по висоті, і виходячи з цього, будемо створювати програми. Якщо у вас інший режим, то ви

легко зможете зробити поправки у програмах. Намалюємо точку та прямокутник. Для цього ми запишемо звернення до двох процедур:

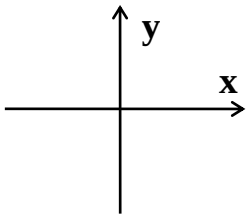
Для **точки** – **PutPixel(14,6, Yellow)**, для **прямокутника** – **Rectangle(4,2,16,10).**

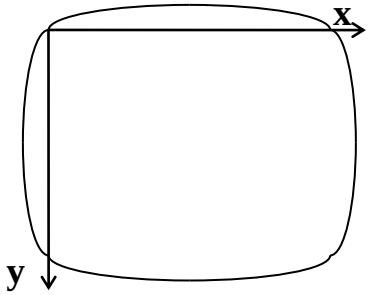
Як бачмио, при зверненні до процедури PutPixel у дужках треба вказувати три елементи (параметри процедури), у Rectangle – чотири.

У PutPixel перший параметр – **горизонтальна координата точки**, другий – **вертикальна**, третій – **колір точки** (жовтий). У Rectangle перша пара параметрів – координати будь-якого з кутів прямокутника, друга пара – координати протилежного йому кута (тільки не сусіднього). У кожній парі першою йде горизонтальна координата, другою – вертикальна. Колір прямокутника задається в іншій процедурі, про яку розмова буде йти пізніше.

У зверненнях до процедур, що мають справу з кольором, замість англійської назви кольору можна писати відповідне число. Наприклад, замість PutPixel(14,6, Yellow) можна написати PutPixel(14,6, 14).

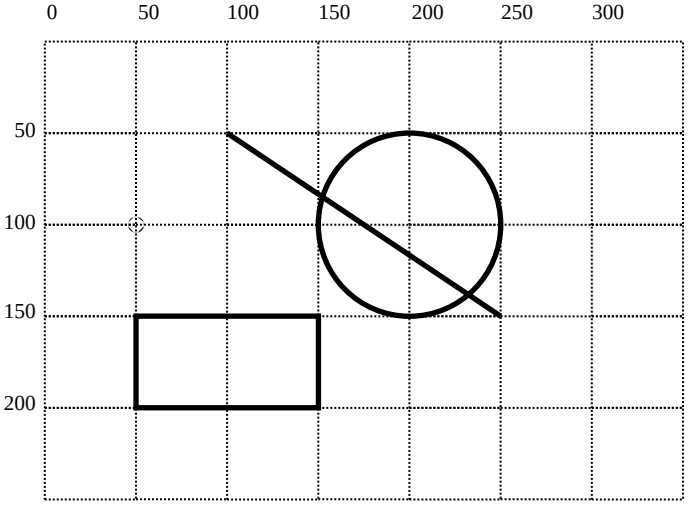
***У школі ви звикли до такої системи координат*: *У комп'ютері ж застосовується така*:**





Як бачите, вісь *Y* направлена донизу.

Напишемо програму, яка малює точку, прямокутник, коло та відрізок прямої так, як це показано на малюнку:



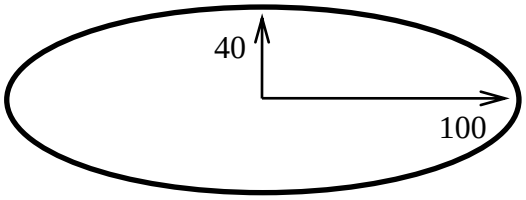

```
USESGraph;
VAR     Driver: integer; { драйвер }
        Mode: integer;  { графічний режим }
```

```
    Path: string;           { місце розташування драйвера }
    ErrCode: integer; { результат ініціалізації граф. режиму }
BEGIN
    Driver:=0;
    Path := 'd:\bp\bgi';
    InitGraph(Driver,  Mode,  Path);
    ErrCode := GraphResult;
     if ErrCode <> grOk then Halt(1);
    PutPixel(50,100,White); {малюємо крапку}
    Rectangle(150,150,50,200);     {правий верхній і лівий нижний кути}
    Circle(200,100,50);            {коло}
    Line(100,50,250,150);    {відрізок прямої}
    ReadLn;
    CloseGraph
END.
```

Коло можна побудувати, якщо відоме положення центру та радіус. Коло малюється процедурою Circle, перші два параметри якої – координати центру, третій – радіус.

Відрізок прямої малюється процедурою **Line**. Ми знаємо, що відрізок прямої можна побудувати, якщо відоме положення його двох крайніх точок. Вони і задаються після звернення до процедури. Перша пара параметрів – координати однієї точки (будь-якої з двох), друга пара – іншої.

Спробуємо тепер намалювати еліпс з центром у точці x=200, y=150 такого вигляду:



Це виконає процедура Ellipse. Ось її виклик – Ellipse(200,150,0,360,100,40). Якщо ви хочете намалювати не повний еліпс, а тільки частину його, замість 0 вкажіть початковий кут дуги еліпса, наприклад 90, а замість 360 – кінцевий кут, наприклад 180.

Процедура ClearDevice стирає все з екрану у графічному режимі.

11.6. Стиль ліній та заливки. Робота з кольором

Спочатку поговоримо про стиль ліній. Якщо ви хочете, щоб лінії, якими кресляться фігури, були товщими, використовуйте процедуру SetLineStyle. Ось її виклик – SetLineStyle(0, 0, ThickWidth). Це наказ відтепер малювати лінії товстими. Якщо ви хочете повернутися до тонких ліній, напишіть SetLineStyle(0, 0, NormWidth).

Перший параметр процедури SetLineStyle керує стилем відрізків прямих (штрихові, пунктирні і ін.). Спробуйте замість 0 написати 1, 2 або 3 і подивіться, що з цього вийде. Другого параметра розглядати поки не будемо.

А зараз поговоримо про колір. Якщо колір точки задається безпосередньо у виклику процедури, яка малює точку (PutPixel), то кольори багатьох інших фігур задаються інакше. Робить це спеціальна процедура SetColor. Приклад: SetColor(Yellow) наказує комп'ютеру відтепер малювати фігури жовтим кольором.

Ось пояснюючий фрагмент програми:

```
Circle(100,100,20);          {коло біле та тонке, оскільки враховуються замовчення}
                             {Pascal все малює білим кольором тонкими лініями}
SetColor(Yellow);
Circle(150,100,20);          {жовте тонке коло }
SetLineStyle(0, 0, ThickWidth);
Circle(200,100,20);          {жовте товсте коло}
SetColor(Blue);
SetLineStyle(0, 0, NormWidth);
Circle(250,100,20);          {синє тонке коло}
```

Працюючи у Pascal, ви можете пофарбувати будь-яким кольором (залити фарбою) будь-яку область екрану всередині замкнутого контура.

Як фарбувати? Почнемо з процедури SetFillStyle. Подібно до того, як процедура SetColor сама нічого не малює, а тільки встановлює колір ліній на майбутнє. Наприклад, SetFillStyle(1, Green) встановлює заливку зеленим кольором. На одиницю поки не звертаємо уваги.

Сам процес заливки виконує процедура FloodFill. Наприклад, FloodFill(200, 150, Blue) викликає дії Pascal, що розглядуються нижче.

У точці екрану з координатами x=200 і y=150 встає "маляр з відром фарби", колір якої був вказаний у зверненні до процедури SetFillStyle (у нашому прикладі – зелений), і починає цю фарбу розливати навколо себе. Фарба вільно розтікається по екрану і перешкодою для неї стає тільки лінія, що має колір, вказаний у зверненні до процедури FloodFill, тобто синій. Очевидно, якщо синій контур не буде замкнутий навколо "маляра", то фарба проллється на весь екран.

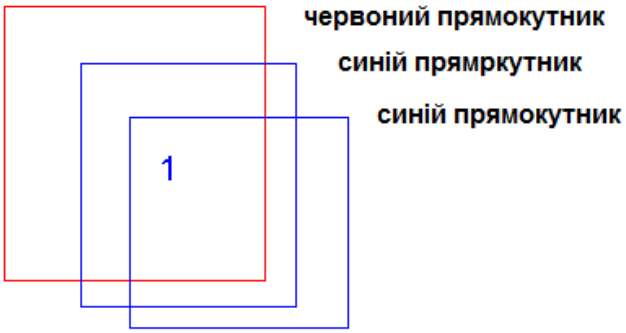
Наведемо фрагмент програми, яка намалює жовтий квадрат і зафарбує його червоним кольором:

```
SetColor(Yellow);
SetFillStyle(1, Red);
Rectangle(200,50,300,150);
FloodFill(250, 100, Yellow);
```

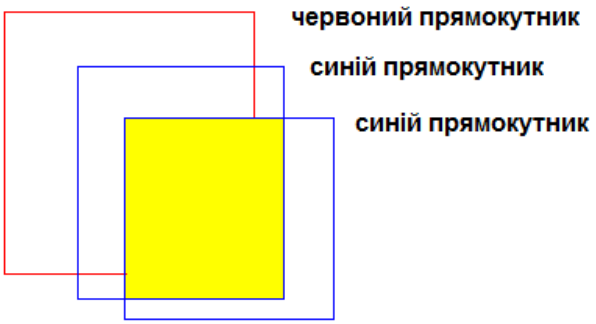
А зараз спробуємо поставити "маляра" поза квадратом, записавши в нашому фрагменті замість FloodFill(250, 100, Yellow) наприклад, FloodFill(250, 200, Yellow). Тепер червоною буде вся поверхня екрану, окрім області всередині квадрата.

Тому, на питання про те, яким кольором фарбуватиме оператора FloodFill(250, 100, Yellow), необхідно відповідати не “Жовтим”, а переглянути попередній оператор SetFillStyle”, в якому вказаний колір заливки.

Нагадаємо ще раз, що у нашому випадку тільки жовті лінії та області є перешкодою для червоної фарби. Наприклад, спробуємо пофарбувати жовтою фарбою маленький прямокутник, помічений одиницею, отриманий перетином великих червоного та синього прямокутників:



Для цього помістимо “маляра” всередину цього прямокутника та використаємо оператори SetFillStyle(1, Yellow) та FloodFill((, (,Blue). Ми отримаємо таку картину:



Отже, за допомогою FloodFill потрібний прямокутник зафарбувати неможливо, оскільки він обмежений лініями різного кольору.

Заливку можна робити не тільки суцільну, але й заштрихованою. Для цього замість одиниці у зверненні до процедури SetFillStyle спробуйте інші цілі числа – 2, 3 та ін.

11.7. Використання випадкових величин при малюванні

Як отримати випадкове число ми вже знаємо. Наприклад, оператор WriteLn(Random(100)) надрукує ціле число, яке ми заздалегідь не знаємо, знаємо тільки, що воно менше 100. Легко здогадатися, що WriteLn (200 + Random(100)) надрукує випадкове число з діапазону від 200 до 299.

А зараз спробуємо намалювати “зоряне небо”. Для цього достатньо у випадкових місцях екрану намалювати деяку кількість різноколірних точок (наприклад, 1000). Точка ставиться процедурою PutPixel. Як зробити

координати та колір точки випадковими? Використовуємо той самий Random. Якщо ваш екран має розмір 640x480 пікселів, то фрагмент програми для малювання однієї точки випадкового кольору виглядатиме так:

```
Randomize;
for i:=1 to 1000 do
    PutPixel (Random(640), Random(480), Random(16));
```

Для того щоб від запуску до запуску набір значень випадкової величини змінювався (а це й означає “зоряне небо”), ми використали процедуру **Randomize до** функції Random . Число 16 у Random(16) взято з тієї причини, що всі кольори у Pascal мають номери від 0 до 16. Для того щоб таких точок було 1000, використовуємо цикл for.

11.8. Виведення текстової інформації

Для виведення тексту на екран використовуються дві процедури:

1. **OutText(S: string)**. Ця процедура виводить рядок S починаючи з поточної позиції. За замовченням поточною позицією для виведення є лівий верхній кут рядка. Поточна позиція задається, наприклад, за допомогою MoveTo.
2. **OutTextXY(x,y: integer; s: string)**. Використовується для виведення рядка у конкретній позиції.

Якщо потрібно вивести числа, то заздалегідь необхідно перетворити їх у рядок, наприклад, за допомогою процедури Str.

Приклад:

```
var r: integer;
    s: string;
. . . .
Str(r,s);
OutTextXY(100,200,’Результат=‘+s);
```

TURBO-PASCAL дозволяє використовувати декілька різних шрифтів для виведення тексту. Крім того, можна міняти напрям виведення тексту, а також розмір символів. З цією метою використовується процедура SetTextStyle(Font, Direction, CharSize : word). Перерахуємо можливі константи та значення для параметрів цієї процедури.

Font (шрифт):

DefaultFont – шрифт 8x8 (за замовченням);
TriplexFont – напівжирний шрифт;
SmallFont – тонкий шрифт;
SansSerifFont – шрифт без засічок;
GothicFont – готичний шрифт.

Direction (орієнтація та напрям виведення символів):

0 – звичне виведення зліва направо;
1 – від низу до верху (напис «покладений на бік»);

2 – зліва направо, але «лежачими» літерами.
Size (розмір шрифту (цілі числа від 0 до 10).

Інша можливість при роботі з текстом (це вирівнювання його відносно координат виведення, які задаються. Для цього використовується процедура SetTextJustify(horiz,wert: word). Horiz указує як текст, який розташований відносно заданої позиції по горизонталі, а vert – по вертикалі. Можливі константи –
для horiz:
LeftText – вказана позиція є лівим краєм рядка;
CenterText – позиція є серединою рядка, що виводиться;
RightText – правим краєм рядка.
для vert:
BottomText – позиція знаходиться на нижньому краю зображення;
CenterText – по центру;
TopText – позиція є верхнім краєм зображення.

11.9. Рух картинок по екрану

Про створення ілюзії руху картинок по екрану ми розглядали раніше. Зараз нагадаємо основну його ідею.

Візьмемо гру, наприклад, гонки автомобілів. Тут потрібно задатися питанням, а що таке рух? Якщо ви тримали в руках кіноплівку фільму, що зображає, скажімо, рух автомобіля, то повинні були звернути увагу, що вона складається з безлічі нерухомих слайдів (кадрів), на кожному наступному з яких автомобіль знаходиться трохи в іншому місці, ніж на попередньому. Показуючи ці кадри один за одним з великою швидкістю, ми створюємо ілюзію руху автомобіля. Так само поступають зі створенням руху на екрані комп'ютера. Запишемо алгоритм руху по екрану зліва направо звичайного кола:

1. Задамо позицію кружечка у лівій частині екрану. Перейдемо до команди 2.
2. Намалюємо кружечок. Перейдемо до команди 3.
3. Зітремо його. Перейдемо до команди 4.
4. Змінимо позицію кружечка на міліметр правіше. Перейдемо до команди 5.
5. Перейдемо до команди 2.

У короткому проміжку часу після виконання команди 2 коло з'являтиметься на екрані і на команді 3 зникати, але це досить, щоб людське око його відмітило. Завдяки циклу коло миготітиме кожного разу у новому місці, а оскільки зміна "кадрів" буде дуже швидкою, нам буде здаватися, що відбувається плавний рух кола.

Спробуємо змусити рухатися по екрану коло зліва направо. Для цього ми повинні спочатку намалювати зліва коло, а потім стерти його, тобто необхідно знов намалювати її, але чорним кольором. Не дивлячись на те, що ми коло одразу ж стерли, воно встигне промайнути на екрані, і око це відмітить. Потім

треба намалювати та стерти таке саме коло трохи правіше, потім ще правіше і так далі. Ось програма:

```
USESCRT, Graph;
VAR        Driver: integer; { драйвер }
       Mode: integer;  { графічний режим }
       Path: string;          { місце розташування драйвера }
       ErrCode: integer; { результат ініціалізації графічного режиму }
       x, shag: integer;
BEGIN
       Driver:=0;
       Path := ‘d:\bp\bgi’;
       InitGraph(Driver,   Mode,   Path);
       ErrCode := GraphResult;
        if ErrCode <> grOk then Halt(1);
       ReadLn;      {Перемикання у графічний режим; іноді займає}
                    { одну-дві секунди, тому, якщо ви хочете побачити}
                    { рух із самого початку, клацніть по клавіші введення }
       x :=40;  Shag:=1;
       repeat
          SetColor(White);
          Circle(x,100,10);        {Малюємо біле коло}
          SetColor(Black);
       Delay(500);
       Circle(x,100,10);   {Малюємо чорне коло}
          x:=x+shag                {Переміщаємося трохи направо}
       until x>600;
       CloseGraph
END.
```

Коли ви спробуєте виконати цю програму на комп’ютері, зображення рухомого кола може вийти неякісним – коло у процесі руху може мерехтіти та пульсувати. Це пов’язано з розгорткою електронно-променевої трубки вашого монітора. Спробуйте змінити радіус кола або крок руху по горизонталі або введіть між малюванням і стиранням кола невелику паузу за допомогою процедури Delay (ця процедура знаходиться у модулі CRT) – ситуація майже напевно покращає. Якщо Delay з якоїсь причини не працює, то скористуйтеся режимом затримки за допомогою порожніх циклів (наприклад, for j:=1 to 32000 do for k:=1 to 1500 do;).

Якщо ви захочете, щоб коло поверталося назад, то необхідно перед оператором CloseGraph (після until x>600;) вставити схожий фрагмент програми:

```
   x:=600; {Починаємо переміщення з правої частини екрану }
   repeat   {Вкладений цикл для руху наліво}
       SetColor(White);
       Circle(x,100,10);   {Малюємо біле коло}
```

```
    SetColor(Black);
    Delay(500);
    Circle(x,100,10);   {Малюємо чорне коло}
     x:=x-shag          {Переміщуємося трохи наліво}
  until x<40;
```

Припустимо, ми хочемо, щоб кулька літала втричі швидше. Для цього нам досить в двох місцях програми замість shag:=1 написати shag:=3 (або зменшити час затримки в операторові Delay).

11.10. Управління комп'ютером з клавіатури

Давайте спробуємо запустити програму, яка виконує фрагмент програми довго, не звертаючи на вас уваги, щоб ви не робили. Наприклад, таку:

BEGIN repeat WriteLn('Я довго гнатиму велосипед.') until 1>2 END.

Ви сидите перед комп'ютером і чекаєте, коли ж він закінчить друкувати свій текст. А він ніколи не закінчить. Ви починаєте стукати по клавішах, сподіваючись, що це перерве безглуздий цикл. Марно. Правда, тільки коли ви, утримуючи натиснутою клавішу Ctrl, клацнете по клавіші Break, програма припинить свою роботу.

Поки програма працює, вона не реагує на клавіатуру, якщо ви про це спеціально не потурбуєтесь. А щоб потурбуватися, ви повинні включити у програму спеціальні функції ReadKey та KeyPressed з модуля CRT. Доповнимо нашу «уперту» програму парою рядків:

```
USES CRT;
BEGIN
  Repeat
    if  KeyPressed then     WriteLn('Увага! Натиснута клавіша.')
    else  WriteLn('Я довго гнатиму велосипед.')
  until 1>2
END.
```

Наштовхнувшись на вираз *if KeyPressed then*, Pascal перевіряє, чи була натиснута клавіша на клавіатурі. Коли ви запустите цю програму, вона нескінченно друкуватиме *Я довго гнатиму велосипед*. Але як тільки ви клацнете по якій-небудь клавіші, програма почне нескінченно друкувати *Увага! Натиснута клавіша*.

Якщо функція *KeyPressed* просто реагує на те, чи була натиснута будь-яка клавіша, то функція *ReadKey* повідомляє, яка саме клавіша була натиснута.

Наведемо приклад програми, яка нескінченно друкує текст *А нам все одно*! і одночасно безперервно чекає натиснення на клавіші, і як тільки клавіша натиснута, одноразово доповідає, чи була натиснута будь-яка клавіша, після чого продовжує друкувати *А нам все одно*! Якщо ми захочемо, щоб програма закінчила роботу, ми повинні натиснути на клавішу "s".

```
USES CRT;
VAR  klavisha : Char;
```

```
    j, до: integer;
BEGIN
  Repeat
    Delay (1000);  {щоб програма не друкувала дуже швидко}
    WriteLn('Я довго гнатиму велосипед.');
    if  KeyPressed then     begin
      klavisha:= ReadKey;
      WriteLn('Натиснута клавіша ', klavisha);
      {щоб встигнути прочитати яка клавіша натиснута}
      for j:=1 to 32000 do for k:=1 to 32000 do;
    end
  until klavisha='s'
END.
```

Наша програма нескінченно друкуватиме *Я довго гнатиму велосипед.*, а при кожному натисненні на клавішу одноразово повідомлятиме *Натиснута клавіша...* Чому одноразово, а не нескінченно? Це можна пояснити тим, що після виконання функції ReadKey Pascal "забуває", що на клавіатурі була натиснута клавіша.

Ви запитаєте, а навіщо тут взагалі потрібний *рядок if KeyPressed then?* Річ у тім, що, якщо перед виконанням функції ReadKey клавішу на клавіатурі не натиснути, то функція ReadKey зупиняє програму і примушує її чекати натиснення на клавішу, а після натиснення програма продовжує роботу. Якщо вам у вашій програмі паузи не потрібні, то вам доведеться використовувати KeyPressed.

Функція ReadKey нагадує процедуру ReadLn. Проте у неї є цікава відмінність: при введенні символів для процедури ReadLn вони з'являються на екрані, а при введенні символу для функції ReadKey – ні. Завдяки цій властивості, за допомогою ReadKey можна організувати "секретне введення" інформації в комп'ютер.

Контрольні запитання для самоперевірки.

1. Які існують режими роботи дисплея для комп'ютера?
2. Які основні процедури та функції використовуються для роботи дисплея у текстовому режимі?
3. Які дані містить у собі модуль GRAPH для роботи дисплея у графічному режимі?
4. Які основні процедури та функції використовуються для роботи дисплея у графічному режимі?
5. Як здійснюється перемикання режимів роботи дисплею?
6. Яким чином малюються стандартні фігури у графічному режимі?
7. Що являють собою стиль ліній та їхня заливка?
8. Як здійснюється виведення текстової інформації у графічному режимі?
9. Як забезпечується рух зображень на екрані дисплея?
10. Як забезпечується управління комп'ютером з клавіатури?

12. Структуризація у програмуванні

У нас вже неодноразово був привід згадати структурне програмування. Настав момент обговорити це явище по суті. До кінця 1960-х років, коли з'явилася ця технологія у світі було створено низку крупних програмних розробок, у ході яких виявилася недосконаліть існуючої технології програмування. Програми страждали від великої кількості помилок, на пошук та усунення яких витрачалося багато часу та ресурсів.

Для перших ЕОМ з обмеженою пам'яттю і невеликою швидкодією основним показником якості програми була її економічність по займаній пам'яті і часу рахунку. Чим програма виходила коротше, тим клас програміста вважався вище. Таке скорочення програми часто вимагало великих зусиль. Іноді програма виходила настільки «хитрою», що могла «перехитрити» самого автора: повернувшись через деякий час до власної програмі і бажаючи щось змінити, програміст міг заплутатися в ній, забувши свою «геніальну ідею». З ростом пам'яті і швидкодії ЕОМ, з удосконаленням мов програмування і трансляторів з цих мов проблема економічності програми стає менш гострою. Все більш важливими якісними характеристиками програм стають їх простота, наочність і надійність, а з появою машин третього покоління ці якості стають основними.

Встановлені терміни розробок постійно зривалися, а заплановані бюджети багато разів перекривалися. Продуктивність праці програмістів виявлялася низькою та оцінювалася у 5-10 відлагоджених операторів на людину у день. Це, звичайно, не означає, що програмісти у ті часи, написавши декілька рядків за день, відпочивали. Рахунок ведеться по готовій програмі. Така цифра виходила, якщо розділити кількість операторів у готовій та відлагодженій програмі на час її розробки та на кількість учасників розробки.

Значне збільшення складності завдань, які вирішуються за допомогою комп'ютерів, призводить до збільшення розмірів та складності програм, що породжує труднощі при їх розробці та відлагодженні. Отже, наприклад, одним з найбільших програмних проектів того часу була розробка операційної системи для комп'ютерів серії IBM/360. При створенні цієї системи повною мірою проявили себе всі перераховані проблеми. Для її створення було залучено більше 1000 осіб одночасно, а трудомісткість склала 5000 "людино-років". Терміни та бюджет були значно перевищені. При цьому у вже готовій системі залишалося дуже багато помилок.

Певний загальноприйнятий спосіб виробництва будь-чого називають технологією. Для розв'язання проблем, які виникають при цьому, у практиці програмування була вироблена низка прийомів та методів, які прийнято також називати **технологією програмування**, основою якої була розробка структурованих програм, які володіють властивостями простоти їх читання та розуміння.

Зокрема, прикладами таких технологій, які базуються на ***методах структурного програмування,*** є ***технологія спадаючого*** та ***технологія висхідного програмування***.

12.1. Характеристика сучасної практики програмування

Мова Object Pascal, як одна із сучасних мов програмування, відображає специфіку та проблематику поточного етапу розвитку програмування. Тут слід виділити принаймні три моменти.

1. Виникло таке порівняно нове поняття, як «програмний продукт» або «програмний виріб». Програма, яка виготовлена в одному колективі або, що на даний час рідше, однією людиною, відчуджується від нього та передається (продається) для використання назовні.

Ця обставина істотно підвищує вимоги до надійності програмного виробу, тобто до зменшення кількості помилок у програмі, які залишилися не виявленими, та таких неврахованих ситуацій, при виникненні яких програма може видавати невизначений результат або взагалі припиняти своє нормальне функціонування. Особливого значення набуває також ефективність програми. Недостатня ефективність при багаторазовому подальшому використанні цієї програми може призвести до вельми істотних непродуктивних витрат машинного часу на її виконання. Наприклад, потрібно відповісти на питання, чи всі елементи масиву A менше елементів масиву B того ж розміру? Можливий фрагмент програми

```
R:=0;
for i:= 1 to N do
     for j:= 1 to N do
        if A[i] > B[j] then R:=1;
if  R=0 then write (‘Yes’)
else write (‘No’);
```

демонструє неефективність роботи особливо при великому значенні N, оскільки при першому виконанні умови, що перевіряється, подальше виконання операторів циклу можна було б припинити.

2. Має місце значне збільшення складності завдань, що вирішуються за допомогою ЕОМ. Це породжує додаткові труднощі при розробці та складанні програми, а також при її відлагодженні.

3. Істотно збільшується тривалість «**життєвого циклу»** програми, тобто того часу, на протязі якого програма проектується (розробляється), створюється та використовується. З часом змінюються умови, в яких використовуються ці програми, а це вимагає регулярної модифікації з метою їх пристосування до умов, що змінилися, зміни спочатку запланованих можливостей програм, підвищення їхньої ефективності, зручності користування ними і таке інше.

Отже, останнім часом істотно підвищилися вимоги до надійності програм при зростанні їх розмірів та складності, а також до зручностей їх подальшого супроводу. Ця ситуація породила низку нових проблем та труднощів. Для їх

подолання практика програмування напрацювала низку методів та прийомів, зокрема таких, які можна охарактеризувати терміном «**структуризація».**

Якщо у програмі використовується велика кількість змінних різного типу, то для створення надійної комп'ютерної програми треба уважно стежити за тим, щоб кожній змінній присвоювалися лише значення відповідного типу. Такий контроль бажано покласти на транслятор. Для цього він повинен мати вичерпну інформацію про використаня у даній програмі величини та про тип значень кожної з них.

У зв'язку з цим Pascal передбачає низку заходів організаційного, «структурного» характеру для того, щоб надати транслятору можливість проконтролювати коректність використання у програмі тих чи інших програмних об'єктів.

Наприклад, не дивлячись на різноманітність припустимих типів даних, у мові Pascal послідовно витримана вимога про те, щоб тип будь-якої константи однозначно визначався за її написом. Таку саму мету переслідує й вимога, щоб кожен програмний об'єкт, у тому числі і кожна змінна, були заздалегідь чітко описані, зокрема, щоб для кожної змінної при використанні був указаний цілком певний тип. З цієї ж причини у Pascal передбачені заходи, які унеможливлюють вироблення неоднозначно отримуваного результату. Отже, якщо i – цілочисельна змінна, то оператор присвоєння є неприпустимим

i := A,

де A – дійсне число. Неоднозначність полягає у тому, що при A = 2.5 змінна i може отримати або значення 2. або значення 3. в залежності від точності обчислення значення A. Це може призвести до абсолютно різних остаточних результатів. Тому Pascal вимагає, щоб подібного роду перетворення були зроблені у програмі в явному вигляді, для чого у мові передбачені, як ви вже знаєте, спеціальні стандартні функції.

Для того, щоб легко було використовувати подібну інформацію не лише транслятору та авторові програми, але й іншим особам, Pascal вимагає чіткої структуризації програми. Та чи інша інформація повинна знаходитись у строго визначеному для неї місці.

Важливе значення має також структуризація даних. Оскільки програма задає правила обробки даних, то проектування самих даних при розробці програми має не менш важливе значення, ніж проектування правил їхньої обробки.

Pascal дозволяє програмістові вводити при вживанні та використовувати у своїй програмі найзручніші для нього структури даних. При цьому мова знову ж таки вимагає від програміста абсолютно чіткого опису кожної структури даних, яка вводиться для вживання, що дозволяє транслятору забезпечувати роботу з кожною такою структурою та стежити за коректністю її використання. Визначення тієї або іншої структури даних міститься в описі відповідного типу даних (найчастіше у розділі опису типів).

Одна із труднощів розробки надійної програми полягає у тому, що не існує формального апарату для отримання потрібного алгоритму. Для будь-якого

завдання часто існує багато алгоритмів його розв’язку, кожен з яких має свої переваги та свої недоліки. Тому розробка алгоритму та програми значною мірою проводиться методом «спроб та помилок». Отриманий з тих чи інших міркувань варіант алгоритму підлягає перевірці його на правильність та аналізу його на ефективність. Це складає серйозну проблему.

Для зменшення труднощів, які виникають тут, останніми роками все більш широке визнання та практичне використання отримують ідеї структурного програмування.

12.2. Розробка структурованих програм

На початку 70-х рр.. XX в. визначається дисципліна, що отримала назву структурного програмування. Її поява і розвиток пов'язаний з іменами Е.В. Дейкстри, Х.Д. Мілса, Д.Є. Батога і інших. Структурне програмування до теперішнього часу залишається основою технології програмування. Дотримання його принципів дозволяє програмісту швидко навчитися створювати ясні, безпомилкові, надійні програми.

Під структурним програмуванням розуміють такі методи розробки та написання програми, які орієнтовані на максимальну зручність сприйняття та розуміння її людиною. Ці ідеї з’явилися з усвідомленням того факту, що «програми пишуться для людей – на машині вони лише обробляються». Сенс сформульованого положення полягає у тому, що обробка програми на комп’ютері, тобто її трансляція та виконання, практично не викликає труднощів. Роботу ж по перевірці правильності роботи програми, внесенню до неї різного роду виправлень та змін доводиться виконувати людині. Таким чином, програма має бути складена та написана так, щоб максимально полегшити та спростити цю роботу. Тому суть концепції структурного програмування Г. Майерс (Надежность программного обеспечения. М.: «Мир», 1980. – 360 с), наприклад, формулює наступним чином: «Я вважаю за краще визначати структурне програмування як програмування, яке орієнтоване на спілкування з людьми, а не з машиною».

Ідеї структурного програмування базуються на усвідомленні того факту, що людина набагато легше читає та розуміє будь-який текст у тому випадку, якщо вона читає фрази у порядку їх дотримання у цьому тексті. Якщо ж по ходу читання її будуть досить часто посилати на фрагменти тексту, які знаходяться, наприклад, на інших сторінках, то це різко ускладнює сприйняття та розуміння прочитаного тексту. Таким чином, при прочитанні програми в її послідовних фрагментах повинна чітко простежуватись логіка роботи, тобто не повинно бути “стрибків” на інші фрагменти програми, які розташовані десь в іншому місці. Тому структурне програмування інколи називають «програмуванням без операторів переходу» (або «програмуванням без **goto**”). Йдеться не про заборону оператора переходу, а про те, щоб не використовувати операторів переходу без особливої на те необхідності.

Структурне програмування зосереджено на логіці програми і включає три головні складові:

1. Проектування зверху донизу.
2. Модульне програмування.
3. Структурне кодування.

Проектування зверху донизу передбачає спочатку визначення завдання у загальних рисах, а потім поступове уточнення структури, шляхом внесення більш дрібних деталей. На кожному кроці такого уточнення необхідно виявити основні функції, які потрібно виконати. Таким чином дана задача розбивається на ряд підзадач, поки ці підзадачі не стануть настільки простими, що кожній з них буде відповідати один модуль. Дії кожного модуля мають бути описані однією фразою.

Проектування має бути завершено до початку програмування.

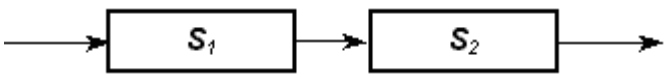
Модульне програмування – це процес поділу програми на логічні частини (модулі) і послідовне програмування кожної з них. Кожен модуль повинен мати своє призначення, бути замкненим, вхід і вихід повинні бути точно визначені.

Структурне програмування є методом складання добре структурованих програм, зручних для їх читання та розуміння людиною, дослідження логіки їх роботи, внесення в них виправлень та інших змін. Розробка структурованих програм, які володіють властивостями простоти їх читання та розуміння, досягається за рахунок того, що будь-яку програму пропонується будувати із стандартних логічних структур невеликої кількості типів.

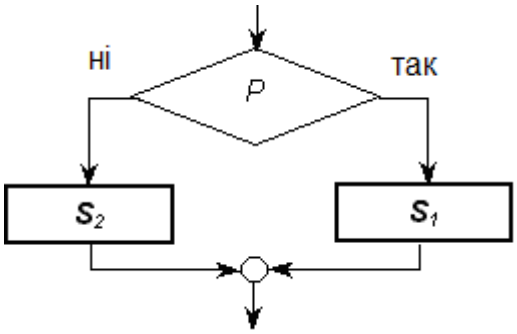
Структурне кодування – це метод написання добре структурованих програм, який дозволяє отримувати програми більш зручні для тестування, модифікації та використання. Логічна структура базується на строго доведеною теоремою про структурування.

У зв’язку з цим окремі фрагменти програми мають бути деякими логічними (керованими) структурами, які визначають порядок виконання правил обробки даних, що містяться в них. Будь-яка програма може бути побудована з цих стандартних логічних структур, кількість типів яких є невеликою. Аби не бути прив’язаними до якоїсь конкретної мови, скористаємося блок-схемами.

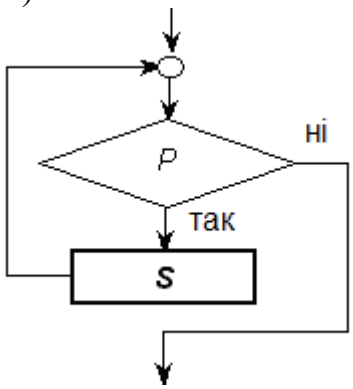
1. ***«Слідування» –*** послідовність операторів, які виконуються один за одним у порядку їх слідуваання. Ця структура є послідовністю блоків S1, S2 ..., SK, які виконуються один за одним, у порядку їх слідування згідно тексту програми.



2. ***Розгалуження*** – управляюча структура, яка залежно від виконання заданої умови (значення істиності логічного виразу P) визначає вибір для виконання одного з двох або більш заданих у цій структурі груп операторів (блоків S1 та S2).



3. ***Повторення*** **–** цикл, в якому група операторів може виконуватися багаторазово («робити, поки»), якщо виконується задана умова (логічний вираз P приймає значення «істина»).



Кожна з цих структур, у свою чергу, може мати серед своїх операторів будь-яку з припустимих структур, тобто припускати вкладення структур. Істотна особливість усіх цих структур полягає у тому, що кожна з них має лише один вхід та лише один вихід, що і забезпечує гарну структуру програми.

Кожен з вхідних у структури лінійних блоків може бути не лише окремим оператором, який задає правила обробки даних, але і будь-якою з припустимих структур, тобто припускається вкладеність структур.

У теорії програмування ще у 1965 році італійськими математиками (К. Бом и Д. Джакопіні) сформулювали теорему («теорема структурності») про те, що будь-яка програма, яка не містить «зациклень» та недосяжних операторів, може бути побудована лише з вказаних вище логічних структур.

12.3. Переваги модульності

У складних і громіздких задачах структурний підхід дає змогу розбити алгоритм на окремі частини – модулі, кожен з яких розв'язує свою самостійну підзадачу. Це дає можливість сконцентрувати зусилля на розв'язуванні кожної підзадачі, яка реалізується у вигляді окремої процедури. Для кожного такого модуля визначаються свої методи реалізації алгоритму та структура даних, якими він оперує.

Не випадково процедури та функції у мініатюрі схожі на програми. Мова Pascal спроектована таким чином, що підпрограми можуть розглядатися як незалежні об’єкти, робота яких не залежить від основної програми. Як наслідок,

наявність функцій та процедур дозволяє проектувати та конструювати програми модульним, структурованим способом.

Принцип модульності поширюється не тільки на програми, але і на дані: будь-який набір параметрів, що характеризують логічний або фізичний об'єкт, повинен бути представлений в програмі у вигляді єдиної структури даних (структурованої змінної).

Найважливіша перевага модульності, яка надається підпрограмами, – це можливість зменшити кількість повторень однієї і тієї ж послідовності операторів. У дуже великих або середніх за розміром програмах існує, принаймні, кілька процесів, виконання яких повторюється у різних частинах програми. Було б безглуздо у програмі, яка виконує якісь, наприклад, тригонометричні перетворення, обчислювати їх у десятках різних місцях. Опис функції – набагато простіше рішення.

Проте використання підпрограм викликається не тільки міркуваннями стислості. Окрім зменшення числа повторень послідовності операторів, застосування підпрограм дозволяє також у межах програми відокремити одне завдання від іншого. Таке виділення завдань цінне з кількох причин.

По-перше, це збільшує наочність та, звідсі, розуміння програми. Якщо тілом програми є набір операторів для розв’зку завдань, то читачеві важко отримати загальне уявлення про роботу програми. У протилежність цьому, якщо програма сконструйована як набір окремих підпрограм, вона стає зрозумілішою, оскільки її логічна структура більш пов’язана з фізичним сенсом.

По-друге, виділення завдань за допомогою підпрограм є також ефективним засобом розробки програм. При розробці великих або середніх за розмірами програм на програміста оідразу обрушується цілий потік деталей проектування.

Використання підпрограм робить процес програмування більш систематичним та регульованим по нарощенню складності. Завдяки використанню підпрограм програміст може розглядати програму як набір більш дрібних компонентів, які проектуються окремо.

Враховуючи складність програм, цей ефект є кориснішим, ніж може здаватись на перший погляд. Як правило, складність програм зростає у геометричній прогресії з її розміром, тому розбиття програми на менші частини дійсно зменшує зусилля, які потрібні для її розробки. Використання підпрограм дозволяє програмісту мінімізувати складність програми за допомогою застосування стратегії “розділяй та володарюй”. Починаючи з рудиментарного (тобто першооснови) опису функцій програми, можна поступово розбивати програму на окремі завдання, які, у свою чергу, можуть бути реалізовані як об’єднання функцій та процедур.

Як бачимо, використання процедур та функцій є природним доповненням до техніки програмування, яка дозволяє створювати якісні програми. Виділення завдань покращує наочність програм, проте це ще не все. Використовуючи

продумані назви для процедур та функцій, програміст може виробити більш наочний стиль написання програм.

Останнім кроком у модульній побудові алгоритмів є об'єднання окремих модулів у єдине ціле. Для цього між модулями повинні встановлюватися зв'язки для передачі інформації від одних модулів до інших: результати виконання одних модулів є вхідною інформацією для інших.

Особливо ефективна така методика під час розробки складних алгоритмічних систем колективами програмістів. У цьому випадку кожному розробнику дається своя цілісна частина алгоритму, яка реалізується ним у вигляді окремого модуля. Завершенням колективної роботи є зведення цих модулів у єдине ціле.

У програмі на прикладі 12.1 проілюстровано застосування такого підходу. Кожне завдання в SearchKey оформлене як окрема підпрограма. Кожній підпрограмі присвоєно ім’я у відповідності з навантаженням, через це основна програма SearchKey стає зрозумілою при мінімальних довідках про підпрограми. В ідеалі тіло основної програми повинне мати вигляд опису, в якому наводиться інформація про те, що програма робить, а не як робить. Деталі алгоритму реалізуються в окремих функціях та процедурах.

У розділі 2.4 ми вже розглядали порядок складання програми. Він підходить для невеликих програм, які не використовують процедури. Проте всі реальні програми використовують процедури, тому цей порядок потрібно доповнити.

12.4. Етапи розробки програмного забезпечення

Програмне забезпечення складається з декількох спільно працюючих програм (програмних модулів), які об’єднані у програмний комплекс, та документів, які необхідні для розробки, супроводу та експлуатації програмного комплексу.

У простому випадку програмний комплекс може включати тільки одну програму. Процес розробки програмного забезпечення можна розбити на етапи (фази). Розглянемо кожен етап більш докладно.

1. Перший етап – постановка завдання. Сучасні програми створюються колективами фахівців у різних областях – аналітиків, проектувальників, кодувальників, тестологів (тестувальників) та менеджерів. Спочатку замовник розповідає розробникам, що він хоче отримати (часто не розуміючи що саме). Розробники (зазвичай аналітики) прагнуть зрозуміти замовника, *аналізують та уточнюють постановку завдання*. Вони також оцінюють тривалість та вартість необхідних робіт, доцільність та план виконання проекту, чим цей проект буде краще за існуючих аналогів. Результат їх роботи – *специфікація завдання* (на мові, яка є зрозумілою розробникам), тобто короткий неформальний опис того, що має бути запрограмовано, які дані обробляє та створює програма. Специфікація завдання слугує відправною точкою для уточнення вимог до

програми, на основі яких вона проектується, документується та супроводжується у подальшому.

Інколи аналіз та уточнення завдання дозволяють створити *формальну постановку завдання*, тобто математично точний та однозначний її опис. Цей опис повинен відповідати тому, що хотів замовник, але так буває не завжди.

Робота над програмним забезпеченням починається зі складання документа, який називається "Завдання на розробку програмного забезпечення (технічне завдання)". У ньому зазвичай вказується наступне:

1. Дається коротке визначення завдання, що вирішується, назва програмного комплексу, вказується середовище програмування для його реалізації та вимоги до апаратного забезпечення.
2. Детально висловлюється постановка завдання, описується вживана математична модель для завдань обчислювального характеру, метод обробки вхідних даних для завдань необчислювального характеру, тощо.
3. Формулюються основні вимоги щодо способу взаємодії користувача з програмою (інтерфейс користувач-комп'ютер).
4. Описуються вхідні дані, вказуються межі, в яких вони можуть змінюватися, значення, які вони не можуть приймати, і т. і., а також джерело даних тобто пристрій, за допомогою якого вони повинні надходитиі у програму.
5. Описуються вихідні дані, вказується, в якому вигляді вони повинні бути представлені – в числовому, графічному або текстовому, а також вказується пристрій відображення цих даних.
6. Наводиться один або декілька прикладів роботи програмного комплексу, на яких у простих випадках проводиться його відладгодження та тестування.

1.2. Другий етап – вибір методу розв'язання. На цьому етапі створюється математична або логічна модель досліджуваного явища реального світу. Якщо програмоване завдання має обчислювальний характер, то наводиться виведення всіх формул, які використовуються, з докладними коментарями. При побудові математичних моделей далеко не завжди вдається знайти формули, які явно відображають потрібні величини через дані. У таких випадках використовуються математичні методи, які дозволяють дати відповіді того чи іншого ступеня точності. Якщо ж завдання не має обчислювальний характер, то наводиться словесний опис логічної моделі, наприклад, у вигляді плану дій.

1.3. Третій етап – розробка алгоритму вирішення задачі (проектування). На етапі *проектування* формується загальна структура програмного комплексу. Відповідно до технології спадаючого структурного програмування, яке розглядається далі, програмний комплекс розбивається на невеликі частини – програмні модулі (блоки). Для кожного програмного модуля формулюються вимоги по функціях, які реалізовуються, та розробляється алгоритм, який реалізує ці функції.

Визначається схема взаємодії програмних модулів, тобто схема потоків даних програмного комплексу.

Результатом виконання цього етапу може бути блок-схема (або псевдо-код) алгоритму розв'язку поставленої задачі.

1.4. Четвертий етап – кодування алгоритму. Етап кодування (програмування) алгоритмів полягає у перетворенні алгоритмів, які розроблені для кожного програмного модуля, у програми на конкретній мові програмування. Результатом виконання цього етапу є файли з початковими текстами програм. Ці файли за своєю природою є текстовими, тільки вони містять тексти, які написані на мові програмування (у нашому випадку це тексти, які написані на мові Pascal).

1.5. П'ятий етап – трансляція та компіляція програми. Після того, як закінчено кодування (написання програми на мові програмування) і початковий текст програми введений у пам'ять комп'ютера, виконується трансляція та компіляція програми.

Спочатку спеціальна програма (транслятор) перевіряє початковий текст програми на наявність у ній так званих синтаксичних помилок, тобто відповідність написаних операторів правилам, які передбачені у даній мові програмування. Причому трансляція проводиться до першої помилки, що зустрілася. При виявленні помилки процес трансляції припиняється, транслятор видає повідомлення про характер та місце знаходження помилки. Необхідно виправити помилку та повторити трансляцію. Так продовжується доти, доки всі помилки трансляції не будуть усунені.

Потім відбувається зборка програми (компіляція), тобто до програми підключаються всі замовлені нею бібліотеки, процедури, функції і т.і. Якщо будь-який компонент не виявлений, видається відповідне повідомлення і процес припиняється. Необхідно переконатися у наявності не знайденого компоненту, у правильності вказаного його імені або шляхів доступу до нього. Потім знову повторити компіляцію. В разі успішного завершення процесу утворюється файл програми (файл з розширенням EXE) до виконання. За допомогою цього файлу програму запускають на виконання.

1.6. Шостий етап – тестування та відлагодження програми. Відлагодження та тестування (англ. test – випробування) – це два чітко помітних та несхожих один на одний етапи:

- при **відлагодженні відбувається локалізація та усунення синтаксичних помилок, а також явних помилок кодування**;
- у процесі ж **тестування перевіряється працездатність програми, яка не містить явних помилок**.

Тестування встановлює факт наявності помилок, а відлагодження з'ясовує їх причину.

Розрізняється два види тестування: автономне та комплексне. При автономному тестуванні тестуються окремі програмні модулі, з яких складається програмний комплекс. Комплексне тестування полягає у перевірці всього програмного комплексу.

Для тестування підбираються такі початкові дані, для яких результат виконання програми вже заздалегідь відомий.

Після того, як при тестуванні виявлена помилка, починається процес відлагодження протестованого програмного модуля або програмного комплексу. Тестування та відлагодження чергуються і завершуються після того, як буде ухвалено рішення про відсутність у програмному комплексі помилок.

Відлагодження у мові Pascal здійснюється з використанням спеціальних програмних засобів (відладчиками), про що ми говорили раніше. Ці засоби дозволяють **досліджувати внутрішню поведінку програми**.

Програма-відладчик зазвичай забезпечує такі можливості:

- **покрокове виконання програми** із зупинкою після кожної команди (оператори F7 або F8);
- **ппереглядпоточного значення будь-якої змінної або знаходження значення будь-якого виразу**, зокрема, з використанням стандартних функцій; при необхідності можна встановити нове значення змінної;
- **встановлення у програмі "Контрольних точок"**, тобто точок, в яких програма тимчасово призупиняє своє виконання, так що можна оцінити проміжні результати.

При відлагодженні програм **важливо пам'ятати наступне:**

- на початку процесу відлагодження треба використовувати **прості тестові дані**;
- труднощі, які виникають, слід чітко **розділяти та усувати строго по черзі**;
- **не треба вважати причиною помилок машину**, оскільки сучасні машини та транслятори володіють надзвичайно високою надійністю.

Зазвичай на етапі трансляції виявляються синтаксичні помилки. Багато ж інших помилок транслятору виявити неможливо, оскільки **транслятору невідомі задуми програміста.**

Виявлення транслятором синтаксичних помилок являє собою самий важливий і необхідний етап налагодження програми.

Якщо під синтаксичної помилкою розуміти «будь-яке порушення вимог мови програмування», то слід визнати, що багато помилок залишаються невиявленими. Прикладами синтаксичних помилок можна назвати:

- пропуск необхідного знака пунктуації;
- неузгодженість дужок;
- пропуск потрібних дужок;
- неправильне формування оператора;
- неправильне використання арифметичних операторів.

Відсутність повідомлень комп'ютера про синтаксичні помилки є **необхідною, але не достатньою** умовою, щоб вважати програму правильною. Приклади синтаксичних помилок:

- пропуск знаку пунктуації;
- неузгодженість дужок;

– неправильне формування оператора;
– неправильне утворення імен змінних;
– неправильне написання службових слів;
– відсутність умов закінчення циклу;
– відсутність опису масиву, тощо.

Але існує безліч помилок, які транслятор виявити не в змозі, якщо оператори, які використовуються у програмі, сформовані правильно. Наведемо деякі приклади таких помилок, які виявляються тільки за допомогою тестування.

Помилки загального характеру. Після того як знайдений відповідний алгоритм вирішення задачі, на етапі програмування також можуть з'явитися помилки, незалежно від вибраної мови:

– помилки через недостатнє знання або розуміння програмістом мови програмування або самої машини;
– помилки, допущені при програмуванні алгоритму, коли команди, що використовуються у програмі, не забезпечують послідовності подій, які встановлені алгоритмом.

Помилки фізичного характеру. Можна назвати кілька типів помилок, що викликаються невірними діями програміста:

1. Пропуск деяких операторів.
2. Відсутність необхідних даних.
3. Непередбачені дані.
4. Невірний формат даних.

Логічні помилки:

– невірна вказівка гілки алгоритму після перевірки деякої умови;
– неповний облік можливих умов;
– пропуск у програмі одного або більше блоків алгоритму.

Помилки у циклах:

– неправильна вказівка початку циклу;
– неправильна вказівка умов закінчення циклу;
– неправильна вказівка кількості повторів циклу;
– нескінченний цикл.

Помилки введення-виведення; помилки при роботі з даними:

– неправильне визначення типу даних;
– організація зчитування меншого або більшого об’єму даних, ніж потрібний;
– неправильне редагування даних.

Помилки у використанні змінних:

– використання змінних без вказівки їх початкових значень;
– помилкова вказівка однієї змінної замість іншої.

Помилки при роботі з масивами:

– масиви заздалегідь не обнулені;

– масиви неправильно описані;
– індекси слідують у неправильному порядку.

Помилки арифметичних операцій:

– невірна вказівка типу змінної (наприклад, цілочисельного замість дійсного);
– невірне визначення порядку дій;
– ділення на нуль;
– обрахунок квадратного кореня з від'ємного числа;
– втрата значущих розрядів числа.

1.7. Сьомий етап – супровід програм. Супровід програм – це роботи, які пов'язані з обслуговуванням програм у процесі їх експлуатації.

Багаторазове використання розробленої програми для розв'язку різних завдань заданого класу вимагає проведення додаткових робіт, які пов'язані з **доопрацюваннями програми для вирішення конкретних** завдань, проведення додаткових тестових обчислень, тощо.

Програма, яка призначена для тривалої експлуатації, повинна мати відповідну **документацію** та інструкцію для її використання.

12.5. Технологія спадаючого структурного програмування

Якщо технологію розробки алгоритмів «згори-донизу» сумістити з використанням тільки структурних схем, то вийде нова технологія, яка називається **структурним програмуванням згори-донизу (*Низхідне програмування*)**, ідея якого полягає у тому, що структура програми повинна відображати структуру вирішення задачі, щоб алгоритм розв'язку був зрозумілий з початкового тексту. Для цього треба мати засоби для створення програми не тільки за допомогою трьох простих операторів, але й за допомогою засобів, які точніше відображають конкретну структуру алгоритму. З цією метою у програмуванні введено поняття підпрограми (процедури та функції) – набору операторів, які виконують потрібну дію та не залежать від інших частин початкового коду. Програма розбивається на безліч дрібних підпрограм (що займають до 50 операторів – критичний поріг для швидкого розуміння мети підпрограми), кожна з яких виконує одну з дій, що передбачені початковим завданням. Комбінуючи ці підпрограми, вдається формувати підсумковий алгоритм вже не з простих операторів, а із закінчених блоків коду, які мають певне семантичне навантаження, причому звертатися до таких блоків можна за назвами. Виходить, що підпрограми – це нові оператори або операції мови, які визначаються програмістом.

На початку 70-х років корпорація IBM повідомила про застосування у розробці програмного забезпечення «Вдосконалених методів програмування», які забезпечують перехід до промислових методів розробки програмного забезпечення. Одним із основних компонентів «Вдосконалених методів програмування» була взята технологія спадаючого структурного

програмування, яка підтримується сучасними мовами програмування. Пізніше на базі технології спадаючого структурного програмування були створені технології об'єктно-орієнтованого та подієво-керованого (візуального) програмування.

Також у 70-х роках з'явилася технологія ***висхідного програмування*** – програмування "знизу догори". Сутність висхідної технології полягає у тому, що спочатку розв'язуються більш прості та зрозумілі завдання (реалізація алгоритму сортування, обробка рядка, введення даних) і тільки потім приступають до побудови великої програми з готових дрібних частин. Тобто, у цьому випадку розв'язок (програма) як би "складався з окремих цеглин", з відомих підзадач. Можна сказати, що суттість технології висхідного програмування – "ВІД ЧАСТКОВОГО ДО ЗАГАЛЬНОГО".

У "чистому вигляді" технологія висхідного програмування зустрічалася рідко, Тому, в основному, програми створювалися і створюються з використанням змішаних технологій, тобто на різних ділянках можуть використовуватися як висхідне, так і спадаюче структурне програмування.

Технологія спадаючого структурного програмування базується на методі програмування "**згори-донизу"**, який часто називають методом ***покрокової деталізації***, і включає, в основному, такі три складові:

- спадаюча розробка;
- структурне кодування (програмування);
- тестування.

Більшість фахівців в області програмування дотримуються тієї точки зору, що саме метод покрокової деталізації створює передумови до розв'язку складних проблем. Основою цього методу є ідея поступової ***декомпозиції*** початкового завдання на низку підзадач. Декомпозиція може бути багаторівневою, кожна з отриманих підзадач також може виявитися досить великою, і тоді до неї теж можна застосувати операцію декомпозиції. Послідовне застосування операції декомпозиції до підзадач різного рівня називається ***низхідним проектуванням***.

Спочатку формулюється загальна модель розв'язку, окремі деталі якої на першому етапі можуть бути досить розпливчастими, як, наприклад, вигляд ділянки землі з великої висоти, коли дрібні подробиці є невиразними. По мірі розробки програми, розбиваючи найбільш незрозумілі частини алгоритму, та досягаючи все більш точних та деталізованих формулювань, ми отримуємо докладніше рішення, як би опускаємося з великої висоти, і починаємо при цьому розрізняти дрібніші деталі, звідси і назва методу – "згори-донизу". Розв'язок окремого фрагмента складного завдання може бути самостійним програмним блоком, який називається підпрограмою. Такий процес деталізації продовжується доти, доки не стануть зрозумілі всі деталі вирішення задачі. У цьому випадку програму вирішення складної задачі можна представити у вигляді ієрархічної сукупності порівняно самостійних фрагментів – підпрограм (модулів).

12.5.1. Спадаюча розробка

Спадаюча розробка – це підхід до розробки програмного комплексу, коли він розбивається на програмні модулі (програми, підзадачі), створюючи багаторівневу структуру. Тобто, сутність цього методу полягає у тому, що, приступаючи до розв'язку задачі, її спочатку розглядають як деяку єдину дію. Потім виділяють у завданні невелику кількість підзадач та встановлюють характер їх взаємодії. Далі кожна підзадача розбивається знов на декілька підзадач, і так доти, доки виділені підзадачі не зможуть бути розв'язані безпосередньо. При розбитті на підзадачі зазвичай керуються правилом «7±2», розділяючи кожне завдання приблизно на 5-9 частин. Вважається, що саме такою кількістю елементів людина може вільно оперувати, не втрачаючи контролю над їх взаємодією.

Кожен програмний модуль є короткою програмою, яка розв'язує окреме завдання (підзадачу). У спроектованій програмі намічається відповідна кількість блоків (модулів, частин програми), кожен з яких призначений для розв'язку однієї з виділених підзадач. Визначається призначення кожної з них, порядок виконання цих блоків та їх зв'язок між собою за даними, які обробляються. На цьому етапі важливо визначити лише функціональне призначення кожного блоку – що він має робити. Питання про те, як будуть реалізовуватись покладені на блок функції, поки що можна не розглядати. Тому вони тимчасово замінюються «**заглушками**».

Таким чином, у будь-який момент розробки програмного комплексу є його діючий варіант (**прототип**), який визначається на певному кроці деталізації. Тестування та відлагодження окремих програмних модулів та програмного комплексу у цілому ведеться по ходу його проектування.

У мовах програмування, які орієнтовані на технологію спадаючого програмування, одним із засобів реалізації модульної структури є процедури та функції.

Спільну схему будь-якої програми можна представити у вигляді трьох послідовних блоків, що виконуються:

Блок1. Завдання вхідних даних.

Блок 2. Розв'язок поставленої задачі.

Блок 3. Видача результатів.

Виділення першого блоку корисне, щоб на самому початку проектування програми поміркувати над тим, що є її вихідними даними, та в якому вигляді вони мають бути представлені у зовнішньому середовищі та у програмі.

При проектуванні програми, особливо її блоку 1, важливо замислитись над тим, як повинна поводитися програма у разі завдання помилкових або некоректних вхідних даних. Якщо такі помилки можуть призвести до значного непродуктивного витрачення машинного часу або до серйозних наслідків через видачу неправильних результатів, то слід передбачити програмний контроль вхідних даних та відповідну реакцію на помилки при їх завданні – припинення виконання програми, друк діагностичних повідомлень і таке інше Як мінімум,

слід передбачити роздруківку вхідних даних, щоб знати, при яких вхідних даних насправді виконувалася програма.

Уважно треба відноситися і до форми подання остаточних результатів, і перш за все треба прагнути до зручності їх подальшого використання людиною.

Якщо у деяких спочатку виділених підзадачах виникають певні труднощі, так що алгоритми їх розв'язку ще не є очевидними, то до кожної з них можна застосувати аналогічну процедуру – це буде другий крок деталізації. Цей процес продовжується доти, доки кожен з виділених блоків програми не виявиться настільки простим, що його реалізація вже не викликає труднощів.

Переходити до наступного рівня деталізації має сенс лише за наявності достатньої впевненості у правильності та ефективності алгоритму на даному рівні його деталізації.

Покрокова деталізація з використанням невеликих програмних модулів має певні переваги. По-перше, тому, що на кожному черговому кроці деталізації доводиться приймати порівняно невелику кількість рішень і тому легко перевірити їх правильність. До того ж, відсутність великої деталізації дозволяє розглянути та оцінити декілька можливих варіантів, обравши найкращий з них.

По-друге, оскільки деталізація будь-якого блоку програми проводиться локально, незалежно від інших блоків, то і перевірка правильності виконаної деталізації має локальний характер. Отже, програміст може не думати про весь алгоритм у цілому, а зосередити всю свою увагу на черговому блоці. З невеликими модулями зручніше працювати, легко модифікувати. Невеликі модулі легше та ефективніше тестуються.

Проектування програми, взагалі кажучи, не пов'язано з тією мовою, на якій буде записуватись остаточний текст програми, – для цього досить мати лише уявлення про міру необхідної деталізації, яка визначається цією мовою. Ця обставина і дозволяє здійснювати розподіл праці з розробки програми та її кодування. Розподіл треба здійснювати навіть у тому випадку, коли ці два етапи робіт виконуються одним і тим самим програмістом. По ходу виконання розробки програми треба весь час якось фіксувати результати виконаної роботи. Для досягнення вказаної мети раніше часто використовувались **блок-схеми**.

Основна перевага блок-схем полягає у тому, що вони не вимагають якоїсь певної деталізації алгоритму і тому можуть використовуватися на будь-яких етапах розробки програми.

Недолік блок-схем полягає у тому, що при чималій мірі деталізації вони стають громіздкими та втрачають свою основну перевагу – наочність структури алгоритму. Крім того, вони вимагають багато часу на їх викреслювання, а також незручні для публікацій. Тому, найчастіше багато програмістів для розробки додаткового документу до програмного забезпечення (ПЗ) у вигляді блок-схем удаються до різних прийомів: креслять блок-схеми вже після розробки ПЗ. Але найчастіше таке «креслення» виконується спеціальною програмою, вхідними даними для якої є текст програми (або фрагменти програми) на мові програмування (так званий «**реінжинірінг**» у сучасному розумінні).

Тому зовсім не обов'язково користуватися блок-схемами при розробці програм за допомогою покрокової деталізації. Навпаки, уніфікація стилю та достатня наочність структурних управляючих операторів, невелика кількість підзадач, які виділяються на кожному кроці, роблять блок-схеми непотрібними. Вони вже не дають більшої наочності, ніж це забезпечує добре структурована програма.

Останнім часом замість блок-схем все частіше використовується спеціальна мова – **псевдокод**, який близький до мови програмування, наприклад до мови Pascal. У ньому використовуються ті ж самі управляючі структури та зарезервовані слова, проте запис правил обробки даних та умов не формалізований, тому ці правила та умови можуть формулюватися будь-яким зручним для людини способом. Наприклад, фрагмент програми на псевдокоді:

if найбільша компонента вектора негативна **then**
усі компоненти збільшити на 2
else
всі компоненти вектора зменшити на 0.5
end;
while необхідна точність не досягнута **do**
обчислити та врахувати черговий доданок до ряду
за допомогою якого обчислюється sin(x);
end;

У зв'язку з цим можна не вводити до вжитку якоїсь спеціальної мови проектування програм, а використовувати для цієї мети сам Pascal, задаючи вміст ще недостатньо деталізованих блоків програми у вигляді коментарів Pascal. Цей прийом дозволяє в результаті отримати добре прокоментовану програму.

12.5.2. Кодування, тестування та відлагодження згори-донизу

Використання невеликих програмних модулів має певні переваги: з такими модулями зручніше працювати, вони дозволяють розробляти програмні комплекси, які легко модифікувати; невеликі модулі легше та ефективніше тестуються.

Розглянемо процеси **кодування та тестування** на прикладі деякого програмного комплексу, який можна представити у вигляді структурної діаграми, яка зображена на Рис. 12.1.

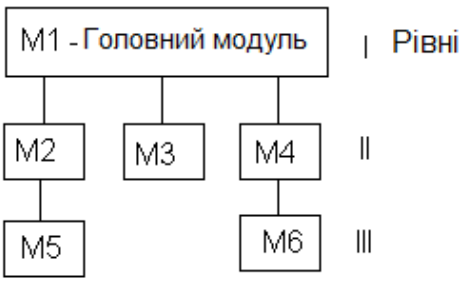


Рис. 12.1. Структурна діаграма програмного комплексу

Як бачимо з рис. 12.1, програмний комплекс містить модулі трьох рівнів. Програмний модуль першого рівня М1 (керівник) викликає три програмні модулі другого рівня М2, М3, М4. Програмні модулі другого рівня М2 та М4 викликають програмні модулі третього рівня М5 та М6 відповідно.

Проектування та кодування програмного комплексу починається з керуючого програмного модуля М1. Для його тестування та відлагодження необхідно мати програмні модулі другого рівня, але оскільки вони ще не спроектовані, замість них використовуються імітатори цих програмних модулів – заглушки. Оскільки призначення заглушок полягає тільки у тому, щоб програмний модуль верхнього рівня був виконаний, вони можуть бути достатньо простими.

Після того, як головний програмний модуль М1 протестований, проектується, кодується та включається замість «заглушки М2» програмний модуль М2. Програмні модулі М3 та М4 як і раніше залишаються заміненими заглушками.

Аналогічним чином поступають при підключенні до програмного комплексу модулів М3 та М4.

Для підключення до програмного комплексу модулів М2 та М4, необхідно програмні модулі М5 та М6, що викликаються ними, також замінити заглушками.

Завершивши тестування та відлагодження модулів першого та другого рівнів, приступають до проектування та відлагодження модулів третього рівня.

Крім тестування та відлагодження кожного програмного модуля одночасно ведеться тестування та відлагодження програмного комплексу у цілому. Внаслідок цього, як вже наголошувалося раніше, після підключення кожного програмного модуля, отримаємо працюючий варіант програмного комплексу.

Розглянемо ці важливі етапи при розробці ПЗ (тестування та відлагодження) детальніше.

Тестування. Спершу наведемо три аксіоми одного з найперших радянських програмістів з декілька незвичним прізвищем Шура-Бура (Інститут Прикладної Математики), які схожі з гаслом Дж. Мартіна, що наведено у першому розділі посібника.

Аксіома 1. У кожній програмі є помилка.

Аксіома 2. Якщо в програмі помилок немає, значить, у вихідному алгоритмі є помилка.

Аксіома 3. Якщо ані у програмі, ані в алгоритмі помилок немає, то така програма нікому не потрібна.

При всій жартівливості цих аксіом в них відбита сувора правда життя. На жаль, усі програмісти припускають помилок. Чим більше програма, тим вище ймовірність виникнення у ній помилок, багато з яких до певного часу залишаються непоміченими та потрапляють у кінцевий програмний продукт, який вже випущений на ринок. З цією сумною істиною важко змиритися, тому створення надійних, вільних від помилок програм має бути завданням номер один для кожного програміста, який серйозно ставиться до своєї справи.

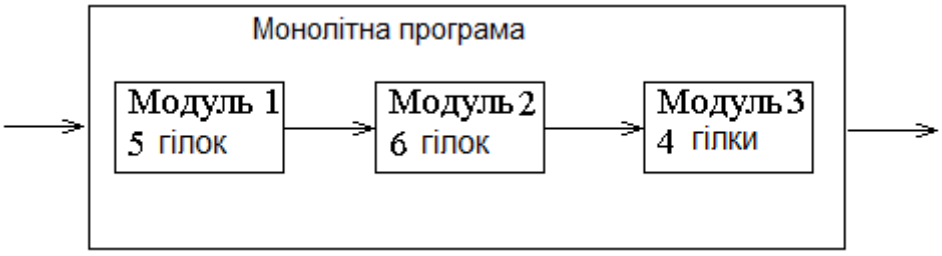
Зазвичай найбільші витрати у багатьох роботах, які пов'язані з програмуванням, доводяться на тестування програм та виправлення помилок. Той, хто розв'яже проблему створення добротних, надійних та безвідмовних програм за короткий термін та з низькими витратами, зробить революцію в усій індустрії програмних продуктів.

Саме тому тестування, а не відлагодження є центральним моментом завершальної стадії розробки програми. *Мета тестування* – переконатися у тому, що програма функціонує як слід, що вона відповідає функціональним вимогам, і що вона розв'язує реальну задачу. *Мета відлагодження* – усунути помилки у програмі. Тут не обов'язково досягати такого ж самого результату, що і при тестуванні. При відлагодженні на будь-якій стадії деякі частини програми можуть залишитися неперевіреними, оскільки розглядаються лише ті помилки, що виявляються, а інші так і залишаються не виявленими.

В ідеальному випадку програму треба протестувати для всіх можливих комбінацій вхідних даних, а вихідні дані для кожної комбінації повинні порівнюватися із заздалегідь розрахованими правильними результатами. Проте майже для всіх програм, навіть для найпростіших, існує нескінчена кількість можливих комбінацій вхідних даних або, щонайменше, ця кількість є настільки великою, що її можна вважати бескінечною.

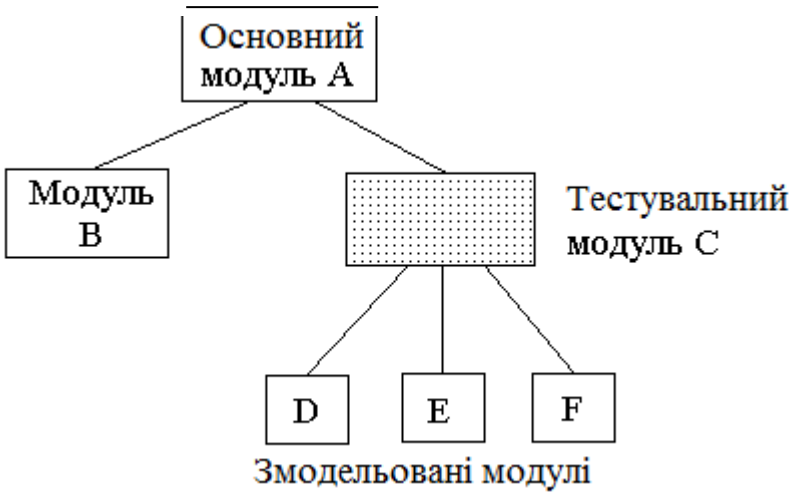
На щастя, внутрішні операції обчислювальної системи можна розглядати узагальнено: якщо вони працюють з одним набором операндів, то вони працюватимуть, в певних межах, для будь-якого аналогічного набору операндів. Це твердження не вірне для програм або їх частин, де мають місце умовні переходи. Тестування тому має бути зосереджене на перевірці невеликих окремих фрагментів програми, які володіють вказаними властивостями. Тоді можна буде зробити висновок про те, що якщо ці невеликі фрагменти правильно працюють для одного набору вхідних даних, то вони правильно працюватимуть і для всіх інших наборів. Головне полягає у тому, щоб визначити, чи достатньо простим є обраний фрагмент програми, щоб його можна було перевірити описаним способом.

Для вичерпного тестування або перевірки програми потрібна перевірка кожної із її гілок. Кількість гілок, що перевіряються, може бути зменшена до прийнятного значення за допомогою розбиття програм та окремої перевірки кожного її фрагменту. Наприклад, розглянемо монолітну програму, яку потім розбили на три модулі.



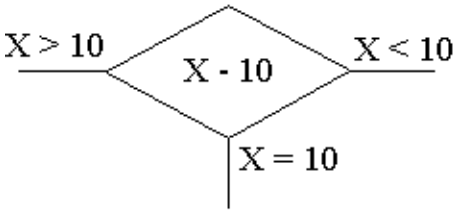
Якщо програма тестується як єдине ціле, то кількість різних гілок дорівнює
5*6*4 = 120, але якщо програма розбивається на модулі, то це число дорівнює 5+6+4 = 15 плюс ще 1, коли модулі збираються разом і кожен розглядається як окремий оператор програми, тобто всього 16 варіантів.

Такий метод добре узгоджується з принципами структурного програмування, які вимагають розбиття програми на логічні та функціональні модулі. Кожен з них перевіряється індивідуально, але не ізольовано. Тестування, згідно зі спільним принципом, проводиться згори-донизу. На кожному етапі функції модулів нижчого рівня моделюються при тестуванні даного модуля по всіх його логічних гілках. Середовище тестування можна представити таким чином:



Тестування такої модульної програми починається з головного модуля в середовищі, що утворюється моделями останніх модулів. Після перевірки модуль може бути приєднаний до програми і можна буде приступити до дослідження підлеглих йому модулів. Тому спільна функція всієї програми видна із самого початку і всі фундаментальні проблеми з'ясовуються на початкових етапах тестування.

Потрібно добре поміркувати про дані для тестування, які забезпечують перевірку кожного модуля по всіх його логічних гілках. Окрім даних, що забезпечують виконання операторів програми в необхідній послідовності, у вхідному наборі повинні міститися дані, які перевіряють граничні умови:



Наприклад, перехід, який здійснюється за умовою, що перевіряється, $X \geq 10$, повинен перевірятися для значень більших, менших та рівних 10. Крім того, повинно бути встановлено, що виконання йшло по правильній гілці. Це досягається

введенням відповідних операторів трасування (або режиму трасування у Відладчику), які вказували б на те, що виконання пройшло по заданій гілці. Після перевірки ці оператори можуть бути вилучені.

Процес тестування засвідчує якість програми, тому він має бути документований. Користувач повинен знати, як і за яких умов програма тестувалася, які були вхідні дані і результати, з тим, щоб тест можна було повторити. У документації повинні міститися детальний опис тестових процедур, функції кожного тестового набору даних та результати, які мають бути виведені правильно працюючою програмою.

Відлагодження. Помилки у програмі або у модулі можуть виявлятися по-різному. Синтаксичні помилки та помилки, які викликані внутрішніми невідповідностями, можуть бути виявлені компілятором. Існують ще і логічні помилки двох видів:

1) помилки, які не викликають припинення виконання програми, але призводять до отримання невірних результатів;
2) помилки, які викликають виконання неприпустимих операцій (ділення на нуль, звернення до неіснуючого елементу масиву) і призводять до передчасного переривання виконання програми.

У деяких відладчиках є засоби «відновлення», які дозволяють продовжувати роботу, якщо виникла непередбачена ситуація, наприклад, ділення на нуль.

Ключем до ефективного відлагодження є систематичне та детальне дослідження симптомів помилок з тим, щоб визначити дійсні причини помилок. Найкориснішим засобом відлагодження є виборчий при виведенні («знімок»), тобто висвітлення обраних змінних у ключових точках програми. Знайдена помилка має бути виправлена. Неуважне виправлення може стати джерелом нових помилок.

Програма може неправильно працювати не внаслідок власних, а внаслідок зовнішніх помилок. Прикладами таких помилок є помилки у компіляторі або у системному програмному забезпеченні, збої у технічному забезпеченні. Але, на щастя, вони у наш час дуже рідкі! Але все ж таки у складних проектах для захисту застосовуються контрольні точки та перезапуски, які обмежують втрати через технічні помилки.

12.5.3. Оформлення програм

Деякі прийоми при оформленні програми вже викладалися у розділі 2.8 «Чи є Ви хорошим програмістом», але враховуючи важливість цього засобу структурного програмування розглянемо їх детальніше. Одним із засобів, який дозволяє легко виявляти вкладеність структур, що управляють одна в одній і тим самим полегшити орієнтацію у програмі, є розташування тексту окремих операторів по аркушу паперу. Службові слова, якими починається і закінчується той чи інший оператор, слід записувати на одній вертикалі, а всі вкладені в нього оператори записують з деяким відступом праворуч. З цією

метою інколи буває зручно об’єднувати у складений оператор і таку послідовність операторів, де за синтаксисом можна обійтися і без складеного оператора.

Найбільш ефективнішим засобом полегшення розуміння програми є її коментування. Часто програміст має намір забезпечити свою програму коментарями пізніше, на останньому етапі її виготовлення. Проте ці наміри зазвичай не реалізуються. Після закінчення роботи зі складання і відлагодження програми робота по коментуванню видається йому вже зайвою. Тим більше, що спеціально для цієї роботи часу зазвичай вже не знаходиться. Буває і так, що через деякий час автор програми і сам вже забув багато її деталей.

Наприклад, той же Г. Майерс у своїй роботі пише: «Некоментована програма – це, ймовірно, найгірша помилка, яку може зробити програміст, а також свідоцтво дилетантського підходу (нехай навіть програміст має тривалий досвід роботи); більш того, це вагома причина для звільнення програміста». Коментувати програми треба по ходу їх розробки і написання. Для гарного коментування потрібні певні практичні навички, які виробляються лише з часом. Зайві та невдало розташовані коментарі, лише захаращують текст програми і мало сприяють її розумінню.

Можна вважати, що програма прокоментована вдало, якщо при першому знайомстві з нею можна зрозуміти структуру програми, її сутність та логіку роботи, лише переглядаючи управляючі структури, що містяться у ній, та переглядаючи коментарі, не аналізуючи детально операторів, які задають правила обробки даних. Вивчення таких операторів потрібне лише для уточнення деталей, що необхідно, наприклад, для ретельної перевірки програми.

Заголовки служать для виділення та пояснення призначення основних блоків програми. Коментарі цього типу повинні відображати результати виконаної раніше роботи при покроковій деталізації алгоритму. Коментарі по рядках відносяться до досить дрібних фрагментів програми (операторів, описів).

Зазвичай кожна програма забезпечується ще коментарями, які розміщуються на початку тексту програми. За їх допомогою задається спільна інформація про дану програму. Ця інформація зазвичай містить в собі такі пункти:

1) призначення програми;
2) відомості про автора програми;
3) відомості про організацію, в якій виготовлена програма;
4) дату написання програми;
5) метод розв’язку задачі (якщо такий є);
6) вказівки по введенню і виведенню даних.

Можуть міститися й інші пункти:

1) час, який потрібний на виконання програми;
2) необхідний об’єм пам’яті;
3) спеціальні вказівки користувачеві (у разі потреби).

12.6. Створюємо першу “велику” програму

Тепер, коли ви познайомилися з однією з технологій програмування – технологією спадаючого структурного програмування (по методу “згори-донизу”), ми разом з вами напишемо першу реальну програму. Достатньо велику для початківців. По ходу постановки завдання ви також дізнаєтеся про найпростіші методи **сортування** та **пошуку** у масивах, які широко росповсюджені у прикладних програмах.

У розділі 3.4 ми вже розбирали порядок складання невеликих програм, які не використовують процедури. Проте всі реальні програми використовують процедури (функції), які розбивають ваше завдання на прості підзадачі.

Постановка завдання. Скласти програму для пошуку ключового елементу, який обирається випадковим чином із заданого діапазону чисел, у масиві чисел, який заданий також випадковим чином з того ж діапазону чисел. Для пошуку ключового елементу застосуємо два методи пошуку – лінійний (пошук у невідсортованому і відсортованому масиві) та діхотомічний або двійковий (у відсортованому масиві). Масиви та результати пошуку (номер елементу у масиві та кількість операцій порівняння) видати на екран після кожного пошуку.

Сценарій нашого завдання буде таким:

1) очистити екран;
2) заповнення масиву випадковими числами із заданого діапазону;
3) отримання випадкового ключа (числа) для пошуку;
4) виведення елементів масиву на екран;
5) лінійний пошук ключового елементу у невідсортованому масиві;
6) виведення результатів пошуку на екран;
7) сортування масиву одним з відомих простих методів;
8) лінійний пошук ключового елементу у відсортованому масиві;
9) виведення результатів пошуку на екран;
10) діхотомічний (бінарний) пошук ключового елементу;
11) виведення результатів пошуку на екран;
12) перегляд та порівняння результатів пошуку.

12.6.1. Програмування за методом “згори-донизу”

Перше завдання, яке стоїть перед програмістом, – це розбити програму на декілька частин, виходячи із змісту завдання. Зазвичай для правильного розбиття потрібний досвід. У нашому випадку розбиття на процедури та функції напрошується само собою, виходячи із послідовності дій сценарію постановки завдання:

1) заповнення масиву випадковими числами із заданого діапазону;
2) отримання випадкового ключа (числа) для пошуку;

3) виведення елементів масиву на екран;
4) лінійний пошук ключового елементу у невідсортованому масиві;
5) сортування масиву одним із відомих простих методів;
6) лінійний пошук ключового елементу у відсортованому масиві;
7) діхотомічний (бінарний) пошук ключового елементу;
8) виведення результатів пошуку на екран.

Тепер, коли ми розбили програму на частини, потрібно вирішити, яку з частин зробити процедурою, а яку можна описати просто групою операторів. Процедура вигідна тоді, коли частина або достатньо складна (діхотомічний пошук), або зустрічається більше одного разу (виведення елементів масиву на екран).

Отже, у нас буде три процедури та три функції. Призначимо їм імена:

- заповнення масиву випадковими числами - procedure InitArray
- генерація випадкового ключа пошуку - function InitKey;
- виведення елементів масиву - procedure ShowArray;
- сортування методом бульбашки - procedure SortBubble;
- лінійний пошук ключового елементу - function SearchLine;
- бінарний пошук ключового елементу - function SearchBin;
- виведення результатів пошуку на екран - procedure Main головна - процедура.

Тепер для кожної з процедур необхідно вирішити, чи слід її розбивати на частини, причому міркувати треба так само, як при розбитті на частини всієї програми.

Всі процедури та функції розбивати не будемо, окрім ShowArray. Для цього подивимося, з чого складається процедура виведення елементів масиву на екран. Вона у нас складається з власно виведення елементів масиву та внутрішньої функції обчислення ширини колонки максимального елементу масиву. Це робиться для того, щоб у рядках розміщувались елементи масиву, які розділені пробілом, та частина цифр останнього елементу у рядку не переходила на інший рядок. Назвемо цю функцію – WidthCol.

Ми на прикладі показали, як треба дробити будь-яке завдання на більш дрібні частини доти, доки не стане зрозумілим, як будь-яку з цих дрібних частин запрограмувати на мові Pascal. Це, як ми вже говорили, і називається **програмуванням за методом "згори донизу"**. Можна програмувати і методом "**знизу догори**", коли програміст спочатку складає список найдрібніших частин, з яких складається завдання, а потім збирає з них крупніші частини.

Перш ніж наповнювати Pascal-вмістом наші процедури, треба зробити "скелет" програми, тобто "порожню" програму, після запуску якої всі функції та процедури нічого не виконують, а тільки повідомлятимуть про своє існування. Тепер після цього можна поступово наповнювати порожні процедури реальним змістом.

Спочатку для простоти відлагодимо "порожню" програму, і якщо вона працює правильно, тоді будемо поповнювати її по черзі реальними

процедурами. Ось “порожня” програма з порожніми процедурами-заглушками та змінними, яким присвоєні більш-менш усвідомлені ідентифікатори:

```
USES    CRT;
CONST maxN = 40;  RendN = 100;
TYPE        Int = integer;
        Indx = 1 .. maxN;
        MasA = array [Indx] of Int;
VAR Key, Ind, Kol:    Integer;
        MasB: MasA;
{Заповнення масиву випадковими числами}
procedure InitArray(var A : MASA; n : Indx);
   var         i: Indx;
   begin
        WriteLn(‘Заповнюємо масив випадковими числами’);
   end;
{Отримання випадкового ключа пошуку}
function InitKey: integer;
   begin
        WriteLn(‘Отримали випадковий ключ для пошуку’);
        InitKey := 1;
   end;
{Виведення елементів масиву}
procedure  ShowArray(var A : MasA; n : Indx);
   var         i: Indx;  j, w: integer;
   {Обчислення ширини (w) колонки для роздруківки елементів масиву}
   function WidthCol: integer;
        var    r, w: integer;
        begin
            WriteLn(‘Обчислили ширину (w) колонки ‘);
             WidthCol := 3;
        end;
   begin
        WriteLn(‘Виведення елементів масиву’);
   end;
{Сортування методом бульбашки}
procedure SortBubble (var A: MasA; n: Indx);
var    i, k: Indx;
        temp: Int;
begin
        WriteLn(‘Сортування масиву методом бульбашки’);
   end;
{Лінійний пошук ключового елементу}
function SearchLine ( var A : MasA; n : Indx; key : Int) : integer;
begin
```

```
        WriteLn('Лінійний пошук ключового елементу');
        SearchLine := 10;
end;
{Бінарний пошук ключового елементу}
function SearchBin ( var A : MasA; n : Indx; key : Int) : integer;
var    i, left, right : integer;
begin
        WriteLn('Бінарний пошук ключового елементу');
        SearchBin := 10;
end;
BEGIN
        ClrScr;                {Очищення екрану, необхідний модуль CRT}
        Randomize; {Щоб випадкові числа були різними при зверненні до
Random()}
        InitArray(MASB, MAXN);  {Заповнюємо масив випадковими числами}
        Key := InitKey; {Отримання випадкового ключа пошуку}
        WriteLn('Кількість елементів масиву = ', MAXN);
        ShowArray(MASB, MAXN);   {Виведення елементів масиву}
        Ind := SearchLine(MASB, MAXN, Key); {Лінійний пошук ключового
елементу}
        WriteLn('Лінійний пошук. Кількість порівнянь = ', Kol);
        WriteLn('N індексу знайденого елементу (',Key') = ', Ind);
        WriteLn;      {Виведення порожнього рядка}
        SortBubble(MASB, MAXN);   {Сортування методом бульбашки}
        WriteLn('Відсортований масив:');
        ShowArray(MASB, MAXN);   {Виведення елементів масиву}
        Ind := SearchLine(MASB, MAXN, Key); {Лінійний пошук ключового
елементу}
        WriteLn('Лінійний пошук. Кількість порівнянь = ', Kol);
        WriteLn('N індексу шуканого елементу (',Key') = ', Ind);
        Ind := SearchBin(MASB, MAXN, Key); {Бінарний пошук ключового
елементу}
        WriteLn('Бінарний пошук. Кількість порівнянь = ', Kol);
        WriteLn('N індексу шуканого елементу (',Key') = ', Ind);
        ReadLn
END.
```

У наший програмі ми використовуємо і модуль CRT, оскільки нам знадобляться вбудовані процедури очищення екрану (ClrScr) та Randomize. Randomize застосовується для того, щоб випадкові числа були різними при багаторазовому зверненні до функції Random – генератора випадкових чисел.

У розділі операторів оператори наведені у тій послідовності, в якій вони повинні бути згідно сценарію виконання програми. Кожна процедура всередині себе містить повідомлення про свою роботу, щоб ми знали чи

правильно виконується сценарій нашої програми. Ті частини програми, які не були оформлені як процедури, розташовані у головній процедурі (програмі).

Отже, скелет нашої програми готовий. Тепер можна нарощувати її операторами. Проте спочатку зробимо декілька порад-зауважень.

1. При роботі на комп'ютері у жодному випадку не вводьте відразу всю програму цілком, а приєднуйте до неї поступово по одній процедурі з подальшим відлагодженням. Чому? Тому що навіть професійні програмісти припускаються описок або помилок. Отже, у вашій програмі може накопичитись набагато більше помилок, ніж в одній окремо взятій процедурі.
2. Не намагайтеся одразу придумати всі змінні (імена та типи), які вам знадобляться у головній процедурі (глобальні змінні) та у процедурах (локальні змінні).
3. Визначіть, які змінні мають передаватися процедурам як параметри-значення та параметри-змінні, а які використовуватимуться як глобальні змінні.
4. Прагніть вводити за один раз не більше двох-трьох рядків, після чого запускайте програму і дивіться, як вона виконується.
5. Обов'язково користуйтеся порадами та вказівками, які перераховані у розділі 2.8 "Чи є ви хорошим програмістом?".

А зараз приступимо до пояснень основних процедур у самій програмі.

12.6.2. Пошук за ключем у масиві

Лінійний пошук. Коли б ви шукали своє прізвище у списках чи номер телефону у довіднику, які розташовані там без упорядкування за абеткою, то потрібне слово довелось би шукати дуже довго, якщо б воно, звичайно, випадково не потрапило на самісінький початок. А пошук слова у відсортованому словнику займає кілька секунд незалежно від його розташування там.

У цьому розділі ми наведемо два алгоритми. Перший описує пошук "підряд" у невпорядкованій послідовності, другий – той пошук, до якого ми звикли, знаходячи слова у словниках. Замість слів розглянемо цілі числові значення елементів масиву.

Нехай елементи масиву $A[1], A[2], \ldots, A[n]$ та змінна *Key* ("ключ") мають той самий тип T. ***Пошук за ключем*** полягає у знаходженні номера i такого, що $A[i]=Key$. За відсутності такого номера результатом будемо вважати 0. Алгоритм пошуку за таким методом реалізований у функції SearchLine нашої задачі (приклад 12.1).

З'ясуємо, яким чином час пошуку за цим алгоритмом залежить від кількості n елементів масиву. Вочевидь кількість елементарних дій при виконанні функції SearchLine прямо пропорційна кількості порівнянь A[i] <> key. У найгіршому випадку з ключем порівнюються всі елементи масиву.

Звідси максимальна кількість дій та найбільший час виконання функції *прямо (лінійно) пропорційні n*. Описаний спосіб пошуку називається ***лінійним***.

Діхотомічний пошук. Тепер опишемо так званий ***діхотомічний пошук*** у відсортованому словнику. Ми розкриваємо словник приблизно в його середині. Якщо слово повинно бути в словнику далі, то шукати треба лише в другій половині словника. Його середина стає для нас початком, і ми розкриваємо його на середині другої половини. Аналогічно, коли слово повинно бути в першій половині, ми залишаємо для пошуків лише її. Отже, кожне перегляд у словнику поділяє "простір пошуку" на дві половини і зменшує його приблизно вдвічі. Звідси і назва, оскільки **діхотомія** – це поділ на дві половини. Такий пошук ще називають ***двійковим***, **логарифмічним** або ***бінарним***.

Нехай значення елементів масиву упорядковані за зростанням, тобто $A[1] <= A[2] <= \ldots <= A[n]$ (кажуть, ***масив відсортований***). Опишемо діхотомічний пошук функцією SearchBin нашої задачі (приклад 12.1).

Виконання тіла циклу **while** вимагає сталого числа елементарних дій незалежно від значень змінних A, Key, i, left, right. Звідси виходить, що загальна кількість дій прямо пропорційна кількості повторів тіла циклу **while**. Але за кожного повторення різниця right-left зменшується принаймні вдвічі. Спочатку right - left = $n - 1$, тому кількість виконань тіла циклу не більше ніж $\log_2 n$. Ця пропорційність зумовлює ще одну назву описаного пошуку – ***логарифмічний***.

Коли треба знайти значення одного разу, можливо, що комп'ютер впорається досить швидко і за лінійним алгоритмом. Але в реальних задачах доводиться шукати за ключем багаторазово, і різниця між лінійним і двійковим пошуком стає дуже відчутною.

Для двійкового пошуку необхідний відсортований масив, тому у наступному підрозділі розглянемо один із найпростіших способів сортування масивів.

Існує кілька інших способів швидкого пошуку у массивах, які ми наводимо в розділі 12 "Приклади програм". Їх докладний опис є у книзі [Д. Кнут. Икусство прграммирования. т.3. Сортировка и поиск. – М.: Мир, 1978. – 844 с].

12.6.3. Бульбашкове сортування

Розглянемо найпростіший (та найгірший з точки зору витрат часу) спосіб сортування масиву. Нехай $A[1], A[2], \ldots, A[n]$ – масив із довільними значеннями елементів. Порівняємо $A[1]$ та $A[2]$: якщо $A[1]>A[2]$, то поміняємо їхні значення місцями. Потім порівняємо $A[2]$ та $A[3]$ та за необхідності обміняємо їхні значення. В результаті $A[3]$ має найбільше значення серед $A[1]$, $A[2]$, $A[3]$. Продовжимо такі порівняння та обміни до кінця массиву.

Якщо значення елементів розглядати як розміри бульбашок, то ці порівняння та обміни схожі на те, як більші бульбашки відтісняють менших униз і спливають нагору. Тому цей метод називається ***бульбашковим сортуванням***. У результаті найбільша бульбашка стає верхньою, тобто

останній елемент *A*[*n*] має найбільше значення. Наприклад, послідовність значень <4,5,3, 1> перетвориться на <4, 3, 1, 5>.

Далі почнемо все спочатку і перемістимо друге за величиною значення до передостаннього елемента *A*[*n*-1], перетворивши, наприклад, <4, 3, 1, 5> на <3, 1, 4, 5>. Потім третє за величиною значення перемістимо до *A*[*n*-2] тощо. Останній крок складається лише з порівняння *A*[1]<*A*[2] (та обміну значень за потреби).

Опишемо бульбашкове сортування такою процедурою SortBubble (приклад 12.1, bubble – це англійське "бульбашка").

Як бачимо, разом виконується (*n*-1)+(*n*-2)+…+1= *n*× (*n*-1)/2 порівнянь. Очевидно, що найбільше можливе число елементарних дій за цим способом прямо пропорційне кількості порівнянь. Тому час сортування масиву з *n* елементів у такий спосіб *прямо пропорційний n*× (*n*-1). Остаточний варіант програми наведений у прикладі 12.1.

Приклад 12.1.

```
{Пошук ключового елементу  у масиві  методом  лінійного пошуку }
{(пошук в невідсортованому та відсортованому масиві) та дихотомічний }
{пошук (у відсортованому масиві).}
PROGRAM  SearchKey;
USES    CRT;
CONST maxN = 40;  RendN = 100;
TYPE        int = integer;
        indx = 1 .. maxN;
        MasA = array [indx] of int;
VAR  Key, Ind, Kol:    Integer;
        MasB: MasA;
{Заповнюємо масив випадковими числами}
procedure InitArray(var A : MasA; n : Indx);
   var         i: Indx;
   begin
        for i := 1 to N do A[i] := Random (RendN);
   end;
{Отримання випадкового ключа пошуку}
function InitKey: integer;
   begin
        InitKey := Random (RendN);
   end;
{Виведення елементів масиву}
procedure  ShowArray(var A : MasA; n : Indx);
   var         i: Indx;  j, w: integer;
   {Обчислення ширини (w) колонки для роздруку елементів масиву}
   function WidthCol: integer;
        var    r, w: integer;
        begin
```

```
            w := 1;    r := RendN-1;
            repeat
                r := r div 10;
                w := w +1;
            until r = 0;
            WidthCol := w;{повертаємо обчислене значення}
        end;
    begin
        w :=  WidthCol;
        for i := 1 to N do begin
            Write(A[i]:w);
            If  i mod (80 div w) = 0 then
                WriteLn;      {переход на наступний рядок}
        end;
        WriteLn;
    end;

{Сортування методом бульбашки}
procedure SortBubble (var A: MasA; n: Indx);
var     i, k: Indx;
        temp: Int;
begin
    for k := n downto 2 do
   { k – права границя в черговому проході }
        for i := 1 to k –  1 do
            if A[i] > A[i+1] then begin  {міняємо місцями}
                temp:= A[i];
                A[i] := A[i+1];
                A[i+1] := temp
            end;
    end;
{Лінійний пошук ключового елементу}
function SearchLine ( var A : MasA; n : Indx; key : Int) : integer;
begin
     kol := 1;  {кількість порівнянь (глобальна змінна)}
     while ( kol <= n ) do
        if A[kol] <> key then
              kol := kol + 1
        else break;  {вихід з циклу}
    { kol = n + 1 або A[kol] = key }
     if kol < (n + 1) then
        SearchLine := kol
     else SearchLine := 0
end;
```

```
{Бінарний пошук ключового елементу}
function SearchBin ( var A : MasA; n : Indx; key : Int) : integer;
var    i, left, right : integer;
begin
    Kol := 0; {кількість порівнянь (глобальна змінна)}
    left := 1; right := n;
    i := (left + right) div 2;
    while (left < right) do  begin
        Kol := Kol +1;
        if A[i] = key then
            break     {вихід з циклу}
        else
            if A[i] > key then
                right := i –  1 { key може бути лише ліворуч від A[i] }
            else left := i + 1; { key може бути лише праворуч від A[i] }
        i := (left + right) div 2
    end;
    { left >= right або A[i] = key }
    if (i = 0) or (i = n+1) then
        SearchBin := 0
    else
        if A[i] = key then
                SearchBin := i
        else  SearchBin := 0
end;

BEGIN
        ClrScr;                {Очищення екрану, необхідний модуль CRT}
        Randomize; {Щоб випадкові числа були різними при зверненні до
Random()}
        InitArray(MASB, MAXN);  {Заповнюємо масив випадковими числами}
        Key := InitKey; {Отримання випадкового ключа пошуку}
        WriteLn('Кількість елементів масиву = ', MAXN);
        ShowArray(MASB, MAXN);   {Виведення елементів масиву}
        Ind := SearchLine(MASB, MAXN, Key); {Лінійний пошук ключового
елементу}
        WriteLn('Лінійний пошук. Кількість порівнянь = ', Kol);
        WriteLn('N індексу шуканого елементу (',Key') = ', Ind);
        WriteLn;      {Виведення порожнього рядка}
        SortBubble(MASB, MAXN);   {Сортування методом бульбашки}
        WriteLn('Відсортований масив:');
        ShowArray(MASB, MAXN);   {Виведення елементів масиву}
```

```
    Ind := SearchLine(MASB, MAXN, Key); {Лінійний пошук ключового
елементу}
    WriteLn('Лінійний пошук. Кількість порівнянь = ', Kol);
    WriteLn('N індексу шуканого елементу (',Key') = ', Ind);
    Ind := SearchBin(MASB, MAXN, Key); {Бінарний пошук ключового
елементу}
    WriteLn('Бінарний пошук. Кількість порівнянь = ', Kol);
    WriteLn('N індексу шуканого елементу (',Key') = ', Ind);
    ReadLn
END.
```

12.7. Успіхи структурного програмування

Після певного періоду дискусій на тему про припустимість оператора goto ідеї структурного програмування були прийняті світовою спільнотою програмістів. Здобули широку популярність результати перших проектів, які були виконані за технологією структурного програмування. Увагу привернув досвід розробки інформаційної системи для газети «Нью-Йорк Таймс». У проекті, яким керував Харлан Міллз (Harlan Mills), був також використаний новий спосіб організації колективу програмістів, так звана бригада головного програміста. Основну роботу по написанню програми у такій бригаді виконує бригадир. Решта забезпечують йому необхідні умови і вирішують допоміжні завдання. Досвід цієї розробки підтвердив переваги структурного програмування. Трудомісткість створення програми розміром більше 100 000 рядків склала 11 "людино-років", а перша помилка проявила себе лише через 13 місяців після початку експлуатації системи.

Мови програмування, які з'явилися після 1968 року, обов'язково містили достатні засоби для структурного програмування, хоча у багатьох з них оператор переходу все ще був передбачений. Такі мови, як Pascal та C інколи навіть називають мовами структурного програмування, хоча цілі творців цих мов, звичайно ж, не зводилися до правильного оформлення розгалужень та циклів.

Все геніальне просто. «Будь простіше, дурник!» – такий заклик став одним із гасел у боротьбі за поліпшення стилю програмування. Англійською це звучить так: «Keep It Simple Stupid!», скорочено KISS. Проголошений був KISS-принцип і увійшов до жаргону програміста одночасно із структурним програмуванням. У ті роки з'явилося багато книжок, які були присвячені технології програмування. Окрім власне структурного програмування та покрокової деталізації у цих книжках давалося безліч рекомендацій щодо покращення стилю. Йшлося і про розміщення тексту програми, використання коментарів та вибір імен змінних, оптимальний розмір модуля, відмови від витончених прийомів та багато ще чого (див., наприклад [Ван Тассел Д. Стиль, разработка, эффективность, отладка и испытание программ. – М. Мир, 1981. – 316 с]). Зрештою усі ці рекомендації спрямовані на спрощення

читання та розуміння програми шляхом стандартизації та спрощення її структури та стилю. Отже, принцип цей сповна фундаментальний, хоч і несерйозний за формою. Він і дотепер не втратив своєї актуальності.

Так, розділом структурного програмування є підстави вважати й об’єктно-орієнтоване програмування (ООП). Робота Уле-Йохана Дала (O.-J. Dahl), творця першої об’єктно-орієнтованої мови Симула-67 (Simula-67), і К. Хоора «Ієрархічні структури програм», в якій обговорюються основні поняття ООП, такі як класи об’єктів, була опублікована у 1972 році в книзі під назвою «Структурне програмування» разом з роботою Э. Дейкстри [Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М. Мир, 1975. -245 с].

12.8. Декілька слів про об’єктно-орієнтоване програмуванн

Об'єктно-орієнтоване програмування (ООП) являє собою метод програмування, який дуже близько нагадує нашу поведінку. Воно є природною еволюцією більш ранніх нововведень у розробці мов програмування. Об'єктно-орієнтоване програмування є більш структурним, ніж всі попередні розробки, що стосуються структурного програмування. Воно також є більш модульним і більш абстрактним, ніж попередні спроби абстрагування даних і перенесення деталей програмування на внутрішній рівень.

Мовні розширення Borland Pascal надають вам всі засоби об'єктно-орієнтованого програмування: велику структурованість і модульність, велику абстрактність і вбудовану безпосередньо в мову можливість повторного використання. Всі ці характеристики відповідають коду, який є більш структурованим, більш гнучким і більш легким для обслуговування.

Об'єктно-орієнтоване програмування об'єднало кращі ідеї структурованого програмування з низкою потужних концепцій, які сприяють більш ефективної організації програм. У найзагальнішому сенсі будь-яку програму можна організувати одним із двох способів, спираючись на код (дії) або на дані (інформація, на яку спрямовані ці дії). При використанні лише методів структурованого програмування програми зазвичай спираються на код. Такий підхід можна виразити у використанні «коду, який діє на дані».

Механізм об'єктно-орієнтованого програмування заснований на виборі другого способу. У цьому випадку ключовим принципом організації програм є використання «даних, які керують доступом до коду». У будь-якій об'єктно-орієнтованоій мові програмування визначаються дані і процедури, які дозволяється застосувати до цих даних. Таким чином, тип даних в точності визначає, операції якого виду можна застосувати до цих даних.

12.8.1. Історія виникнення об’єктно-орієнтованого програмування

Після Низхідного та Висхідного програмування з'явилася ще одна технологія програмування, що дістала назву технології пакетів прикладних програм. Ідея її полягає в тому, що кожна нова задача розв'язується шляхом

комбінації пакетів програм, якими є готові налагоджені автономні блоки. Пакетна технологія знайшла своє продовження і розвиток в об'єктно-орієнтованому програмуванні.

Об'єктно-орієнтоване програмування виникло в результаті розвитку ідеології процедурного програмування, де дані і підпрограми (процедури, функції) їх обробки формально не пов'язані. Для подальшого розвитку об'єктно-орієнтованого програмування часто велике значення мають поняття події (так зване подієво-орієнтоване програмування) і компонента (компонентне програмування).

Формування компонентного програмування від об'єктно-орієнтованого програмування сталося, як сталося формування модульного від процедурного програмування: процедури сформувалися в модулі – незалежні частини коду до рівня збірки програми, так об'єкти сформувалися в компоненти – незалежні частини коду до рівня виконання програми. Взаємодія об'єктів відбувається за допомогою повідомлень.

Основоположниками об'єктного підходу у програмуванні вважаються норвежці Оле Джохан Дав і Крістен Нюгорт – автори мови Симула. Історія Симула почалася в 1962 р. з проекту Simulation Language, призначеного для програмного моделювання методу Монте-Карло. У 1965 р. у авторів цього проекту виникла ідея об'єднати дані з процедурами, які їх обробляють. У мову були введені нові засоби моделювання і імітації мультипроцессорної роботи, а також терміни «клас» та «об'єкт». Тоді ж виникла і технологія успадкування, тобто творці Симула ввели в мову можливість використання різними класами загальних властивостей за допомогою вказівки назви класу у вигляді префікса.

Алан Кей (Алан Кертіс Кей – американський вчений у галузі теорії обчислювальних систем) ввів нову концепцію розробки програм, відповідно до якої набір послідовно виконуваних інструкцій міг бути замінений на багатовимірне середовище взаємодії об'єктів, що спілкуються один з одним за допомогою асинхронного обміну повідомленнями. Цю ідею він втілив в новій мові SmallTalk. Термін «об'єктно-орієнтоване програмування» вперше був застосований саме по відношенню до мови SmallTalk. Найбільш відома версія мови – SmallTalk-80.

Об'єктно-орієнтовані засоби являють собою інструментарії мови дуже високого рівня, побудовані з інтелектуальних модулів, тобто таких, які виконують певні змістовні функції. Існують інтелектуальні модулі обслуговування екрана, формування меню, підготовки відповідей, контролю помилок при введенні даних користувачем і т. ін. Крім того, в інтелектуальні модулі можуть входити об'єкти, які створені самим користувачем або програмістом для подальшої їх обробки і представлення.

На сьогодні розроблені та продовжують розроблятися нові об'єктно-орієнтовані інструментальні засоби – середовища, в яких реалізовані мови програмування, що, в свою чергу, містять можливості об'єктного програмування. Методика об'єктно-орієнтованого програмування лежить в основі розробки нових прикладних систем та додатків. Зараз майже кожна

традиційна мова програмування розробляється з підмножиною або надмножиною засобів підтримки об'єктного програмування. Це, наприклад, такі нові мови програмування і середовища, як Visual Basic, Visual C++, Visual FoxPro, Clarion, середовища Delphi, Power Builder. Однією з найбільш поширених бібліотек мультиплатформенного програмування є об'єктно-орієнтована бібліотека Qt, написана мовою C++.

12.8.2. Поняття об’єктно-орієнтованого програмування

Об'єктно-орієнтоване програмування виникло в результаті розвитку ідеології процедурного програмування, де дані і підпрограми їх обробки (процедури, функції) формально не пов'язані. Для подальшого розвитку об'єктно-орієнтованого програмування часто велике значення мають поняття події (так зване подієво-орієнтоване програмування) і компонента (компонентне програмування, КОП).

Формування ООП начиналося від процедурного до модульного програмування: процедури сформувалися в модулі – незалежні частини коду до рівня збірки програми, об'єкти сформувалися в компоненти – незалежні частини коду до рівня виконання програми.

ООП являє собою метод програмування, який дуже близько нагадує нашу поведінку. Воно є природною еволюцією більш ранніх нововведень у розробці мов програмування. Об'єктно-орієнтоване програмування є більш структурним, ніж всі попередні розробки, що стосуються структурного програмування. Воно також є більш модульним і більш абстрактним, ніж попередні спроби абстрагування даних і перенесення деталей програмування на внутрішній рівень.

Об'єктно-орієнтоване програмування об'єднало кращі ідеї структурованого програмування з низкою потужних концепцій, які сприяють більш ефективній організації програм. У найзагальнішому сенсі будь-яку програму можна організувати одним із двох способів, спираючись на код (дії) або на дані (інформація, на яку спрямовані ці дії). При використанні лише методів структурованого програмування програми зазвичай спираються на код. Такий підхід можна виразити у використанні «коду, діючого на дані».

Механізм об'єктно-орієнтованого програмування заснований на виборі другого способу. У цьому випадку ключовим принципом організації програм є використання «даних, керуючих доступом до коду». У будь-якій об'єктно-орієнтованій мові програмування визначаються дані і процедури, які дозволяється застосувати до цих даних. Таким чином, тип даних в точності визначає, операції якого виду можна застосувати до цих даних.

Першою мовою програмування, в якій були запропоновані принципи об'єктної орієнтованості, була Симула. У момент своєї появи (в 1967 році), ця мова програмування запропонувала воістину революційні ідеї: об'єкти, класи, віртуальні методи тощо, однак це все не було сприйнято сучасниками як щось грандіозне. Тим не менше, більшість концепцій були розвинені Аланом Кеєм і

Деном Інгаллсом в мові Smalltalk. Саме мова Smalltalk стала першою широко поширеною об'єктно-орієнтованою мовою програмування.

На даний час кількість прикладних мов програмування, що реалізують об'єктно-орієнтовану парадигму, є найбільшим по відношенню до інших парадигм. В області системного програмування досі застосовується парадигма процедурного програмування, і загальноприйнятою мовою програмування є мова С.

Для підтримки принципів об'єктно-орієнтованого програмування всі мови ООП характеризуються такими загальними властивостями: инкапсуляцией, поліморфізмом та успадкуванням. Розглянемо коротко кожне з цих властивостей.

12.8.3. Інкапсуляція

Інкапсуляція – це такий механізм програмування, який пов'язує воєдино код і дані, які їм оброблюються, щоб убезпечити їх як від зовнішнього втручання, так і від неправильного використання. В об'єктно-орієнтованій мові код і дані можуть бути пов'язані способом, при якому створюється самодостатня *чорна скриня*. У цій «чорній скрині» містяться всі необхідні (для забезпечення самостійності) дані і код. При такому зв'язуванні коду і даних створюється об'єкт, тобто ***об'єкт*** – це конструкція, яка підтримує инкапсуляцию.

Всередині об'єкту код, дані або обидві ці складові можуть бути закритими в "рамках" цього об'єкта або відкритими. *Закритий* код (або дані) відомий і доступний тільки іншим частинам того ж об'єкта. Іншими словами, до закритого коду або даних не може отримати доступ та частина програми, яка існує поза цього об'єкта. *Відкритий* код (або дані) доступний будь-яким іншим частинам програми, навіть якщо вони визначені в інших об'єктах. Зазвичай відкриті частини об'єкта використовуються для надання керованого інтерфейсу з закритими елементами об'єкта.

Базовою одиницею інкапсуляції є клас. ***Клас*** визначає новий тип даних, який задає формат об'єкта. Клас включає як дані, так і код, що призначений для виконання над цими даними. Отже, клас пов'язує дані з кодом. Специфікація класу використовується для побудови об'єктів. ***Об'єкти*** - це екземпляри класу. Але по суті, клас являє собою набір планів, які визначають, як будувати об'єкт.

У класі дані оголошуються у вигляді змінних, а код оформляється у вигляді функцій (підпрограм). Функції та змінні, складові класу, називаються його членами. Таким чином, змінна, оголошена в класі, називається ***членом даних***, а функція, оголошена в класі, називається ***функцією-членом***. Іноді замість терміна член даних використовується термін *змінна екземпляра* (або *змінна реалізації*).

Зараз найчастіше до функцій (підпрограм) всередині класу застосовують термін «***метод***». Термін «метод» став популярний з появою мови Java. Те, що в Pascal або C++ програміст називає функцією, Java-програміст називає *методом*,

C#-програмісти також використовують термін «метод». В зв'язку з такою широкою популярністю цей термін все частіше стали застосовувати і до Pascal і C++ функціям.

12.8.4. Поліморфізм

Поліморфізм (від грецького слова polymorphism, що означає "багато форм") – це властивість, що дозволяє використовувати один інтерфейс для цілого класу дій. В якості простого прикладу поліморфізму можна навести кермо автомобіля. Для керма (тобто інтерфейсу) байдуже, який тип рульового механізму використовується в автомобілі. Іншим словами, кермо працює однаково, незалежно від того, чи оснащений автомобіль рульовим управлінням прямої дії (без підсилювача), рульовим управлінням з підсилювачем або механізмом рейкової передачі. Якщо ви знаєте, як поводитися з кермом, ви зможете вести автомобіль будь-якого типу.

Той же принцип можна застосувати до програмування. Розглянемо, наприклад, стек, або список, додавання і вилучення елементів до якого здійснюється за принципом «останнім прибув – першим обслужений». У вас може бути програма, в якій використовуються три різних типи стека. Один стек призначений для зберігання цілочисельних значень, другий – для значень з плаваючою точкою і третій – для символів. Алгоритм реалізації всіх стеків – один і той самий, незважаючи на те, що в них зберігаються дані різних типів. У необ'єктно-орієнтованій мові програмісту довелося б створити три різні підпрограми обслуговування стека, причому підпрограми повинні були б мати різні імена, а також кожна підпрограма – власний інтерфейс. Але завдяки поліморфізму можна створити один загальний набір підпрограм (один інтерфейс), який підходить для всіх трьох конкретних ситуацій. Таким чином, знаючи, як використовувати один стек, ви можете використовувати всі інші.

У більш загальному вигляді концепція поліморфізму виражається фразою «один інтерфейс – багато методів». Це означає, що для групи пов'язаних дій можна використовувати один узагальнений інтерфейс. Поліморфізм дозволяє знизити рівень складності за рахунок можливості застосування одного і того ж інтерфейсу для завдання *загального класу дій*. Вибір же *конкретної дії* (тобто функції) стосовно тієї чи іншої ситуації лягає «на плечі» компілятора. Вам, як програмісту, не потрібно робити цей вибір вручну. Ваше завдання використовувати загальний інтерфейс.

12.8.5. Успадкування

Успадкування це процес, завдяки якому один об'єкт може набуваті властівостей іншого. Завдякі успадкуванню підтрімується концепція ієрархічної класифікації. У вігляді керованої ієрархічної (низхідної) класіфікації організується більшість галузей знань. Наприклад, яблука *Антонівка*, є частинами класіфікації *яблука*, яка в свою чергу є частинами класу *фрукти*, а

тієї частинами ще більшого класу *їжа*. Таким чином, клас *їжа* має пєвні якості (їстівність, поживність тощо), які можна застосувати й до підкласу *фрукти*. Крім ціх якостей, клас *фрукти* має специфічні характеристики (соковітість, солодкість тощо), які відрізняють їх від других харчових продуктів. У класі *яблука* візначаються якості, специфічні для яблук (ростуть на деревах, нетропічні тощо). Клас *Антонівка* успадковує якості всех попередніх класів и при цьому візначає якості, які є унікальними для цього сорту яблук.

Якщо не використовувати ієрархічне представлення ознак, то для кожного об'єкту довелося б в явній формі визначити всі властиві йому характеристики. Але завдяки успадкуванню об'єкту потрібно довизначити тільки ті якості, яке роблять його унікальним всередині його класу, оскільки він (об'єкт) успадковує загальні атрибути свого батька. Отже, саме механізм успадкування дозволяє одному об'єкту представляти конкретний екземпляр більш загального класу.

12.8.6. Сутність об'єктно-орієнтованого підходу до програмування

Основні ідеї об'єктно-орієнтованого підходу спираються на наступні положення:

1. Програма являє собою модель деякого реального процесу, частини реального світу.

2. Модель реального світу або її частини може бути описана як сукупність взаємодіючих між собою об'єктів.

3. Об'єкт описується набором параметрів, значення яких визначають стан об'єкта, і набором операцій (дій), які може виконувати об'єкт.

4. Взаємодія між об'єктами здійснюється посилкою спеціальних повідомлень від одного об'єкта до іншого. Повідомлення, отримане об'єктом, може вимагати виконання певних дій, наприклад, зміни стану об'єкта.

5. Об'єкти, що описані одним і тим же набором параметрів і здатні виконувати один і той самий набір дій, являють собою клас однотипних об'єктів.

Що первинне: алгоритм (процедура, функція) або оброблювані їм дані? У традиційній технології програмування взаємини процедури-дані мають більш-менш вільний характер, причому процедури (функції) є провідними в цій зв'язці: як правило, функція викликає функцію, передаючи дані один одному по ланцюжку. Відповідно, технологія структурного проектування програм насамперед приділяє увагу розробці алгоритму.

У технології ООП взаємовідносини даних і алгоритму мають більш регулярний характер. По-перше, клас (базове поняття цієї технології) об'єднує в собі дані (структурована змінна) і методи (функції). По-друге, схема взаємодії функцій і даних принципово інша. Метод (функція), що викликається для одного об'єкта, як правило, не викликає іншу функцію безпосередньо. Для початку він повинен мати доступ до іншого об'єкта (створити, отримати покажчик, використовувати внутрішній об'єкт у поточному тощо), після чого він вже може викликати для нього один з відомих методів. Таким чином,

структура програми визначається взаємодією об'єктів різних класів між собою. Як правило, має місце ієрархія класів, а технологія ООП інакше може бути названа як програмування "від класу до класу".

Також, з точки зору об'єктно-орієнтованої мови програмування клас об'єктів можна розглядати як тип даного, а окремий об'єкт – як дане цього типу. Визначення програмістом власних класів об'єктів для конкретного набору завдань повинно дозволити описувати окремі завдання в термінах самого класу завдань (при відповідному виборі імен типів та імен об'єктів, їх параметрів і виконуваних дій).

Таким чином, об'єктно-орієнтований підхід передбачає, що при розробці програми повинні бути визначені класи використовуваних у програмі об'єктів і побудовано їх опис, потім створені екземпляри необхідних об'єктів і визначено взаємодію між ними.

Класи об'єктів часто зручно будувати так, щоб вони утворювали ієрархічну структуру. Наприклад, клас "Студент", що описує абстрактного студента, може служити основою для побудови класів "Студент 1 курсу", "Студент 2 курсу тощо, які мають всі властивості студента взагалі і деяки додаткові властивості, що характеризують студента конкретного курсу. При розробці інтерфейсу з користувачем програми можуть використовувати об'єкти загального класу "Вікно" і об'єкти класів спеціальних вікон, наприклад, вікон інформаційних повідомлень, вікон введення даних тощо. У таких ієрархічних структурах один клас може розглядатися як базовий для інших, похідних від нього класів. Об'єкт похідного класу має всі властивості базового класу і деякими власними властивостями, він може реагувати на ті ж типи повідомлень від інших об'єктів, що і об'єкт базового класу і на повідомлення, що мають сенс тільки для похідного класу. Зазвичай кажуть, що об'єкт похідного класу успадковує всі властивості свого базового класу.

Деякі параметри об'єкта можуть бути локалізовані всередині об'єкта і недоступні для прямого впливу ззовні об'єкта. Наприклад, під час руху об'єкта-автомобіля об'єкт-водій може впливати тільки на обмежений набір органів управління (рульове колесо, педалі газу, зчеплення і гальма, важіль перемикання передач) і йому недоступний цілий ряд параметрів, що характеризують стан двигуна і автомобіля в цілому.

Очевидно, для того щоб продуктивно застосовувати об'єктний підхід для розробки програм, необхідні мови програмування, що підтримують цей підхід, тобто дозволяють будувати опис класів об'єктів, утворювати дані об'єктних типів, виконувати операції над об'єктами. Однією з перших таких мов стала мова SmallTalk, в якій всі дані є об'єктами деяких класів, а загальна система класів будується як ієрархічна структура на основі визначених базових класів.

Досвід програмування показує, що будь-який методичний підхід в технології програмування не повинен застосовуватися сліпо з ігноруванням інших підходів. Це стосується і об'єктно-орієнтованого підходу. Існує ряд типових проблем, для яких його корисність найбільш очевидна. До таких проблем відносяться, зокрема, завдання імітаційного моделювання,

програмування діалогів з користувачем. Існують і завдання, в яких застосування об'єктного підходу ні до чого, крім зайвих витрат праці, не приведе. У зв'язку з цим найбільшого поширення набули об'єктно-орієнтовані мови програмування, що дозволяють поєднувати об'єктний підхід з іншими методологіями. У деяких мовах і системах програмування застосування об'єктного підходу обмежується засобами інтерфейсу з користувачем (наприклад, Visual FoxPro ранніх версій).

Контрольні запитання для самоперевірки

1. Що явдяє собою технологія програмування?
2. Які головні складові включає структурне програмування?
3. З яких етапів складається розробка програми?
4. Що являє собою нисхідне програмування?
5. Які переваги має розробка програмних модулів?
6. У чому полягає оформлення програм?
7. Що являє собою технологія об'єктно-орієнтованого програмування?
8. На яких основних властивостях базується об'єктно-орієнтоване програмування?
9. У чому полягає сутність об'єктно-орієнтованого програмування?

13. Оцінки часу виконання алгоритмів

Оскільки впродовж всього навчання ми мали справу з алгоритмами, то на закінчення обговоримо одну з характеристик алгоритму – його складність. Розвиток і вдосконалення комп'ютерної техніки спричинили створення всіляких алгоритмів, що забезпечують розв'язання багаточисельних прикладних завдань, причому навіть для однотипних завдань створювалося (і створюється) багато алгоритмів. Подібні алгоритми раніше були названі еквівалентними. Виконання будь-якого алгоритму вимагає певного об'єму пам'яті комп'ютера для розміщення даних і програми, а також часу центрального процесора для обробки цих даних – ці ресурси обмежені і, отже, правомірне питання про ефективність їх використання. Таким чином, в найширшому сенсі поняття ефективності пов'язане з усіма обчислювальними ресурсами, які необхідні для роботи алгоритму.

Часова складність стає особливо важливою для задач, які передбачають інтерактивний режим роботи програми, або для завдань управління в режимі реального часу. Часто програмісту, який створює програму управління яким-небудь технічним пристроєм, доводиться шукати компроміс між точністю обчислень та часом роботи програми. Як правило, підвищення точності веде до збільшення часу.

Об'ємна складність програми стає критичною, коли обсяг оброблюваних даних виявляється на межі обсягу оперативної пам'яті ЕОМ. Для сучасних комп'ютерів гострота цієї проблеми знижується завдяки зростанню обсягу їх ОЗУ і ефективному використанню багаторівневої системи запам'ятовуючих пристроїв. При цьому програмі стає доступною дуже велика, практично необмежена область пам'яті (віртуальна пам'ять), і брак основної пам'яті призводить лише до деякого уповільнення роботи через виконання обмінів з диском. Існують прийоми, що дозволяють мінімізувати втрати часу при такому обміні: використання кеш-пам'яті і апаратного перегляду команд програми вперед, що дозволяє завчасно переносити з диска в основну пам'ять необхідні значення.

У програмуванні зазвичай під "найефективнішим" розуміється алгоритм, що забезпечує найшвидше здобуття результату, оскільки в практичних ситуаціях саме обмеження за часом часто є домінуючим чинником, що визначає придатність того чи іншого алгоритму. Зі сказаного можна зробити висновок, що мінімізація ємнісної складності за розміром не є першочерговим завданням, тому далі ми будемо цікавитися в основному часовою складністю алгоритмів. Детальніше про складність алгоритмів ви познайомитеся при вивченні відповідних дисциплін.

Як правило, часова складність алгоритму залежить від вхідних даних, причому як від величин, так і від їх обсягу, а також від кількості виконуваних операцій.

Часова складність алгоритму **–** це функція, яка кожній вхідній довжині слова n ставить у відповідність максимальний (для всіх конкретних однотипних завдань довжиною N) час, що витрачається алгоритмом на її розв'язаення. При

цьому, безумовно, передбачається, що в усіх завданнях використовується однакова схема кодування вхідних слів.

Для оцінки продуктивності алгоритмів можна використовувати різні підходи. Найхитріший – просто запустити кожен алгоритм на декількох завданнях і порівняти час виконання. Інший спосіб – математично оцінити час виконання підрахунком операцій.

Розглянемо алгоритм обчислення значення многочлена степені *n* в заданій точці *x*.

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_i x^i + ... + a_1 x^1 + a_0$$

Алгоритм 1 – для кожного доданку, окрім a0 піднести x до заданого ступеня послідовним множенням і потім помножити на коефіцієнт. Потім доданки скласти.

Обчислення і-го доданку ($i=1..n$) вимагає і множень. Значить, всього $1 + 2 + 3 + ... + n = n(n+1)/2$ множень. Крім того, потрібне n+1 складання. Всього $n(n+1)/2 + n + 1 = n2/2 + 3n/2 + 1$ операцій.

Алгоритм 2 – винесемо *x*-и за дужки і перепишемо многочлен (методом Горнера) у вигляді

$$P_n(x) = a_0 + x(a_1 + x(a_2 + ... (a_i + .. x(a_{n-1} + a_n x))).$$

Наприклад,

$$P_3(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0 = a_0 + x(a_1 + x(a_2 + a_3x))$$

Обчислюватимемо вираз зсередини. Сама внутрішня дужка вимагає 1 множення і 1 додавання. Її значення використовується для наступної дужки..., тобто, 1 множення і 1 додавання на кожну дужку, яких у нас *n-1*. І ще після обчислення самої зовнішньої дужки помножити на *x* і додати a_0. Всього *n* множень + *n* додавань = *2n* операцій.

Частенько така детальна оцінка не потрібна. Замість неї наводять лише асимптотичну швидкість зростання кількості операцій при збільшенні *n*.

Функція $f(n)= n^2/2 + 3n/2 + 1$ зростає приблизно як $n^2/2$ (відкидаємо порівняно повільно зростаючий доданок 3n/2+1). Константний множник 1/2 також прибираємо і отримуємо асимптотичну оцінку для *Алгоритму 1*, яка позначається спеціальним символом $O(n^2)$ (читається як "О велике від ен квадрат", "О" – від англійського order, тобто порядок).

Це – верхня оцінка, тобто кількість операцій (а значить, і час роботи) зростає не швидше, ніж квадрат кількості елементів. Аби відчути, що це таке, погляньте на таблицю, де наведені числа, які ілюструють швидкість росту для декількох різних функцій.

N	**log n**	**n*log n**	**n^2**
1	0	0	1
16	4	64	256

256	8	2,048	65,536
4,096	12	49,152	16,777,216
65,536	16	1,048,565	4,294,967,296
1,048,576	20	20,969,520	1,099,301,922,576
16,775,616	24	402,614,784	281,421,292,179,456

Якщо вважати, що числа в таблиці відповідають мікросекундам, то для завдання з *n=1048576* елементами алгоритму з часом роботи *O(log n)* буде потрібно 20 мікросекунд, алгоритму з часом *O(n)* – 17 хвилин, а алгоритму з часом роботи $O(n^2)$ – більше 12 днів... Тепер перевага *Алгоритму 2* з оцінкою *O(n)* перед *Алгоритмом 1* досить очевидна.

Найкращою є оцінка *O(1)*. В цьому випадку час взагалі не залежить від *n*, тобто постійний при будь-якій кількості елементів.

Таким чином, *O()* – "урізана" оцінка часу роботи алгоритму, яку частенько набагато простіше отримати, чим точну формулу для кількості операцій.

Отже, сформулюємо два правила формування оцінки *O()*.

При оцінці за функцію береться кількість операцій, що зростає найшвидше.

Тобто, якщо в програмі одна функція, наприклад, множення, виконується *O(n)* разів, а додавання – $O(n^2)$ разів, то загальна складність програми – $O(n^2)$, оскільки врешті-решт при збільшенні *n* швидші (у певне, константне число разів) додавання стануть виконуватися настільки часто, що впливатимуть на швидкодію куди більше, ніж повільні, але рідкі множення. Символ *O()* показує виключно асимптотику!

При оцінці *O()* константи не враховуються. Хай один алгоритм робить *2500n + 1000* операцій, а інший – *2n+1*. Обидва вони мають оцінку *O(n)*, оскільки час їх виконання зростає лінійно.

Зокрема, якщо обидва алгоритми мають оцінку, наприклад, як *O(n*log n)*, то це зовсім не означає, що вони однаково ефективні. Перший може бути, скажімо, в 1000 разів ефективніше. *O()* означає лише те, що їх час зростає приблизно як функція *n*log n*.

Інший наслідок опускання константи – алгоритм з часом $O(n^2)$ може працювати значно швидше за алгоритм *O(n)* при малих *n*. За рахунок того, що реальна кількість операцій першого алгоритму може бути *n2 + 10n + 6*, а другого – *1000000n + 5*. Втім, другий алгоритм рано чи пізно обжене перший, n^2 зростає куди швидше *1000000n*.

Основа логарифму всередині символу *O()* не пишеться. Причина цього вельми проста. Нехай у нас є $O(\log_2 n)$. Але $\log_2 n = \log_3 n / \log_3 2$, а $\log_3 2$, як і будь-

яку константу, асимптотика – символ *O()* не враховує. Таким чином, $O(\log_2 n) = O(\log_3 n)$.

Зазвичай при аналізі алгоритмів розрізняють випадки найгіршої і очікуваної поведінки. Важко строго визначити, що таке "очікувана" поведінка, тому що визначення залежить зазвичай від припущень про можливі вхідні дані. Тому, найчастіше, ми можемо точно вказати найгірший випадок (*O()*), хоча інколи і тут можна помилитися. Наприклад, для швидкого сортування (quicksort) в найгіршому випадку час роботи зростає як $O(n^2)$, а середній ("очікуваний") час – як $O(n*\log n)$. Добре реалізоване швидке сортування дійсно зазвичай поводиться як $O(n \log n)$.

Розглянемо верхні оцінки деяких основних алгоритмів:

Запис	Назва часу	Приклад
O(1)	Константне	Індексування масиву
O(log n)	Логарифмічне	Двійковий пошук
O(n)	Лінійне	Порівняння рядків
O(n log n)	n log n	Quicksort
$O(n^2)$	Квадратичне	Прості методи сортування
$O(n^3)$	Кубічне	Перемножування матриць
$O(2^n)$	Експоненціальне	Перебір всіх підмножин

Доступ до елементу в масиві – операція, що працює за константний ($O(1)$) час.

Алгоритм, за кожен крок що відсіває половину вхідних даних, як двійковий пошук, зазвичай займе час $O(\log n)$.

Порівняння двох рядків довжиною в n символів за допомогою strcmp займе $O(n)$.

Прості методи сортування оцінюються як $O(n^2)$.

Традиційний алгоритм множення двох квадратних матриць порядку n займає $O(n^3)$, оскільки кожен елемент виходить в результаті перемножування n пар чисел і підсумовування результатів, а всього елементів n^3.

Експоненціальний час роботи алгоритму зазвичай є результатом перебору всіх варіантів: в множині з n елементів – 2^n різних підмножин, тому алгоритм, якому треба пройтися по всіх підмножинах, виконуватиметься за час $O(2^n)$, тобто буде експоненціальним. Експоненціальні алгоритми зазвичай дуже довго працюють, якщо n не дуже мале, оскільки додавання одного елементу подвоює час роботи алгоритму. На жаль, існує багато задач, таких як, наприклад, знамените "задача комівояжера", для яких відомі лише експоненціальні рішення. Коли задача така, то часто замість точних рішень беруть алгоритми, що знаходять деяке наближення до відповіді.

На закінчення розглянемо ще два приклади оцінки часової складності алгоритму.

Потрібно оцінити часову складність алгоритму обчислення факторіала цілого позитивного числа. Програма рішення:

```
Function Factorial(x:  Integer):  Integer;
Var m,  i  :  Integer;
Begin
    m  : =  1;
    For i  := 2  To x Do m := m*i;
    Factorial  := m
End;
```

Підрахуємо загальне число операцій, які виконуються програмою при даному значенні *x*. Один раз виконується оператор *m: = 1*; тіло циклу (у якому дві операції: множення і присвоєння) виконується *(x-1)* разів; один раз виконується присвоєння *Factorial: = m*. Якщо кожну з операцій прийняти за одиницю складності, тимчасова складність всього алгоритму буде мати вигляд *1+2*(x-1)+1 = 2x*, звідки зрозуміло, що в якості параметра тимчасової складності слід прийняти значення *x*. Функція тимчасової складності вийде наступною: *O(n)*. У цьому випадку можна сказати, що часова складність лінійно залежить від параметра даних – значення аргументу функції *Factorial*.

Потрібно обчислити скалярний добуток двох векторів: $A = (a_1, a_2, a_k)$, $B = (b_1, b_2, ..., b_k)$. Програма рішення:

```
AB  := 0;
For i  := 1 To k Do AB  :=AB+A[i]*B[ i ];
```

У цій задачі обсяг вхідних даних *n = 2k*. Число виконуваних операцій *1+ 3k = 1+ 3 (n/2), k = n/2*. Складність алгоритму залежить від значень елементів векторів *A* і *B*. Як і в попередньому прикладі, тут можна говорити про лінійну залежність часової складності від параметра даних.

Контрольні запитання для самоперевірки

1. Що являє собою часова складність алгоритму і від чого вона задежить?
2. Які ісгують методи оцінки часу виконання програм?
3. Які зазвичай розрізняють оцінки часу виконання програм?
4. Як поділяються програми за часом виконання?

14. Поняття динамічного програмування

Говорячи про рекурсію, ми маємо на увазі спеціальний спосіб побудови програми, при якому в тексті з'являються процедури, що викликають самі себе в процесі виконання. Але можливість «звернення до себе» дає рекурсивній процедурі абсолютно унікальні властивості, що роблять рекурсию потужним і багатогранним інструментом.

Досить часто програмісти працюють з величинами, розрахунок яких має рекурентний характер. Розглянемо як приклад задачу обчислення факторіала. Визначення факторіала свідчить, що N! =1*2*...*N.

Це не рекуррентне визначення. Виконавши просте перетворення, отримаємо:

N! = N*(N- 1)!.

А це вже рекуррентне визначення, тобто те визначення, в якому функція факторіалу визначається через неї ж, але від меншого аргументу.

Рекурсія природним чином виникає там, де вдається дати величині рекуррентне визначення. Факторіал можна визначити рекуррентно, тому для факторіала можлива побудова рекурсивної процедури. Проте використання рекурсії не завжди виправдано. Приклад розрахунку факторіалу не тільки добре показує механізм рекурсії, але це також хороший приклад того, коли рекурсію не треба використовувати.

Чому пошук рекурсивного рішення не завжди виправданий? Якщо в тексті процедури існує виклик її ж, то в момент передачі управління на цей виклик виконується активація нової копії процедури з новим набором локальних змінних, при цьому набір локальних величин тієї копії, яка з'явилася причиною активації, також зберігається в оперативній пам'яті. Таким чином, рекурсивний процес веде до того, що в оперативній пам'яті зберігаються набори даних всіх активованих копій процедур. Знищення даних кожної копії відбувається тільки по завершенні роботи копії та повернення управління в точку виклику процедури.

Наприклад, для обчислення 20! буде виконано 20 активацій і в оперативній пам'яті виникне 20 наборів даних. Зрозуміло, що це не завжди розумна трата пам'яті. Тим більше, що обчислення факторіалу легко реалізується елементарною послідовністю ітерацій. Звідси – просте правило: якщо задача реалізується лінійною послідовністю ітерацій, то рекурсивне рішення можливе, але не розумно. Тому пошук рекурсивного рішення виправданий, якщо процес не можна уявити у вигляді лінійної послідовності ітерацій, тобто якщо процес розгалужений.

Для завершення послідовності рекурсивних викликів необхідно визначити деякий параметр, у загальному випадку числову функцію, змінна якої обчислюється на кожному кроці рекурсивного процесу і для якої визначається одне або кілька значень, які є сигналом для переривання рекурсивних активацій. Досить часто ця функція зводиться до числового параметру, який

має початкове значення, що запускає рекурсивний процес і, звичайно, його зупиняє.

Будь-яку задачу можна вирішити, як рекурсивно так і нерекурсивно. Питання полягає тільки в доцільності. Іноді буває, що рекурсивне рішення сильно спрощує життя програмісту і при цьому не вимагає занадто великих ресурсів, в цьому випадку рекурсивне рішення безумовно виправдане. Іноді буває, як у задачі про факторіал, ресурсів потрібно небагато, але нерекурсівние рішення все ж простіше. А іноді буває, що рекурсивне рішення дуже просте, але вимагає стільки ресурсів, скільки ми не можемо собі дозволити, в цьому випадку необхідно шукати нерекурсівне рішення. У такому пошуку програміст може піти складним шляхом – шляхом пошуку математичного рішення і чисто технічним шляхом – шляхом імітації рекурсії.

Необхідно розуміти, що рекурсія – це механізм, який забезпечений можливостями мови і можливостями компілятора. Неважко припустити, що цей механізм припускає узагальнення до методу мислення, що не зводиться до властивостей мови програмування. Таким методом є метод **динамічного програмування**.

Нехай, наприклад, вирішується якась задача, що вимагає певного розміру обчислень. Нехай дана задача млже бути представлена як сума завдань меншого обчислювального обсягу. Тоді рішення задачі в цілому може бути отримано вирішенням задач меншого обсягу і їх «підсумовуванням».

Природно, що термін «підсумовування» не можна розуміти в арифметичному сенсі. Можливість підсумовування означає, що підзадачі незалежні, в тому сенсі, що рішення однієї з них не впливає на рішення іншої. Рекурсивне визначення задачі через підзадачі є окремим випадком динамічної задачі.

Динамічним програмуванням називається метод, який дозволяє вирішувати «переборні» задачі, спираючись на вже вирішені підзадачі меншого розміру. При цьому необхідно, щоб усі рішення можна було запам'ятати в таблиці (тобто дані, пораховані одного разу, не перераховуються).

Ключова ідея в динамічному програмуванні досить проста. Як правило, щоб вирішити поставлену задачу, вимагається вирішити окремі частини задачі (підзадачі), після чого об'єднати рішення підзадач в одне загальне рішення. Часто багато з цих підзадач однакові. Підхід динамічного програмування полягає в тому, щоб вирішити кожну підзадачу тільки один раз, скоротивши тим самим кількість обчислень. Це особливо корисно у випадках, коли число підзадач, що повторюються, експоненціально велике. Словосполучення «динамічне програмування» уперше було використане в 1940-х роках Р. Беллманом.

Ідеї динамічного програмування дуже схожі на ідеї методу доказу за індукцією, відомого з шкільної програми математики.

Складемо план вирішення задачі методом динамічного програмування :

1. Записати те, що вимагається знайти в завданні, як функцію від деякого набору аргументів (числових, строкових або ще яких).

2. Звести вирішення задачі для цього набору параметрів до вирішення аналогічних підзадач для інших наборів параметрів (як правило, з меншими значеннями). Якщо завдання нескладне, то корисно буває виписати явне рекурентне співвідношення, яке задає значення функції для цього набору параметрів.
3. Задати початкові значення функції, тобто ті набори аргументів, при яких завдання тривіальне і можна явно вказати значення функції.
4. Створити масив (чи іншу структуру даних) для зберігання значень функції. Як правило, якщо функція залежить від одного цілочисельного параметра, то використовується одновимірний масив, для функції від двох параметрів двовимірний масив і т. д.
5. Організувати заповнення масиву з початкових значень, визначаючи черговий елемент масиву за допомогою виписаного на кроці 2 рекурентні співвідношення або алгоритми.

В якості прикладу розглянемо найпростішу комбінаторну задачу: скількома способами можна з даних *n>= 0* предметів вибрати деякі *k* предметів (*0<=k<=n*), якщо порядок їх вибору не важливий? Відповіддю на це завдання є величина $C_n^k = \frac{n!}{k!(n-k)!}$ звана числом сполучень з *n* елементів по *k*. Запис *n*! означає *1 • 2 • 3 • ... • n*, зване факторіалом числа *n*, при цьому вважається, що 0! = 1.

Розглянемо вирішенея цієї задачі кількома методами.

Наївний метод. Як же обчислити значення C_n^k по даним *n* и *k*? Скористаємося виведеною формулою. Функція *factorial (n)* повертає факторіал числа *n*, а функція *Com (n, k)* повертає значення C_n^k .

```
Function Factorial(n: Integer): Integer;
Var i, j, f  Integer;
Begin
        f := l;
        for  i:=2 to n do
           f := f*i;
        Factorial := f;
End;
Function  Com(n,  k: Integer): Integer;
Begin
        Com :=  factorial(n)/factorial(k)/factorial(n-k);
End;
```

Цей алгоритм, зрозуміло, дуже швидкий (його обчислювальна складність *O(n)*, оскільки обчислення значення *n!* вимагає *n-1* множень), використовує обмежений розмір додаткової пам'яті. Але у нього є істотний недолік: він буде коректно працювати тільки для невеликих значень *n* і *k*. Річ у тому, що величина *n!* дуже швидко росте зі збільшенням *n*. Наприклад, значення 13! = 6 227 020 800 перевершує максимальне можливе значення, яке можна записати в

32-розрядному цілому числі, а величина 21! = 51 090 942 171 709 440 000 не вміщується в 64-розрядному цілому числі. Тому користуватися цією функцією можна тільки для *n<=12* при використанні 32-бітової арифметики або для *n<=20* при використанні 64-бітової арифметики. При цьому саме значення C_n^k може бути невелике, але при проведенні проміжних обчислень факторіалів може виникнути переповнення. Наприклад, C_{30}^{15} = 155117520 можна записати з використанням 32-бітової арифметики, проте, значення 30! при цьому не поміститься і в 64-розрядному цілому числі і обчислити значення C_{30}^{15} таким методом не вдасться.

Рекурсивний метод. Проблем з переповнюванням можна уникнути, якщо скористатися для обчислення відомою формулою: $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ (при $0 < k < n$).

Дійсно, вибрати з *n* предметів *k* можна C_n^k способами. Помітимо один з даних *n* предметів. Тоді всі вибірки можна розбити на дві групи: в які входить позначений предмет (їх буде C_{n-1}^{k-1}) і які не містять поміченого предмету (таких вибірок буде C_{n-1}^k).

Додавши до цієї формули крайові значення: $C_n^n = C_n^0 = 1$ (вибрати всі предмети або не вибрати жодного предмета можна єдиним способом), можна написати рекурсивну функцію обчислення числа сполучень:

```
Function  Com(n,  k: Integer): Integer;
Begin
      if (k=0) or (k=n) then
         Com := 1;
      else
         Com :=  Com(n-1,k-l)+Com(n-l,k);
End;
```

При такому вирішенні задачі проблем з переповненням не виникне: якщо значення кінцевої відповіді не призводить до переповнення, то оскільки при його знаходженні ми підсумовуємо менші натуральні числа, всі проміжні значення, які повертаються функцією Com (n, k: Integer) теж не будуть призводити до переповнення, і така програма, наприклад, зможе обчислити значення C_{30}^{15} .

Але таке рішення вкрай погано тим, що працювати воно буде дуже довго, оскільки функція Com(n,k) викликатиметься багаторазово для одних і тих же значень параметрів. Наприклад, якщо ми викличемо функцію Com(30,15). то вона викличе функції Com(29,14) і Com(29,15). Функція Com(29,14) викличе функції Com(28,13) і Com(28,14), а Com(29,15) викличе функції Com(28,14) і Com(28,15). Ми бачимо, що функція Com(28,14) буде викликана двічі. Зі збільшенням глибини рекурсії кількість викликів функції, що повторюються, швидко зростає: функція Com(27,13) буде викликана три рази (двічі її викличе функція Com(28,14) і ще один раз її викличе Com(28,13) і так далі.

Таким чином, складність такого рекурсивного алгоритму для обчислення C_n^k можна оцінити не менше, аніж саме значення C_n^k .

Метод динамічного програмування. Отже, одна з проблем рекурсивних алгоритмів (і ця проблема виникає вельми часто, не тільки в розглянутому прикладі) тривала робота рекурсії за рахунок повторюваних викликів рекурсивної функції для однакових наборів параметрів. Щоб не витрачати машинний час на обчислення значень рекурсивної функції, які ми вже колись вирахували, можна зберегти ці значення в масиві і замість рекурсивного виклику функції ми будемо брати це значення з масиву. У задачі обчислення числа сполучень створимо двовимірний масив *B* і будемо в елементі масиву *B[n,к]* зберігати величину C_n^k . В результаті отримаємо наступний масив (по рядках значення *n*, починаючи з *0*, за стовпчиками значення *k*, починаючи з *0*, кожен елемент масиву *B[n, к]* дорівнює сумі двох елементів: що стоїть безпосередньо над ним *B[n-1, к]*, і який стоїть над ним зліва *B [n-1, к-1]*).

Отримана числова таблиця, складена із значень C_n^k , в якій кожен елемент дорівнює сумі двох елементів, що стоять над ним, називається *трикутником Паскаля*.

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		
1	5	10	10	5	1	
1	6	15	20	15	6	1

Оскільки кожен елемент цього масиву дорівнює сумі двох елементів, що стоять в попередньому рядку, то цей масив треба заповнювати по рядках. Відповідна функція, що заповнює масив і обчислює необхідне число поєднань може бути такою:

```
Function  Com(n,  k: Integer): Integer;
Var i, j, f  Integer; B: array[1..10] of byte;
Begin
      For  i=0 to i<=n do;                { Заповнюємо і-й рядок масиву}
      Begin
            B[i,0] := l;   {На кінцях рядка стоять одиниці}
            B[i,i] := l;
            For  j=l to j<i do    { Заповнюємо остальні елементи і-го рядка}
                B[i,j] := B[i-l,j-l]+B[i-l,j];
      End;
      Com :=  B[n,k];
End;
```

Наведений алгоритм для обчислення C_n^k вимагає обсягу пам'яті $O(n^2)$, і така ж його часова складність (для заповнення одного елемента масиву потрібно $O(1)$ операцій, всього ж елементів у масиві $O(n^2)$).

Подібний метод рішення, коли одна задача зводиться до менших підзадач, обчислені ж відповіді для менших підзадач зберігаються у масиві або іншій структурі даних, називається ***динамічним програмуванням***. У розглянутій задачі обчислення числа поєднань використання методу динамічного програмування замість рекурсії дозволяє істотно зменшити час виконання програми за рахунок деякого збільшення об'єму використовуваної пам'яті.

Контрольні запитання для самоперевірки

1. Що являє собою метод динамічного програмування?
2. Яка основна проблема рекурсивних алгоритмів?
3. Коли доцільно використовувати рекурсивні алгоритми?
3. Як використовуються масиви при динамічному програмуванні?

15. Приклади програм

Навчитися програмувати можна тільки програмуючи – цей постулат програміста виник, як результат багаторічної роботи авторів посібника, і підтверджений багатьма вченими в області програмування. Іншого способу немає. Але перш ніж самому почати проектувати алгоритми та записувати їх на мовах програмування, корисно розібрати велику кількість різноманітних алгоритмів та програм, які реалізовують їх.

У цьому розділі ми наведемо приклади кількох програм на різні теми, з метою закріплення пройденого матеріалу. Знання необхідних теоретичних основ, які ви отримали в попередніх розділах цього посібника, тепер допоможуть вам для аналізу наведених програм та практичного написання своїх програм.

Тематично розділ-практикум розбитий на декілька підрозділів, які охоплюють обробку числової, текстової та графічної інформації, а також програмування елементарних ігор у DOS-оточенні.

Практично всі задачі та алгоритми, зрозуміло, не є новими. У деяких окремих випадках наведені посилання на конкретну книгу або конкретного автора. Разом з тим в деяких випадках алгоритми, що реалізовані у програмах, викладені лаконічно, коротко та виразно.

Таким чином, цілі даного розділу-практикуму наступні:

1. Швидке залучення читача до самостійного і свідомого складання закінчених програм на популярних прикладах мови програмування Pascal.
2. Засвоєння основних навичок алгоритмічної та програмістської грамотності:
   - чіткого та зрозумілого стилю;
   - надійності рішень;
   - економії обчислень і т.і.

15.1. Арифметичні алгоритми

У цьому розділі ми наведемо низку простих програм, які не вимагають особливих знань для аналізу та створення алгоритмів, а лише тих знань, які ви отримали у середній школі.

Приклад 15.1.1.

```
{Обчислення висоти трикутника за його трьома сторонами}
program Vysota_Tr;
var a, b, c: real;
      h, p: real;
begin
   repeat
        write(‘ Введіть сторони трикутника (a,b,c): ‘);
        readln(a,b,c);
```

```
  until ((a+b) > c) and ((a+c) > b) and ((b+c) > a);
{Півпериметр}
  p := (a+b+c)/2;
{Висота}
  h := (2*sqrt(p*(p-a)*(p-b)*(p-c)))/a;
  Writeln(‘ Висота трикутника = ‘,h);
  ReadLn;
end.
```

Приклад 15.1.2.

{Перевірка двійкового числа на парність (непарність)}

```
PROGRAM  Chet2_Nechet;
var  A: Byte;
     i: 0..7;
     bin: 0..1;
BEGIN
  Write (‘ Введіть ціле число від 1 до 255: ‘);
  Readln (A);
  Write(‘ Число в двійковому уявленні = ‘);
  For i:=7 downto 0  do
     Write ((A SHR i) AND $01);       {виділяємо і друкуємо і-тий розряд числа}
  Writeln;
  bin := A AND $01;  {виділяємо молодший розряд}
  If  bin = 0 then
     Writeln(‘ Число парне’)
  else
     Writeln(‘ Число непарне’);
  Readln;
END.
```

Приклад 15.1.3.

{Перевірка десяткового числа на парність}

```
program Chet_Nechet;
var N: integer;
begin
  write(‘Введіть ціле число: ‘);
  readln(N);
  if (N mod 2) = 0 then
     Writeln(N’, –  число парне ‘)
  else
     Writeln(N’, –  число непарне ‘);
  ReadLn;
end.
```

Приклад 15.1.4.

```
{Підрахунок одиниць (нулів) у двійковому поданні числа}
PROGRAM  Chet2_Nechet;
var  A, edin, nuli: Byte;
    i: 0..7;
    bin: 0..1;
BEGIN
  Write (‘ Введіть ціле число від 1 до 255: ‘);
  Readln (A);
  Write(‘ Число у двійковому уявленні = ‘);
  edin := 0;
  nuli := 0;
  For i:=7 downto 0  do
    begin
      bin := (A SHR i) AND 1;   {виділяємо і-тий розряд}
      If bin = 0 then
        nuli := nuli + 1
      else
        edin := edin + 1;
      Write ((A SHR i) AND 1);  {друкуємо і-тий розряд}
    end;
  Writeln;
  Writeln(‘У 2-му записі числа цифра "0" зустрічається ‘, nuli ‘ раз, цифра "1" –
‘, edin ‘ раз’);
   Readln;
END.
```

Приклад 15.1.5.

```
{ Дано натуральне число N (N >= 2). Знайти всі менші N прості числа }
{ використовуючи решето Ератосфена }
Program Eratosfen;
Const N = 255;
       P: set of Byte = [2..N];    {множина цілих чисел від 2 до N}
var   i,j: byte;
begin
 for i:=2 to N do
       if i in P then
       begin
            Write(i:4);              { P:= P –  [i*j]}
            for j:=i to N div i do
               Exclude(P,i*j)       {вилучення кратних}
       end;
 Readln
End.
```

Приклад 15.1.6.

```
{Дано натуральне число N. Підрахувати кількість цифр числа}
Program KOL_Zifr;
var i,k: Integer;
    N: LongInt;
begin
  Write('Введіть натуральне число > 0 : ');
  Readln(N);
  до := 0;
  While N > 0 do begin
        K := K+1;
        N := N div 10;
  end;
  WriteLn('Кількість цифр числа = ', K);
  Readln
End.
```

Приклад 15.1.7.

```
{Дано натуральне число N. }
{Підрахувати кількість цифр числа, суму цифр, 1-у цифру}
Program KOL_Zifr;
        var z, Kol, Sum: Integer;
            N: LongInt;
begin
  Write('Введіть натуральне число > 0 : ');
  Readln(N);
  Z := 0;  Sum := 0;
  Kol := 0;
  While N > 0 do begin
        Z := N mod 10;
        Kol := Kol+1;
        Sum := Sum + Z;
        N := N div 10;
  end;
  WriteLn('Перша цифра числа = ', Z);
  WriteLn('Сума цифр числа = ', Sum);
  WriteLn('Кількість цифр числа = ', Kol);
  Readln
End.
```

Приклад 15.1.8.

```
{Обчислити  2^n методом зрушення}
Program Stepen_N;
        var   N: Byte;
              M: LongInt;
```

```
begin
  Write('Введіть натуральне число (покажчик ступеня) < 15: ');
  Readln(N);
  M := (1 SHL N);{Зсув 1 ліворуч, отримуючи 2^14, 2^13 ..., 2^1, 2^0}
  WriteLn('2^',N,'=',M);
  Readln
End.
```

Приклад 15.1.9.

{Перетворення з 10 у 2 систему числення, використовуючи формулу:}
{A10 = Bn^2n+ Bn-12^(n-1)+…+ B12^1+ B02^0, де Bi – двійкові цифри}

```
Program Dec_Bin;
var i, K, N, M, R: Integer;
begin
  Write('Введіть натуральне число: ');
  Readln(N);
  Write('У двійковій системі числення = ');
   R := 0;     {Ознака не друку лідируючих нулів зліва}
   for i := 14 downto 0 do begin {старший ступінь двійки в числі типу Integer =
2^14}
      M := 1 SHL i;         {Зсув 1 ліворуч, отримуючи 2^14, 2^13 ..., 2^1, 2^0}
       K := 0;
       while N >= M do begin    {скільки разів міститься ступінь 2 у відповідому
розряді числа}
            K := K + 1;
            N := N –  M;
        end;
        if i = 0 then R := 1;
        R := R + K;
        if R <> 0 then Write(K);  {друкуємо розряди числа зліва направо}
  end;
  Readln
End.
```

Приклад 15.1.10.

{Перетворення з 10 в 8 систему числення, використовуючи формулу:}
{A10 = Bn8^n+ Bn-186(n-1)+...+ B18^1+ B08^0, де Bi – вісімкові цифри}

```
Program Dec_Oct;
       var i, K, N, M, R: Integer;
begin
       Write('Введіть натуральне число: ');
       Readln(N);
  Write('У 8-ій системі числення = ');
  R := 0;       {Ознака не друкувати лідируючих нулів зліва}
  i := 12;      {старший ступінь 8 для Integer –  2^12 = 8^4}
  While i >= 0 do begin
```

```
        M := 1 SHL i;       {Зсув 1 ліворуч, отримуючи 8^4=2^12, 8^3=2^9 ..., 8^1,
8^0}
        K := 0;
        while N >= M do begin   {визначаємо скільки разів міститься ступінь 8 у
відповідному розряді числа}
                K := K + 1;
                N := N –  M;
        end;
        if   i = 0 then R := 1;
        R := R + K;
        if R <> 0 then Write(K); {друкуємо розряди числа зліва направо}
             i := i –  3;
   end;
   Readln
End.
```

Приклад 15.1.11.

{Переведення з 10 в 16 систему числення, використовуючи формулу:}
{A10 = Bn16^n+ Bn-116^(n-1)+...+ B116^1+ B016^0, де Bi – 16-кові цифри}

```
Program Dec_Hex;
var i, K, N, M, R: Integer;
begin
   Write('Введіть натуральне число: ');
   Readln(N);
   Write('У 16-ій системі числення = ');
   R := 0;     {Ознака не друкувати лідируючих нулів зліва}
   i := 12;    {старший ступінь 16 для Integer –  2^12 = 16^3}
   While i >= 0 do begin
        M := 1 SHL i;       {Зсув 1 ліворуч, отримуючи 16^3=2^12, 16^2=2^8,
16^1=2^4, 16^0}
        K := 0;
        while N >= M do begin {поки міститься ступінь 16 у відповідному розряді
числа}
           K := K + 1;          N := N –  M;
        end;
        if i = 0 then R := 1;
        R := R + K;
        if R <> 0 then begin  {друкуємо розряди числа зліва направо}
                if K < 10 then Write(K)
                else if K = 10 then Write('A')
                else if K = 11 then Write('B')
                else if K = 12 then Write('C')
                else if K = 13 then Write('D')
                else if K = 14 then Write('E')
```

```
            else if K = 15 then Write('F');
        end;
        i := i –  4;
    end;
    Readln
End.
```

Приклад 15.1.12.

{Переведення чисел з однієї системи числення (від 2 до 16) в іншу (від 2 до 16)}

```
program Perevod;
{uses Crt;}
const
        max = 255;
        Zifra:  array[0..15] of string =
('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F');
{Цифри, які використовуються при перетворенні чисел у різні системи
числення 2-16}
var
   In_S, Out_S :byte;  {In_S –  з якої, Out_S –  в яку систему числення
переводити}
   I, J, K, Len_Ch        :byte;    {допоміжні змінні}
   N                      :string;  {початкове число, яке потрібно переводити}
   Arr_Zifr               :array[1..max] of byte;
   Ch10                   :longint;
   Oshibka, Err           :boolean;
   Str                    :string;  {допоміжні змінні}
function Stepen(Chislo, Step: byte):longint; {піднесення числа (Chislo) до ступеня
(Step)}
var
   k1    :longint;
   i1    :byte;
begin
  k1 := 1;
  for  i1 := 1 to Step do k1 := k1*Chislo;
  Stepen := k1;
end;
begin
 { ClrScr;}
  repeat
    Write('Введіть систему числення переведення  (2-16): ');
    Readln(In_S);
  until (In_S > 1) and (In_S < 17);
  repeat
```

```
   Write('Введіть число в (',In_S') системі: ');
   Readln(N);
   {перевірка на наявність помилок, при введенні числа}
   Len_Ch := length(N);     {знаходимо довжину числа}
   j := 1;
   Oshibka := True;
   for i := 1 to Len_Ch do begin
     Str  := Copy(N,i,1); {виділяємо з рядка один символ для перетворення його у
число}
     Str  := UpCase(Str[1]);       {переводимо символ у верхній регістр}
     Err  := False;            {перевірка на наявність помилок при введенні числа}
     for до :=0 to In_S –  1 do begin
         if  Zifra[k]= Str  then    {якщо символ є в масиві, то цифра введена
правильно}
         begin
           Arr_Zifr[j]:= до;                  {додати її у масив цифр}
           j := j+1;
           Err := True;
         end;
     end;
     if Err = False then Oshibka := False;
   end;
   if Oshibka = False then
     WriteLn('Помилка при введенні числа: цифри не входять в ',In_S', систему
числення.');
  until Oshibka;
  repeat
    Write('У яку систему числення переводити?(2-16): ');
    Readln(Out_S);
  until (Out_S > 1) and (In_S < 17);
  {Переведення числа у десяткове уявлення}
  Ch10 := 0;
  for i := 1 to Len_Ch do  Ch10 := Ch10+Arr_Zifr[i]*Stepen(In_S,Len_Ch –  i);
  {у Ch10 – число, яке переведене у десяткову систему числення}
  {переводимо число Ch10 з десяткової системи числення у задану}
  i:=0;
  repeat
      i := i+1;
     Arr_Zifr[i]:= Ch10 mod Out_S; {залишок від ділення на основу системи
перетворення}
     Ch10 := Ch10 div Out_S;
  until Ch10 < Out_S;
  if Ch10 <> 0 then begin          {остання (старша) ненульова цифра}
     i := i+1;
```

```
    Arr_Zifr[i]:= Ch10;
  end;
  Write(N ‘ (‘,In_S’) = ‘);
  for j := i downto 1 do Write(Zifra[Arr_Zifr[j]]);   {виведення перетвореного
числа}
  Write(‘ (‘,Out_S’)’);
  Writeln;
  ReadLn;
end.
```

Приклад 15.1.13.

{Перетворення арабських чисел у римські}

```
Program DecToRoman;
 Uses Crt;
 Var   Dec: Integer;
     Roman: String;
Function Rom(Dec: Integer): String;
Const
   Arabic: array [1..13] of Integer = ( 1,   4,  5,  9,  10,  40, 50,  90, 100,400,
500,900,1000);
   Romans: array [1..13] of String  =
(‘I’,’IV’,’V’,’IX’,’X’,’XL’,’L’,’XC’,’C’,’CD’,’D’,’CM’,’M’);
Var
    i:   Integer;
    Str: String;
 Begin
     Str := ‘‘;
     for i := 13 downto 1 do
          while (Dec >= Arabic[i]) do begin
             Dec := Dec –  Arabic[i];
              Str := Concat(Str, Romans[i]);
          end;
     Rom := Str;
 End;

Begin
    Clrscr;
    Writeln(‘Перетворення арабських чисел у римські’);
    Write(‘Введіть ціле десяткове число: ‘);
    Readln(Dec);
    Roman := Rom(Dec);
    Writeln(‘У римському уявленні = ‘,Roman);
    ReadKey;
end.
```

Приклад 15.1.14.

```
{Перетворення римських чисел в арабські}
Program RomanToDec;
 Uses Crt;
 Var   Dec: Integer;
          Roman: String;
Function Arab(Str: String): Integer;
 Const
    Arabic: array [1..13] of Integer = ( 1,   4,  5,  9,  10,  40, 50,  90, 100,400,
500,900,1000);
    Romans: array [1..13] of String  =
('I','IV','V','IX','X','XL','L','XC','C','CD','D','CM','M');
 Var
    i,j: Integer;
    Dec: Integer;
    S:   String;
 Begin
   Dec := 0;
   j := 1;
   while j <= length(Str) do begin
      if (j+1) <= length(Str) then begin
         S := Copy(Str,j,2);
         for i := 1 to 13 do begin
            if S = Romans[i] then begin
               Dec := Dec + Arabic[i];
               j := J + 2;
               break;
            end;
         end;
      end;
      S := Str[j];
      for i := 1 to 13 do begin
         if  S = Romans[i] then begin
            Dec := Dec + Arabic[i];
            j := j + 1;
            break;
         end;
      end;
   end;
   Arab := Dec;
 End;

 Begin
   Clrscr;
```

```
   Writeln('Перетворення римських чисел в арабські');
   Write('Введіть число у римській системі числення: ');
   Readln(Roman);
   Dec := Arab(Roman);
   Writeln('Арабське  уявлення = ',Dec);
   ReadKey;
end.
```

Приклад 15.1.15.

{Дізнатися, чи ділитися дане натуральне число на 4 (на 8)}

```
Program Div_4_8;
var N: Integer;
begin
  Write('Введіть натуральне число: ');
  Readln(N);
  if (N and $03) = 0 then
        Write('Число ',N', ділиться на 4')
  else
        Write('Число ',N', не ділиться на 4');
  if (N and $07) = 0 then
        WriteLn(' і ділиться на 8')
  else
        WriteLn(' і не ділиться на 8');
  Readln
End.
```

Приклад 15.1.16.

{ Обчислити y=1!+2!+3!+...+n!, n>1 }

```
Program Faktor;
var     i,k,N: integer;
        f,y: Longint;
begin
  Write('Введіть натуральне число 1 < N < 200: ');
  Readln(N);
  y := 0;
  for i := 1 to N do begin
        f := 1;
        for до := 1 to i do
            f := f*k;
        y := y + f;
  end;
  WriteLn('y = ',y);
  Readln
End.
```

Приклад 15.1.17.

```
{ Обчислити y=sqrt(3+sqrt(6+...sqrt(96+sqrt(99))))}
Program Kor;
var     i: Byte;
        y: real;
begin
     i := 99;
    y := 0;
    While i >= 3 do begin
         y := sqrt(i+y);
         i := i –  3;
   end;
   WriteLn(‘y = ‘,y);
    Readln
End.
```

Приклад 15.1.18.

```
{ Дани три натуральні числа. Визначити їх найбільшого спільного
дільника. }
PROGRAM  NOD_3;
 var
         A,B,C: Integer;

FUNCTION  NOD(A,B: Integer): Integer;
var i: integer;
BEGIN
  While A <> B do  begin
         If A > B then
         A:= A-B
       else
          B:= B-A;
  end;
  NOD := A;
END;

BEGIN
     Write (‘Введіть три натуральні числа: ‘);
     ReadLn (A,B,C);
     A := NOD(A,B);
     A := NOD(A, C);
     WriteLn (‘НОД = ‘,A);
     ReadLn
END.
```

Приклад 15.1.19.

```
{Дани довжини a,b і із сторін деякого трикутника. Знайти медіани трикутника}
{сторонами якого є медіани початкового трикутника.}
{ Довжина медіани, яка проведена до сторони a, дорівнює 0.5*sqrt(2*b^2 +
2*c^2 –  a^2)}
program Med_Tr;
var a,b,c,Ma,Mb,Mc,MTA,MTb,MTc: real;
```

```
{Функція обчислення медіани трикутника}
function Mediana(a,b,c: real): real;
begin
  Mediana := 0.5*sqrt(2*b*b+2*c*c-a*a);
end;

begin
  repeat
       write('Введіть сторони трикутника a, b, з: ');
       readln(a,b,c);
  until ((a+b)>з) and ((a+c)>b) and ((b+c)>a);
  Ma := Mediana(a,b,c);
  Mb := Mediana(b,a,c);
  Mc := Mediana(c,a,b);
  MTa:= Mediana(Ma,Mb,Mc);
  MTb:= Mediana(Mb,Ma,Mc);
  MTc:= Mediana(Mc,Ma,Mb);
  writeln('Медіани трикутника:');
  writeln(MTa);
  writeln(MTb);
  writeln(MTc);
  ReadLn;
end.
```

Приклад 15.1.20.

```
{Дани координати вершин двох трикутників.}
{Визначити, який з них має більшу площу.}
Program Vhod_Treug;
var Xa1,Xb1,Xc1,Ya1,Yb1,Yc1,Xa2,Xb2,Xc2,Ya2,Yb2,Yc2: Byte;
     Lab,Lac,Lbc: Real;
     Sabc1,Sabc2: Real;
{Функція обчислення сторони трикутника}
function D_Tr(Xa1,Xb1,Ya1,Yb1: real): real;
begin
  D_Tr := sqrt(sqr(Xa1-Xb1)+sqr(Ya1-Yb1));
end;

{Функція обчислення площі трикутника}
function S_Tr(Lab,Lac,Lbc: real): real;
var p: Real;
begin
  p := (Lab+Lbc+Lac)/2;
  S_Tr := sqrt(p*(p-Lab)*(p-Lac)*(p-Lbc));
end;
begin
 WriteLn('Введіть координати вершин 1-го трикутника');
 Write('Input Xa1, Ya1 = '); ReadLn(Xa1,Ya1);
 Write('Input Xb1, Yb1 = '); ReadLn(Xb1,Yb1);
 Write('Input Xc1, Yc1 = '); ReadLn(Xc1,Yc1);
 WriteLn('Введіть координати вершин 2-го трикутника');
 Write('Input Xa2, Ya2 = '); ReadLn(Xa2,Ya2);
```

```
  Write('Input Xb2, Yb2 = '); ReadLn(Xb2,Yb2);
  Write('Input Xc2, Yc2 = '); ReadLn(Xc2,Yc2);

  Lab := D_Tr(Xa1,Xb1,Ya1,Yb1);
  Lac := D_Tr(Xa1,Xc1,Ya1,Yc1);
  Lbc := D_Tr(Xb1,Xc1,Yb1,Yc1);
  Sabc1 := S_Tr(Lab,Lac,Lbc);

  Lab := D_Tr(Xa2,Xb2,Ya2,Yb2);
  Lac := D_Tr(Xa2,Xc2,Ya2,Yc2);
  Lbc := D_Tr(Xb2,Xc2,Yb2,Yc2);
  Sabc2 := S_Tr(Lab,Lac,Lbc);
  if Sabc1 > Sabc2 Then WriteLn('Площа 1-го > 2-го');
  if Sabc1 < Sabc2 Then WriteLn('Площа 1-го < 2-го');
  If Sabc1 = Sabc2 Then WriteLn('Площі рівні');
  Readln
End.
```

Приклад 15.1.21.

```
{Два прості числа називаються  близнятами, якщо вони відрізняються }
{один від одного на 2}
{(наприклад, числа 41 і 43). Надрукувати всі пари "близнят" з відрізка [n,2n] }
{ де n –  задане ціле число, більше два.}

Program Twin_Ch;
var     Xa1,Xb1,Xc1,Ya1,Yb1,Yc1,Xa2,Xb2,Xc2,Ya2,Yb2,Yc2: Byte;
        Lab,Lac,Lbc: Real;
        Sabc1,Sabc2: Real;
{Функція обчислення простого числа }
Function Prost(n): boolean;
 var b: boolean;
      i: integer;
begin
   b := true;
   if n mod 2 = 0 then
     b := false
   else
     begin
       i := 3;
      while i < n to
      begin
         if n mod i = 0 then
         begin
            b := false;
```

```
            break;
          end;
          i := I+2;
       end;
     end;
  Prost := b;
end;

begin
  Write('Введіть натуральне число N більше 2-х:');
  ReadLn(N);
  P1 := 0;
  P2 := o;
  For i:=N to N do
        if Prost(i) then
        begin
            P1 := P2;
            P2 := i;
        end;
        if (P1 <> 0) and (P1 = P2) then
        begin
          Write('Input Xc2, Yc2 = ');
          ReadLn(Xc2,Yc2);
        end;
  end;
  Write('Input Xc2, Yc2 = '); ReadLn(Xc2,Yc2);
  Readln
End.
```

Приклад 15.1.22.

```
{Трикутник задано координатами його вершин. На площині задається крапка}
{Визначити, належить вона трикутнику (на лінії), знаходиться в середині }
{або поза трикутником}
Program Vhod_Treug;
        var Xa,Xb,Xc,Xd,Ya,Yb,Yc,Yd: Byte;
            Lab,Lac,Lbc,Lad,Lbd,Lcd: Real;
        Sabc,Sabd,Sbcd,Sacd,p: Real;
begin
  Write('Input Xa, Ya = '); ReadLn(Xa, Ya);
  Write('Input Xb, Yb = '); ReadLn(Xb,Yb);
  Write('Input Xc, Yc = '); ReadLn(Xc,Yc);
  Write('Input Xd, Yd = '); ReadLn(Xd,Yd);
  Lab := sqrt(sqr(Xa-Xb)+sqr(Ya-Yb));
  Lac := sqrt(sqr(Xa-Xc)+sqr(Ya-Yc));
```

```
  Lbc := sqrt(sqr(Xb-Xc)+sqr(Yb-Yc));
  Lad := sqrt(sqr(Xa-Xd)+sqr(Ya-Yd));
  Lbd := sqrt(sqr(Xb-Xd)+sqr(Yb-Yd));
  Lcd := sqrt(sqr(Xc-Xd)+sqr(Yc-Yd));
  p := (Lab+Lbc+Lac)/2;
  Sabc := sqrt(p*(p-Lab)*(p-Lac)*(p-Lbc));
  p := (Lab+Lbd+Lad)/2;
  Sabd := sqrt(p*(p-Lab)*(p-Lbd)*(p-Lad));
  p := (Lbc+Lbd+Lcd)/2;
  Sbcd := sqrt(p*(p-Lbc)*(p-Lbd)*(p-Lcd));
  p := (Lac+Lcd+Lad)/2;
  Sacd := sqrt(p*(p-Lac)*(p-Lcd)*(p-Lad));
  if ((Sacd+Sabd+Sbcd) –  Sabc) > 1 Then WriteLn('Out side');
  if abs((Sacd+Sabd+Sbcd)-Sabc)< 1 Then WriteLn('In side');
  If (Sabd = 0) or (Sbcd = 0) or (Sacd = 0) then WriteLn('On line');
  Readln
End.
```

Приклад 15.1.23.

```
{ Умовний оператор REPEAT }
{ Ця програма визначає, чи є число простим }
program P_Chislo;
Var     Number: Integer;   {Досліджуване число}
        Count:  Integer;    {Можливий дільник}
Begin
   WriteLn('Яке число повинне бути перевірене?: ');
   Read(Number);
   Count := 1;
   Repeat
      Count := Count+1;
  until (Number mod Count)=0;
  If Count = Number then
      WriteLn(Number', є простим числом')
  else
      WriteLn(Number', ділиться на ',Count);
  ReadLn
End.
```

Приклад 15.1.24.

```
{ Арифметичні алгоритми: піднесення цілого числа до натурального
ступеня  }
{ Варіант 2 (швидший і ефективніший, ніж ми розглядали, при   XY = eYLn X )
}
```

```
var x,y:integer;

function Degree(a,b:integer):longint;
var r:longint; c:integer;
begin
    r:=1; c:=a;
    while b>0 do begin
      if odd(b) then begin  {функція odd(x) повертає true, якщо X непарне число}
                r:=r*c;
                dec(b);
           end else begin
                c:=c*c;
                b:=b div 2;
       end;
    end;
    Degree:=  r;
end;
begin
   writeln('введіть число і (через пропуск) ступінь числа');
   readln(x,y);
   writeln(Degree(x,y)); { print x^y }
end.
```

Приклад 15.1.25.

{Дані натуральні числа a, b. Обчислити добуток
a*b, використовуючи у програмі лише операції +, =, <>.}

```
Var   a, b, c, k: integer;
begin
        write('Введіть два натуральні числа a і b: ');
        readln(a,b);
        k := 0; c := 0;
        { c = a * k}
        while  k <> b do begin
             k := k + 1;
             c := c + a;
        end;
        {c = a * k  та  k = b, отже, c = a * b}
        writeln('a*b = ',c);
        readln
end.
```

Приклад 15.1.26.

{Дано натуральне (ціле додатнє) число a та b. }

```
{Обчислити частку q та залишок r при діленні a на d, }
{не використовуючи операцій div та mod.}
{Згідно із визначенням, a = b * d + r, 0 <= r < d.}

Var   a, b, r, q: integer;
begin
      write('Введіть натуральні числа a та b (a >= 0; b > 0): ');
      readln(a,b);
      {a >= 0; b > 0}
      r := a; q := 0;
      { a = q * d + r, 0 <= r}
      while  not (r < b) do begin
         {r >= d}
         r := r –  b; {r >= 0}
         q := q + 1;
      end;
      writeln('q = ',q,'  r = ',r);
      readln
end.
```

Приклад 15.1.27.

```
{Скласти програму, яка друкує розкладання на прості множники
задане натуральне число n > 0 (іншими словами, потрібно друкувати
лише прості числа і добуток надрукованих  чисел має бути рівним  n.}
Var   n, i, k: integer;
begin
  write('Введіть натуральне число n: ');
       readln(n);
  k := n;
  { добуток надрукованих чисел k  дорівнює n, надруковані лише прості числа}
 while not (k = 1) do begin
     i := 2;
     {инвариант: k не має дільників в интервале (i,l)}
     while k mod i <> 0 do begin
        i := i + 1;
     end;
     {i –  наименший у відповід k, більший за 1, отже, простий}
     writeln (i);
     k:=k div i;
  end;
  readln
end.
```

Приклад 15.1.28.

```
{Програма обчислює K-ю цифру послідовності чисел Фібоначчі}
program N_K_Fib;
uses crt;
var k,i,s,b,c,n,a:longint;

function Fib(i:integer):longint;
begin
    if i<=2 then
       Fib:=1 {функція визначення чисел Фібоначчі}
    else
       Fib:=Fib(i-2)+Fib(i-1);
end;

begin
    clrscr;
    write('Enter K = ');
    readln(k); {номер цифри}
    writeln('Cifri rjada Fibonachi do K:');
    s:=0;i:=0;
    while s<k do begin
       i:=i+1;
       b:=Fib(i); {числа Фібоначчі }
       c:=1;
       n:=10;
       while b div n<>0 do begin
          n:=n*10;
          c:=c+1; {рахуємо скільки цифр у числі}
       end;
       s:=s+c; { рахуємо скільки цифр всього }
       write(b);
    end;
    s:=s-c; {повертаємося назад, шукаємо потрібну цифру}
    n:=n div 10;
    while s<>k do begin
       a:=b div n mod 10;
       n:=n div 10;
       s:=s+1;
    end;
    writeln;
    write('Zadannaja cifra=',a);
    readln;
end.
```

Приклад 15.1.29.

```
{ Арифметичні алгоритми: моделювання додавання двійкових чисел }
var  sr,sf,ss:string;

function BinAdd(s1,s2:string):string;
var  s:string; l, i, d, carry:byte;
begin
   {вирівнювання рядків по довжині}
   if length(s1)> length(s2) then
        while length(s2)< length(s1) do s2:='0'+s2
    else
        while length(s1) <length(s2) do s1:='0'+s1;
   l := length(s1);
   s := '';  carry:=0;
   for i:=l downto 1 do begin
      d := (ord(s1[i]) -ord('0')) + (ord(s2[i]) -ord('0')) + carry;
      carry := d div 2;
      d :=  d mod 2;
      s := char(d+ord('0'))+ s;
   end;
   if carry <> 0 then s := '1'+s;
   BinAdd :=s;
end;

begin
    writeln('введіть 1-е двійкове число:');
    readln(sf);
    writeln('введіть 2-е двійкове число:');
    readln(ss);
    sr:=BinAdd(sf,ss);
    writeln('результат додавання = ',sr);
end.
```

Приклад 15.1.30.

```
{Арифметичні алгоритми: моделювання віднімання двійкових чисел }
var   sr,sf,ss:string;
{ віднімання двійкових рядків, перше число повинно бути >= другого }
function BinSub(s1,s2:string):string;
var   s:string; l,i,j:byte;
begin
   {вирівнювання рядків по довжині}
   if length(s1) >length(s2) then
        while length(s2) <length(s1) do s2:='0'+s2
   else
```

```
        while length(s1) <length(s2) do s1:='0'+s1;
    l := length(s1); {початок алгоритму віднімання}
    s := '';
    for I := l downto 1 do begin
      case s1[i] of
       '1': if s2[i]='0' then s:='1'+s else s:='0'+s;
       '0': if s2[i]='0' then s:='0'+s else begin
              s:='1'+s;
              if (s1[i-1]='1') then
                s1[i-1]:='0'
              else begin
                j:=1;
                while (i-j>0) and (s1[i-j]='0') do begin
                          s1[i-j]:='1';
                          inc(j);
                end;
                s1[i-j]:='0';
              end;
            end;
      end;
    end;
    {Знищення передніх нулів}
    while (length(s) >1) and (s[1]='0') do delete(s,1,1);
    BinSub:=s;
end;
BEGIN
     writeln('введіть 1-е двійкове число:');
     readln(sf);
     writeln('введіть 2-е двійкове число:');
     readln(ss);
     sr:=BinSub(sf,ss);
     writeln('результат віднімання = ',sr);
END.
```

Приклад 15.1.31.

{Є n бактерій червоного кольору. Через 1 такт часу червона бактерія міняється на зелену, потім через 1 такт часу ділиться на червону та зелену. Скільки буде всіх бактерій через *k* тактів часу? }

```
uses crt;
var i,k,n,z,nz,nk:longint;
begin
        clrscr;
        write('кіл-сть бактерій:'); readln(n);
        write('кіл-сть тактів часу:'); readln(k);
```

```
        z:=0;
        for i:=1 to k do   begin
           nz:=0;
           nk:=0;
           nz:=nz+z;
           nk:=nk+z;
           nz:=nz+n;
           n:=nk;
           z:=nz;
        end;
        n:=z+n;
        writeln('otvet=',n); readln;
end.
```

Приклад 15.1.32.

```
{ Дано число n. Вилучити з нього всі одиниці та п'ятірки, залишивши порядок
цифр }
{ Приклад: 527012 перетвориться в 2702 }
uses crt;
var     b : array[1..10]of string;
        a,c : string;
        i,j,k : integer;
begin
        clrscr;
        write('введіть число '); readln(a);
        j:=0; k:=0; c:='';
        for i:=1 to length(a) do
             if (a[i]<>'1') and (a[i]<>'5') then  begin
                j:= j+1;
                k:= k+1;
                b[j]:= a[i];
        end;
        for j:=1 to k do c:=c+b[j];
        write('отримали число ',c);
        readln;
end.
```

Приклад 15.1.33.

{Обчислення цілої частини кореня квадратного з додатнього числа.}
{Ідея алгоритма полягає у тому, що сумма *K* перших непарних чисел дорівнює K^2.}
{Наприклад, $1+3 = 2^2$, $1+3+5 = 3^2$, $1+3+5+7 = 4^2$, $1+3+5+7+9 = 5^2$ і т.д.}

```
PROGRAM  Kvadrat_Kor;
```

```
  var  i,j: Integer;
       x  : Real;    { Результат }
BEGIN
  Write (‘Введіть додатнє число, з якого ‘);
  Write (‘хочете знайти корень: ‘); ReadLn (x);
  i:=-1; j:=0;
  While  j<=x  do
    begin  i:=i+2; j:=j+i  end;
  Write (‘Результат: ‘,(i-1) DIV 2);
  ReadLn
END.
```

Приклад 15.1.34. Рішення диофантових рівнянь.

Наступна програма призначена для вирішення діофантових рівнянь. Відкриття діофантових рівнянь пов'язано з ім'ям грецького математика Діофанта. Іноді аналіз простого на вигляд нелінійного диофантова рівняння може викликати величезні труднощі навіть для математика високої кваліфікації. Використовуючи комп'ютер, оснащений системою програмування на TURBO-PASCAL, ми зможемо застосувати його можливості для вирішення як лінійних, так і нелінійних діофантових рівнянь.

Найпростіше лінійне диофантово рівняння має вигляд $ax + by = c$, де a, b і c – задані числа, а x і y – невідомі. Особливість цих рівнянь полягає у тому, що для них шукаються цілочисельні рішення. Це можна зробити методом перебору.

Рішення лінійного диофантова рівняння. Вирішимо наступну старовинну задачу. Селянин витратив 100 рублів на покупку 100 різних домашніх тварин. Кожна корова обійшлася йому у 10 рублів, свиня у 3 рублі, а вівця у 50 копійок. Припускаючи, що селянин придбав принаймні по одній тварині кожного виду, знайдемо, скільки голів худоби кожного виду він купив. Умова задачі записується у вигляді двох рівнянь:

$$10x + 3y + z/2 = 100;\quad x + y + z = 100,$$

де x, y, z – кількість корів, свиней і овець відповідно. Позбудемося знаменника у першому рівнянні, помноживши його на 2. З отриманого таким чином рівняння віднімемо друге. Це дозволяє виключити змінну z. Отримуємо рівняння $19x + 5y = 100$. Рішеннями даного рівняння повинні бути цілі позитивні числа, менші 100. Наступна програма призначена для вирішення даного рівняння.

```
Program Diophant_1;
var
   x, y: Integer;
begin
   WriteLn(’Цілі рішення рівняння 19x + 5y = 100 з діапазону’);
   WriteLn(’ 1<=x<=100, 1<=y<=100: ’);
   For  x := 1 to 100 do
      For y := 1 to 100 do
```

```
        If 19*x + 5*y = 100 then
            WriteLn(’(x, y) = )’,x,’,’,y,’)’);
  Write(‘Натисніть <Enter>’);
  ReadLn;
end.
```

Рішення нелінійного диофантова рівняння. Рішення нелінійних діофантових рівнянь – це більш складна проблема. Але комп'ютер і вміле застосування методів обчислювальної математики часто дозволяють швидко отримати рішення навіть найскладніших завдань. Ось приклад кубічного диофантова рівняння: $x^3 = y^2 + 2$.

Відомо його рішення: $x = 3, y = 5$. Ускладнимо завдання і вирішимо рівняння $x^3 = y^2 + 63$.

```
Program Diophant_2;
var
  x, y, z, w, n: Longint;
begin
  { Спочатку знайдемо найбільшу n, для якого n*2 + 63 <= MaxLongint.
  MaxLongint = 2 147 483 647 – вбудована константа модуля System,
  задає максимальне значення змінної типу Longlnt}
  n := MaxLongint - 63;
  n := Trunc(Sqrt(n)); {Trunc(t) – ціла частина величини t}
  n := n - 8;
  x := 0;
  WriteLn('Все цілі рішення рівняння x^3 = y^2 + 63.');
  WriteLn('для 1 <= y <= ', n, ':');
  for y := 1 to n do
  begin
    z := y * y + 63;
    repeat
      Inc(x); {Функція Inc(x) збільшує значення x на одиницю }
      w := x * x * x;
    until w >= z;
    if w = z then
      WriteLn('(x, y) = (', x, ', ', y, ')')
    else
      Dec(x); {Функція Dec(x) зменшує значення x на одиницю }
  end;
  Write('Натисніть <Enter>');
  ReadLn;
end.
```

Для того щоб збільшити діапазон пошуку рішень рівняння, цілі змінні програми *x, y, z, w, n* описані як змінні типу LongInt («довге ціле»). Діапазон значень для типу «довге ціле» становить [- 2147483648, +2 147483647].

Приклад 15.1.35. Обрахунок квадратного кореня з числа методом Герона.

Метод Герона – це метод послідовних наближень. Якщо задано число *a* і з нього потрібно наближено обчислити квадратний корінь, то спочатку обирається довільне початкове наближення *x0*. Потім задається точність обчислень $e > 0$ і будується послідовність $x_{n+1} = (x_n + a/ x_n) /2$. Обчислення припиняються при виконанні умови $|x_{n+1} - x_n| < e$.

Цикл while у подпрограмі-функції *geron* виконується доти, доки не порушиться задана в ньому умова (нерівність). Неможливо заздалегідь передбачити, коли це відбудеться, тому, в такій ситуації цикл *for* був би даремний. Підпрограма *geron* викликається в операторі WriteLn. Такий виклик функції в операторі виведення припускається. Ц другому операторі виведення міститься звернення до вбудованої функції обрахунку квадратного кореня. Таким чином, можна порівняти обидва результати.

```
Program Geron_Sqrt;

function geron(x: Real): Extended;
const
   eps = 1.0E-100;
var
   y, z: Real;
begin
   y := 1.0;
   {Цикл while виконується доти, доки не буде досягнута
    задана точність обчислення квадратного кореня }
   while Abs(z - y) >= eps do
   begin
      z := y;
      y := (y + x / y) / 2;
   end;
   geron := y;
end;
begin
   WriteLn( 'Алгоритм Герона для обчислення квадратного кореня з двійки:');
   WriteLn('Geron(2.0) =', geron(2.0));
   WriteLn('Sqrt(2.0) =', Sqrt(2.0));
   Write('Натисніть <Enter>');
   ReadLn;
end.
```

Приклад 15.1.36. Вирішенняя нерівності.

У наступному прикладі мова йде про рішення нерівності $b^n <= a <= b^{n+1}$ щодо *n* за умови *a>= 1, b> 1*. Нерівність вирішується перебором значень *n*, метод вирішення реалізований у функції **largest**, аргументами якої є значення *a* і *b*.

```
program Inequality;

function 1argest(a, b: LongInt): Word;
var
   n: Word;
   x: LongInt;
begin
   x := b;
   n := 0;
   while x <= a do
   begin
      x := b*x;
      Inc(n);
   end;
   largest := n;
end;
begin
   WriteLn('3^n <= 10000 < 3^(n+l)');
   WriteLn('n = ', largest(10000, 3));
   Write('Натисніть <Enter>');
   ReadLn;
end.
```

Приклад 15.1.37.

```
{ Вирішення системи 3-х рівнянь з трьома невідомими  }
{ ------------------------------------------------------------------}
{ вирішення рівнянь вигляду:                                        }
{ |a1*x + b1*y + c1*z = d1|                                         }
{ |a2*x + b2*y + c2*z = d2|                                         }
{ |a3*x + b3*y + c3*z = d3|                                         }
{ метод рішення:                                                    }
{    |d1 b1 c1|      |a1 d1 c1|      |a1 b1 d1|                    }
{    |d2 b2 c2|      |a2 d2 c2|      |a2 b2 d2|                    }
{    |d3 b3 c3|      |a3 d3 c3|      |a3 b3 d3|                    }
{ x = ----------   y = ----------   z = ----------                  }
{    |a1 b1 c1|      |a1 b1 c1|      |a1 b1 c1|                    }
{    |a2 b2 c2|      |a2 b2 c2|      |a2 b2 c2|                    }
{    |a3 b3 c3|      |a3 b3 c3|      |a3 b3 c3|                    }
{ виражаємо визначників третього порядку:                           }
```

```
{ e  := (a1*b2*c3+b1*c2*a3+c1*a2*b3-a3*b2*c1-b3*c2*a1-c3*a2*b1);    }
{ ex := (d1*b2*c3+b1*c2*d3+c1*d2*b3-d3*b2*c1-b3*c2*d1-c3*d2*b1);   }
{ ey := (a1*d2*c3+d1*c2*a3+c1*a2*d3-a3*d2*c1-d3*c2*a1-c3*a2*d1);    }
{ ez := (a1*b2*d3+b1*d2*a3+d1*a2*b3-a3*b2*d1-b3*d2*a1-d3*a2*b1);   }
{ x = ex/e                                                          }
{ y = ey/e                                                          }
{ z = ez/e                                                          }
{ ------------------------------------------------------------------- }
var a1,a2,a3,b1,b2,b3,c1,c2,c3,d1,d2,d3,x,y,z,e,ex,ey,ez:real;
begin
 writeln('введіть коефіцієнти рівняння:a1,b1,c1,d1,a2,b2,c2,d2,a3,b3,c3,d3');
 readln(a1,b1,c1,d1,a2,b2,c2,d2,a3,b3,c3,d3);
 e  := (a1*b2*c3+b1*c2*a3+c1*a2*b3-a3*b2*c1-b3*c2*a1-c3*a2*b1);
 ex := (d1*b2*c3+b1*c2*d3+c1*d2*b3-d3*b2*c1-b3*c2*d1-c3*d2*b1);
 ey := (a1*d2*c3+d1*c2*a3+c1*a2*d3-a3*d2*c1-d3*c2*a1-c3*a2*d1);
 ez := (a1*b2*d3+b1*d2*a3+d1*a2*b3-a3*b2*d1-b3*d2*a1-d3*a2*b1);
 if ( e=0 ) and ( (ex=0) or (ey=0) or (ez=0) ) then
    writeln('нескінченна безліч рішень')
 else if ( e<>0 ) and ( (ex=0) or (ey=0) or (ez=0) ) then
    writeln('немає рішень')
 else begin
    x:=ex/e; y:=ey/e; z:=ez/e;
    writeln('x = ', x); writeln('y = ', y); writeln('z = ', z);
 end;
end.
```

Приклад 15.1.38.

Програма, яка визначає кількість та суму цифр у записі будь-якого натурального числа, не перевершуючого maxint, а також виводить запис числа у зворотному порядку, опускаючи незначимі нулі на його початку. Подібну задачу для тризначного числа ми наводили раніше. Якщо кількість цифр числа невідома, то таку задачу можна вирішити, використовуючи цикл (краще repeat або while).

```
{ Виділення цифр числа }
var k,s,n: integer; a: longint;
begin
    write('Введіть натуральне число: ');  readln(n);
    k:=0; s:=0; a:=0;
    repeat
        k := k+l;
        s := s+n mod 10;
        a := a*10+n mod 10;
        n := n div 10
    until n = 0;
```

```
    writeln('Число цифр:  ', к, ', сума цифр:  ' ,s)
    writeln('Запис у зворотному порядку: ', a);
end.
```

15.2. Операції з масивами

Особливо хотілося б наивести декілька програм роботи з масивами, оскільки ці завдання особливо улюблені в багатьох підручниках. Не відступимо від традиції і ми. Основними завданнями, що пов'язані з масивами, є сортування елементів, пошук у відсортованому масиві, а також операції над матрицями (двовимірними масивами). Також будуть розглянуті деякі методики вирішення перерахованих завдань.

Приклад 15.2.1.

```
{ Друк найчастішого елементу, що зустрічається в масиві }
var a:array[1..10] of integer;
    i,j,m,p,n:integer;
begin
    writeln('введіть 10 елементів масиву');
    for i:=1 to 10 do readln( a[i]);
    m:=1;
    p:=1;
    for i:=1 to 10 do begin
       n:=0;
       for j:=1 to 10 do begin
          if a[i]=a[j] then inc(n);
       end;
       if n>m then begin
          m:=n;
         p:=i;
       end;
    end;
    writeln('найчастіший елемент, що зустрічається:',a[p]);
end.
```

Приклад 15.2.2.

```
{ Пошук максимального елементу в 1-мірному масиві }
var a:array[1..10] of integer;
    max:integer;
    i:integer;
begin
    writeln('введіть 10 елементів масиву');
    max:=-(MAXINT+1);
    for i:=1 to 10 do begin
       readln( a[i]);
       if max<a[i] then max:=a[i];
    end;
    writeln( 'Максимальний елемент масиву = ', max );
end.
```

Приклад 15.2.3.

```
{ Підрахунок суми елементів двовимірного масиву }
var a:array[1..10,1..2] of integer;
     s:longint;
     i,j:integer;
begin
    writeln('введіть 20 елементів масиву');
    s:=0;
    for i:=1 to 10 do begin
       for j:=1 to 2 do begin
         readln( a[i,j]);
         s:=s+a[i,j];
       end;
    end;
    writeln( 'Сума елементів масиву = ', s );
end.
```

Приклад 15.2.4.

```
{ Поміняти порядок елементів масиву на зворотний}
Program Revers_M;
uses  crt;
const  n = 10;   {кількість елементів масиву}
type  mas = array[1..n] of integer;
var  M: mas;
{Процедура заповнення  масиву випадковими числами}
procedure Init (n: Integer; var M: mas);
var  i: integer;
begin
   Randomize;
   for i := 1 to n do M[i]:= Random (255 );
end;
{ Процедура друку масиву }
procedure ShowArray(n: integer; var M: mas);
var  i: integer;
begin
     for i := 1 to n do Write(M[i]:4);
end;
{ Процедура реверсування масиву}
procedure Revers(n: Integer; var M: mas);
var   i, R: integer;
{ У цій процедурі застосовується простій алгоритм перестановки (поміняти
місцями) двох чисел }
{ з проміжною змінною R. Початковий стан: a   b;     R := a;  b := a;  a := R; }
{Ніколи не застосовуйте "незрозумілих" алгоритмів (нібито для економії
пам'яті) типу: }
{ a := a + b;   b := a – b;  a := a – b;  Отримаємо перевернуту послідовність: b  a }
{Або на мові Pascal: }
{    for i := 1 to  n div 2 do   begin }    {цикл до середини масиву, якщо цикл буде
від 1 до n}
```

```
{       M[i]:= M[i]+ M[n-i+1];    }  {то масив реверсує двічі, тобто залишиться без
змін}
{       M[n-i+1]:= M[i] –  M[n-i+1]; }
{       M[i]:= M[i] –  M[n-i+1]; }
{   end;  }
begin
    for i := 1 to  n div 2 do   begin   {цикл до середини масиву, якщо цикл буде від
1 до n}
        R := M[i];                      {то масив реверсує двічі, тобто залишиться без
змін}
        M[i]:= M[n-i+1];
        M[n-i+1]:= R;
   end;
end;

begin
    ClrScr;
   {Заповнюємо масив випадковими числами}
   Init (n, M);
   Writeln ( 'Масив до реверсування: ' ) ;
   ShowArray (n, M);
   Revers(n, M);    { реверсування масиву}
   WriteLn;
   Writeln ( 'Масив в зворотному порядку:');
   ShowArray (n, M);
   readkey;
end.
```

Приклад 15.2.5.

```
{ Видалення  негативних  елементів масиву }
Program Delete_M;
uses  crt;
const  n = 10;   {кількість елементів масиву}
type  mas = array[1..n] of integer;
var  M: mas;
     i, j, Raz: integer;
{Заповнення  масиву випадковими числами: додатними та від'ємними}
procedure Init (n: Integer; var M: mas);
var  i: integer;
begin
   Randomize;
   for i := 1 to n do M[i]:= Random (255 ) –  Random (255 );
end;
{ Друк масиву }
procedure ShowArray(n: integer; var M: mas);
var  i: integer;
begin
     for i := 1 to n do Write(M[i]:5);
end;

begin
```

```
   ClrScr;
   Init(n, M);
   Writeln ( ‘Початковий масив: ‘ ) ;
   ShowArray(n, M);
   Writeln;
   i := 1;
   Raz := n;  {запам’ятовуємо первинну кількість елементів масиву}
   while I <= Raz do  begin
      if M[i] <0  then  begin
         For j:=i to Raz-1 do     {зсуваємо ліворуч решту всіх елементів }
            M[j]:=M[j+1];        { на місце від’ємного елементу }
         Raz :=Raz-1;
      end
      else
         i:=i+1;              {переходимо до наступного елементу масиву}
    end;  {кінець циклу while}
    Writeln ( ‘Масив без від’ємних елементів: ‘ ) ;
    ShowArray(Raz, M);
    readkey;
end.
```

Приклад 15.2.6.

{Злиття 2-х масивів, з однакових елементів залишити один}
{та відсортувати результуючий масив за зростаниям елементів}

```
Program Slijanie;
const m=9; n=8;
var     A: array [1..m] of integer;
   B: array [1..n] of integer;
   C: array [1..m+n] of integer;
   D: array [1..m+n] of integer;
        i, j, до, kol: integer;
{Виведення елементів масиву на екран}
procedure ShowArray;
begin
    for i := 1 to m+n do Write(D[i]:3);
    WriteLn
end;
{Сортування масиву за збільшенням }
procedure SortArray;
var     buf: integer;
        i, j:integer;
begin
   for i := 1 to (m+n-1) do
      for j := 1 to (m+n-i) do
```

```
          if (D[j]> D[j+1]) then begin
             buf := D[j];
             D[j]:= D[j+1];
             D[j+1]:= buf;
          end;
end;

begin
  {Заповнюємо масиви випадковими числами}
  Randomize;
  for i := 1 to m do A[i]:= Random (99);
  for i := 1 to n do B[i]:= Random (99);
  {Злиття масивів A і B}
  for i:= 1 to m do
           D[i]:=A[i];
  for j:= 1 to n do
     D[j+m]:=B[j];
  WriteLn('Масив після злиття A і B:');
  ShowArray;
  ReadLn;
  SortArray; {сортування за збільшенням}
  WriteLn('Масив після сортування:');
  ShowArray;

  {Знаходження однакових елементів}
  C[1]:=D[1];
  kol:= 1;
  for k:= 2 to m+n do begin
       if C[kol] <> D[k] then begin
             kol:= kol+1;
        C[kol]:=D[k]
      end;
  end;
  WriteLn('Масив після виключення однакових елементів:');
  for j:= 1 to kol do
       write (C[j]:3);
  writeLn;
  WriteLn ('Кількість елементов масиву = ',kol);
  ReadLn
end.
```

Приклад 15.2.7.

{Всі великі символи латинського алфавіту масиву замінити на малі та навпаки}

```
uses  crt;
const  n = 5;   {кількість елементів масиву}
type  mas= array[1..n] of char;
var  M: mas;
      i, j: integer;
procedure init(n:integer; var M:mas);
var
  i: integer;
begin
    Writeln ( ‘Введіть  ‘, n ‘ елементів (букв) масиву підряд без пробілів та
натисніть Enter:’ ) ;
    for i:=1 to n do
    read(M[i]);
end;
{ Друк масиву }
procedure ShowArray(n: integer; var M: mas);
var  i: integer;
begin
      for i := 1 to n do Write(M[i] ‘ ‘);
end;
begin
  clrscr;
  init(n,M);
  Writeln ( ‘Масив до перетворення: ‘ ) ;
  ShowArray (n, M);
  Writeln;
  j:=ord(‘a’) -ord(‘A’);
  for i:=1 to n do
    if (M[i] >= ‘A’) and (M[i] <= ‘Z’)
         then
           M[i]:= chr(ord(M[i])+j)
         else
           if (M[i] >= ‘a’) and (M[i] <= ‘z’)
             then
               M[i]:= chr(ord(M[i]) -j);
 Writeln ( ‘Масив після перетворення: ‘ ) ;
  ShowArray (n, M);
  readkey;
end.
```

Приклад 15.2.8.

```
{Знайти суму двох максимальних елементів масиву за один перегляд.}
{Ось один з прикладів.}
Program MaxSum2;
uses crt;
```

```
const n=10;
type mas = array[1..n] of integer;
Var A: mas;
   i, max1, max2: integer;
{Заповнення матриці випадковими числами}
Procedure RandMas(var A: mas);
Var k:integer;
begin
    Randomize;
    for k:= 1 to n do
      A[k] := Random(255);
end;
{Виведення масиву на екран}
procedure ShowArray(var A: mas);
var k: integer;
begin
   for k := 1 to n do begin
     Write(A[k]:4);
   end;
   Writeln( ‘  ‘ ) ;
end;
{Пошук двох максимальних елементів массиву}
procedure MaxElem(var A: mas; var max1,max2: integer);
begin
  For i:=1 to n do begin
     If A[i] > max1 then begin
       max2:=max1;
       max1:=A[i]
     end
     else if A[i] > max2 then
       max2:=A[i];
  end;
end;
Begin
  clrscr;
  RandMas(A);
  ShowArray(A);
  MaxElem(A,max1,max2);
  writeln(‘Maxsum = ‘,max1,’+’,max2,’ = ‘,max1+max2);
  readln;
End.
```

Приклад 15.2.9.

{Деяке число міститься у кожному з трьох цілочисельних неубутних масивів x[1] <= ... <= x[p], y[1] <= ... <= y[q], z[1] <= ... <= z[r]. Знайти одне з таких чисел.
Число дій має бути порядку p + q + r.
(із книги: Д.Грис. Наука программирования. М.:Мир, 1984.) }
{Наведемо основний фрагмент алгоритму}

```
p1:=1; q1=1; r1:=1;
{ x[p1]..x[p], y[q1]..y[q], z[r1]..z[r]    містять спільний елемент}
while not ((x[p1]=y[q1]) and (y[q1]=z[r1])) do begin
      if x[p1]<y[q1] then begin
         p1:=p1+1;
      end else if y[q1]<z[r1] then begin
         q1:=q1+1;
      end else if z[r1]<x[p1] then begin
         r1:=r1+1;
      end else begin
         { такого не може бути }
      end;
end;
{x[p1] = y[q1] = z[r1]}
writeln (x[p1]);
```

Приклад 15.2.10.

Решето Ератосфена – простий алгоритм знаходження всіх простих чисел до деякого цілого числа n. Він був створений старогрецьким математиком Ератосфеном. Розглянемо приклад для n = 20. Запишемо натуральні числа, починаючи від 2 до 20 у ряд:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Перше число у списку 2 – просте.
Пройдемо по ряду чисел, викреслюючи усі числа кратні 2 (починаючи з $2^2 = 4$, замінючи на 0):

2 3 0 5 0 7 0 9 0 11 0 13 0 15 0 17 0 19 0

Наступне невикреслене число 3 – просте.
Пройдемо по ряду чисел, викреслюючи всі числа, які кратні 3 (починаючи з $3^2 = 9$):

2 3 0 5 0 7 0 0 0 11 0 13 0 0 0 17 0 19 0

Наступне невикреслене число 5 – просте.
Пройдемо по ряду чисел, викреслюючи всі числа кратні 5 (починаючи з $5^2 = 25$). Тобто, викреслюємо кратні числа для всіх простих чисел p, для яких $p^2 <= n$. В результаті всі складені числа будуть викреслені, а невикресленими залишаться всі прості числа.
Для n = 20 вже після викреслювання кратних числу 3 всі складені числа будуть викресленими. Складність алгоритму Ератосфена O(NlnlnN).

```
Program ERATOSF;
Uses Crt;
Const          limit = 1000;
Var    i, j, n, prost: integer;
       p: array[1..limit] of integer;
Begin
  Clrscr;
  write('Введіть ціле число <= ', limit, ' : ');
  readln(n);
  for i:=1 to n do    {Заповнюємо масив числами 1 по n.}
    p[i] := i;
  prost:=1; {Підготовка параметра для знаходження простих чисел}
  for i:=1 to n do begin {Цикл руху по масиву чисел}
     if p[i]=0 then
        continue      {Щоб зайвий раз не заходити в цикл}
     else
        inc(prost);    {Збільшення параметра}
     if prost*prost > n then break;       {p^2 > n –  припиняємо роботу алгоритму}
     j:=i;             {Підготовка параметра вкладеного циклу}
     while j<=n do begin {Вкладений цикл для визначення простих чисел}
       if (p[j] mod prost=0)    {Якщо залишок від ділення елементу на просте
число = 0..}
        and(p[j]<>0)                {..і цей елемент не дорівнює 0..}
        and(p[j]>prost) then        {..і цей елемент більше параметра prost..}
        p[j]:=0;                {..це СКЛАДЕНЕ число, вилучаємо його з масиву }
       inc(j);                      {інакше це просте число, та подальший пошук}
    end; {КОНЕЦ while}
  end;   {КОНЕЦ for}
  writeln('Прості числа у диапазоні [1,n] :');
  for i:=1 to n do    {Друкуємо масив з ненульовими числами.}
    if p[i] <> 0 then
        write(p[i], '  ');
  ReadKey;
End.
```

Приклад 15.2.11.

```
{Побудувати "трикутник Pascal", використовуючи масив}
uses crt;
var a: array[1..200] of  integer;
    n, i, j, k, h: integer;    {n - кількість рядків,  k - кількість пропусків}
begin
    clrscr;
    write('n='); readln(n);
    h:=4;                      {індекс для масиву}
```

```
    a[1]:=1; a[2]:=1; a[3]:=1;     {початкові дані}
    for i:=3 to n do  begin
        a[h]:=1;
        for j:=2 to i do begin
            h:=h+1;
            if j=i then a[h]:=1
          else  a[h]:= a[h-i]+a[h-i+1];
       end;
       h:=h+1;
   end;
   if n < 10 then k:= 4 else k:= 6;
   h:=1;
   for i:=1 to n do   begin
        write(‘ ‘:(n-i+1+((k div 2)-1)*(n-i)));
        for j:= 1 to i do begin
             write(a[h]:k);
              h:=h+1;
        end;
        writeln;        {Перехід на новий рядок}
   end;
   readln
end.
```

15.3. Операції з матрицями

Приклад 15.3.1.

Операції над матрицями. Розглянемо низку прикладів, в яких обробляються двовимірні масиви або матриці. У першому прикладі заповнимо квадратну матрицю випадковими цілими числами, а потім запишемо середнє арифметичне елементів рядків у ті елементи матриці, які розташовані на центральній діагоналі.

```
program Matrix_1;
uses Crt;
var
        A: array [1..5,1..5] of real;
        i, j: byte;
        sum: real;
{Виведення елементів масиву}
procedure ShowArray;
begin
        for i := 1 to 5 do begin
           for j := 1 to 5 do Write(A[i,j]:7:1);
           Writeln( ‘  ‘ );
        end;
end;
begin
```

```
    ClrScr;
    {Заповнюємо матрицю випадковими числами}
    Randomize;
    for  i := 1 to 5 do
        for  j := 1 to 5 do A[i,j]:= Random(255);
    Writeln('Початкова матриця:');
    ShowArray;
    {Обраховуємо середнє арифметичне елементів рядків}
    for  i := 1 to 5 do  begin
        sum := 0;   {Сума елементів рядка}
        for  j := 1 to 5 do sum := sum + A[i,j];
        A[i,i]:= sum/5;
    end;
    Writeln('Отримана матриця:');
    ShowArray;
    ReadKey;
end.
```

Приклад 15.3.2.

Операції над матрицями. У другому прикладі реалізуємо сортування всіх елементів матриці, які розташовані зліва від головної діагоналі, за збільшенням, а розташованих справа – за зменшенням.

```
program Matrix_2;
uses Crt;
var
        A: array [1..5,1..5] of byte;
        B: array[1..10] of byte;    {Тимчасовий масив}
        i, j: byte;
{Виведення елементів масиву}
procedure ShowArray;
begin
        for i := 1 to 5 do begin
            for j := 1 to 5 do Write(A[i,j]:3);
            Writeln( ' ' );
        end;
end;
{Сортування масиву. Якщо ByAsc = True, то –  за збільшенням.
Якщо ByAsc = False, то –  за зменшенням.}
procedure SortArray(ByAsc: boolean);
var
        buf: byte;
begin
        for i := 1 to 9 do
```

```
            for j := i+1 to 10 do
                if (ByAsc and (B[i]> B[j])) or
                   (not ByAsc and (B[i]< B[j]))  then begin
                   buf := B[i];
                   B[i]:= B[j];
                   B[j]:= buf;
                end;
end;
{Процедура копіювання елементів з одного масиву в інший.
Якщо FromAtoB = True, то копіюємо з A у B
Якщо FromAtoB = False, то копіюємо з B в A}
procedure CopyArrays (Lefthalf, FromAtoB: boolean);
var
        iBegin, iEnd, jBegin, jEnd, з: byte;
begin
   {Визначаємо початкове та кінцеве значення лічильників циклів}
    if LeftHalf then  begin
        {Для лівої частини матриці}
        iBegin := 2;   iEnd := 5;
        jBegin := 1;   jEnd := 1;
    end
    else  begin
   {Для правої частини матриці}
        iBegin := 1;   iEnd := 4;
        jBegin := 2;   jEnd := 5;
    end;
    {Копіюємо елементи з масиву у масив}
    з := 1;   {Лічильник тимчасового масиву}
    for i := iBegin to iEnd do  begin
        for j := jBegin to jEnd do  begin
            if FromAtoB  then
                B[c]:= A[i,j]
            else A[i,j]:= B[c];
            Inc(c);
        end;
        if LeftHalf  then
            Inc(jEnd)
        else Inc(jBegin);
    end;
end;
{Якщо LeftHalf = True, то сортуємо ліву частину, інакше –  праву частину}
procedure SortHalf(LeftHalf: boolean);
begin
    CopyArrays(LeftHalf,True);   {Формуємо тимчасовий масив}
```

```
    SortArray(LeftHalf);   {Сортуємо тимчасовий масив}
    {Переписуємо значення у матрицю}
    CopyArrays(LeftHalf,False);
end;
begin
    ClrScr;
    {Заповнюємо матрицю випадковими числами}
    Randomize;
    for i := 1 to 5 do
        for j := 1 to 5 do A[i,j]:= Random(99);
    Writeln('Початкова матриця:');
    ShowArray;
    SortHalf(True); {Сортуємо ліву частину}
    SortHalf(False); {Сортуємо праву частину}
    Writeln('Отримана відсортована матриця:');
    ShowArray;
    ReadKey;
end.
```

Приклад 15.3.3.

Операції над матрицями. У третьому прикладі виконаємо дзеркальне відображення (транспонування) матриці щодо головної діагоналі. При транспонуванні елементи матриці переставляються таким чином, що рядки вихідної матриці стають стовпчиками транспонованої матриці. При цьому елементи, розташовані на головній діагоналі вихідної та транспонованої матриць, однакові.

```
program Matrix_3;
uses Crt;
var
        A: array [1..5,1..5] of byte;
        i, j: byte;
{Виведення елементів масиву}
procedure ShowArray;
begin
        for i := 1 to 5 do begin
            for j := 1 to 5 do Write(A[i,j]:5);
            Writeln( ' ' );
        end;
end;
procedure TranspArray;
var
        jEnd, buf: byte;
begin
        jEnd := 1;
```

```
        for i := 2 to 5 do  begin
            for j := 1 to jEnd do  begin
                buf := A [ i, j ] ;
                A[i, j]:= A [ j, i ] ;
                A [ j, i ] := buf;
            end;
            Inc(jEnd);
        end;
end;

begin
        ClrScr;
        {Заповнюємо матрицю випадковими числами}
        Randomize;
        for i := 1 to 5 do
             for j := 1 to 5 do A[i,j]:= Random(255);
        Writeln('Початкова матриця:');
        ShowArray;
        TranspArray;   {Дзеркальне відображення матриці}
        Writeln('Отримана дзеркальна матриця:');
        ShowArray;
        ReadKey;
end.
```

Приклад 15.3.4.

```
{Відсортувати за збільшенням головну діагональ матриці}
{та всі діагоналі нижчі та вищі за головну діагональ}
Program SORT_DIAG;
const m=5; n=5;
var      A: array [1..m, 1..n] of integer;
        B: array [1..m] of integer;
        i, j, до, i1: integer;
{Виведення елементів масиву на екран}
procedure ShowArray;
begin
  for i := 1 to m do begin
            for j:= 1 to n do
               Write(A[i,j]:3);
     WriteLn
  end;
end;
{Сортування масиву за збільшенням }
procedure SortArray(до: integer);
var     buf: integer;
```

```
        i, j:integer;
begin
   for i := 1 to (k-1) do
      for j := 1 to (k-i) do
         if (B[j]> B[j+1]) then begin
            buf := B[j];
            B[j]:= B[j+1];
            B[j+1]:= buf;
         end;
end;
begin
  {Заповнюємо матрицю випадковими числами}
  Randomize;
  for i := 1 to m do
    for j := 1 to n do A[i,j]:= Random (99);
  WriteLn('Матриця після заповнення:');
  ShowArray;

  {Сортування лівої частини матриці}
  for i := 1 to m do
    for j := 1 to n do
          if i = j then B[i]:= A[i,i];
  if m > n then
       i:= n
  else i:= m;
  SortArray(i); {сортування за збільшенням}
  for i := 1 to m do
       for j := 1 to n do
            if i = j then A[i,i]:= B[i];

  {Сортування лівої частини матриці}
  for i := 2 to m-1 do begin
       k:= 0;
            for j:= 1 to m-i+1 do begin
              k:= K+1;
                B[k]:= A[i+j-1,j]
            end;
      SortArray(k); {сортування за збільшенням}
       k:= 0;
            for j:= 1 to m-i+1 do begin
              k:= K+1;
                A[i+j-1,j]:= B[k];
      end;
   end;
```

```
  {Сортування правої частини матриці}
  for j := 2 to n-1 do begin
       k:= 0;
            for i:= 1 to n-j+1 do begin
              k:= K+1;
                B[k]:= A[i,i+j-1]
            end;
      SortArray(k); {сортування за збільшенням}
       k:= 0;
            for i:= 1 to n-j+1 do begin
               k:= K+1;
               A[i,i+j-1]:= B[k];
       end;
  end;
  WriteLn('Матриця після сортування:');
  ShowArray;
  ReadLn
end.
```

Приклад 15.3.5. Добуток матриць.

Створити добуток C=AxB двох дійсних матриць A і B розміром 20 x 20. Нагадаємо, що елемент матриці C=AxB визначається як

$$Cij = \sum_{k=1}^{20} aik * bkj$$

Таким чином, для кожної з пар (i, j) треба обрахувати суму Cij = SUMMA(i, j, 20), де

*Cij = SUMMA(i, j, k) ={0, якщо k = 0; SUMMA(i, j, k – 1)+aik*bkj, якщо k > 0}*

Отримуємо наступну програму:

```
Program IncreaseMat;
   const M = 20;
   var A,B : array [1..M, 1..M] of real; S:real; I,J,K : integer;
begin
   for I := 1 to M do for J := 1 to M do read(A[I,J]) end end;
   for I := 1 to M do for J := 1 to M do read(B[I,J]) end end;
   for I := 1 to M do
      for J := 1 to M do
         S := 0;  {S = SUMMA(I,J,0)}
         for K := 1 to M do
              {S = SUMMA(I,J,K-1)}
              S := S + A [I,K]*B [K,J]            {S = SUMMA(I,J,K)}
         end;
         {S = SUMMA (I,J,M)}
         write(S:4:2)
```

```
      end;  writeLn
   end;
   ReadLn
end.
```

Приклад 15.3.6.

```
{Побудувати “трикутник Pascal”}
uses crt;
var a: array[1..20,1..20] of integer;
    n, i, j, k: integer;       {n - кількість рядків,  k - кількість пробілів}
begin
   clrscr;
   write(‘n=‘); readln(n);
   a[1,1]:=1; a[2,1]:=1; a[2,2]:=1;       {початкові дані}
   for i:=3 to n do  begin
        a[i,1]:=1;
        for j:=2 to i do begin
            if j=i then a[i,j]:=1
        else  a[i,j]:= a[i-1,j-1]+a[i-1,j];
        end;
   end;
   if n < 10 then k:= 4 else k:= 6;
   for i:=1 to n do   begin
        write(‘ ‘:(n-i+1+((k div 2)-1)*(n-i)));
        for j:= 1 to i do
        write(a[i,j]:k);
        writeln;       {Перехід на новий рядок}
   end;
   readln
end.
```

Приклад 15.3.7.

Перевірити, чи є задана цілочисельна матриця A(N, N) "**магічним квадратом**" (це означає, що суми чисел в усіх її рядках, усіх стовпчиках та двох діагоналях однакові). Наприклад, матриця A – “магічний квадрат”, а B – “не магічний квадрат”:

Матриця **A**	1 2 3 4 2 0 1 2 3	1 2 3 4 5 2 1 2 3 4 3 0 1 2 3 4	‘‘Магічний квадрат’’
Матриця **B**	2 1 1 2		‘‘Не магічний квадрат

```
Program MagicSquare;
Uses Crt;
Var A : Array [1..20, 1..20] of Integer;
    i, j, N : Integer;
    S_d, S : Integer; {S_d –  сума-еталон, S –  поточна сума}
    Flag : Boolean;
{ Процедура введення-виведення матриці }
Procedure InputOutput;
Begin
  ClrScr;
  Write('Кількість рядків та стовпчиків: ');
  ReadLn(N);
  For i := 1 to N do
   For j := 1 to N do begin
            Write('A[' , i , ', ' , j , '] = ');
            ReadLn(A[i, j])
   end;
  ClrScr;
  WriteLn('Початкова матриця :'); WriteLn;
  For i := 1 to N do begin
     For j := 1 to N do Write(A[i, j] : 5);
     WriteLn
  end;
  WriteLn
End; {InputOutput}
{ Процедура визначення, чи є квадрат "магічним"}
Procedure Magic(Var Flag : Boolean);
 Begin   {обрахунок еталонної суми елементів головної діагоналі}
  S_d:=0;
  For i := 1 to N do S_d := S_d + A[i,i];
  Flag:=TRUE; i:=1;
  While (i<=N) and Flag do begin { обрахунок сум елементів рядків}
     S:=0;
     For j := 1 to N do S := S+A[i, j];
     If S<>S_d then Flag := FALSE else i:=i+1
  end;
  j:=1;
  While (j<=N) and Flag do begin { обрахунок сум елементів стовчиків}
     S:=0;
     For i := 1 to N do S:=S+A[i, j];
     If S<>S_d then Flag := FALSE else j := j+1
  end;
  If Flag then begin
     S:=0; { обрахунок суми елементів побічної діагоналі}
```

```
    For i := 1 to N do S := S+A[i, N+1-i];
    If S<>S_d then Flag := FALSE;
  end;
End;
BEGIN
  InputOutput;
  Magic(Flag);
  If Flag then WriteLn('Це магичний квадрат.')
        else WriteLn('Це не магичний квадрат.');
  ReadLn
END.
```

15.4. Обробка тексту

IBM-сумісні комп'ютери обробляють 256 різних символів, кожен з яких кодується одним байтом. Відповідність символів та байтів задається ***таблицею кодування***, в якій для кожного символу вказується відповідний байт.

Коди 0...127 (кодування ASCII)

	000	016	032	048	064	080	096	112	
00		◆		0	@	P	`	p	00
01		◆	!	1	A	Q	a	q	01
02		↕	"	2	B	R	b	r	02
03	♥	‖	#	3	C	S	c	s	03
04	◆	¶	$	4	D	T	d	t	04
05	♣	§	%	5	E	U	e	u	05
06	♠		&	6	F	V	f	v	06
07	•		'	7	G	W	g	w	07
08		↑	(	8	H	X	h	x	08
09	°	↓	)	9	I	Y	i	y	09
10		→	*	:	J	Z	j	z	10
11		←	+	;	K	[	k	{	11
12		⌙	,	<	L	\	l	|	12
13		•	-	=	M	]	m	}	13
14		•	.	>	N	^	n	~	14
15	Ц	◆	/	?	O	_	o	⌂	15

Коди 128...255 (модифікований альтернативний варіант)

	128	144	160	176	192	208	224	240	
00	А	Р	а	░	└	╨	р	≡	00
01	Б	С	б	▒	┴	╤	с	±	01
02	В	Т	в	▓	┬	╒	т	≥	02
03	Г	У	г	|	├	╓	у	≤	03
04	Д	Ф	д	┤	─	╙	ф	⌠	04
05	Е	Х	е	╡	┼	╒	х	⌡	05
06	Ж	Ц	ж	╢	╞	╫	ц	÷	06
07	З	Ч	з	╖	╟	╪	ч	≈	07
08	И	Ш	и	╕	╚	╪	ш	°	08
09	Й	Щ	й	╣	╔	┘	щ	∙	09
10	К	Ъ	к	‖	╩	┌	ъ	·	10
11	Л	Ы	л	╗	╦	▉	ы	√	11
12	М	Ь	м	╝	╠	▄	ь	ⁿ	12
13	Н	Э	н	╜	¥	▌	э	²	13
14	О	Ю	о	╛	╪	▐	ю	■	14
15	П	Я	п	┐	╧	▀	я		15

Символи з кодами від 0 до 127 побудовані за **стандартом ASCII** (**American Standard Code** for **Information Interchange** – Американський стандартний код обміну інформацією, читається "аски"). Друга половина таблиці (коди 128 ... 255) у нашій країні містить російські букви (кирилицю),

українські букви, які неспівпадають із зображенням російських букв (і, ї, є), та символи псевдографіки.

Для того щоб визначити по цих таблицях код того або іншого символу, потрібно **скласти номер рядка з номером стовпчика**, в яких він розташований. Так, код цифри 2 дорівнює 02+048 = 050.

Приклад 15.4.1.

```
{ Виведення таблиці кодування символів }
var     ch:char; { символ }
        dec:integer; { десятковий код символу }
        i,j:integer;
begin
  dec:=0;
  for i:=0 to 15 do begin   {шістьнадцять рядків}
        dec := i;
{ щоб отримати таблицю кодування для символів з кодами 128-255 }
{ цю інструкцію треба замінити на dec:=i+128;}
        for j:=1 to 8 do begin      { вісім колонок}
           if (dec < 7) or ((dec >= 14) and (dec <> 26)) then
                write(dec:4,'-  ',chr(dec):1,chr(179))
           else { символи CR,LF,TAB, «пробіл –  26» не відображаються }
                write(dec:4,'-   ',chr(179));
           dec:=dec+16;
        end;
        writeln; { перехід до нового рядка екрану }
  end;
  readln;
end.
```

Приклад 15.4.2.

```
{ Тип даних String }
{Програма зчитує список символьних рядків та друкує їх у порядку за
абеткою}
program Alpha;
Const   LineSize = 5;
Var     Line: array[1..LineSize] of String;
        Count, LineEntry: Integer;
        LowString: 1..LineSize;
Begin
  { Читання списку }
  WriteLn('Введіть ',LineSize', рядків тексту, які потрібно впорядкувати');
  WriteLn('При кожному введенні задається значення одного рядка');
  WriteLn;
  For LineEntry := 1 to LineSize do
```

```
      ReadLn(Line[LineEntry]);
  WriteLn;
{ Друк списку }
  For LineEntry := 1 to LineSize do
    Begin
       LowString := 1;
       While Line[LowString]= ‘‘ do
          LowString := LowString + 1;
       For Count := 2+(LowString-1) to LineSize do
        If (Line[LowString] <> ‘‘) and (Line[Count]< Line[LowString]) and
        (Line[Count] <> ‘‘)     then   LowString := Count;
       If Line[LowString] <> ‘‘ then
         WriteLn(Line[LowString]);
       Line[LowString]:= ‘‘
    end;
  ReadLn
End.
```

Приклад 15.4.3.

{ Вилучення коротких слів (<= 2) з вхідного файлу Old_File.txt }
{ та запис нових рядків у файл New_File.txt }

```
Program Udal;
uses crt;
Var Old_file, new_file: text;
     Str_old, Str_new: string;
      Flag, Poz: integer;
Begin
   assign(Old_file,’Old.txt’);
   assign(New_file,’New.txt’);
   reset(Old_file);
   {$I-}   {Вимикаємо автоконтроль}
   rewrite(new_file);
   {$I+}   {Вмикаємо автоконтроль}
   While not EOF (Old_file) do begin
         readln(Old_file,Str_old);
         Str_New := ‘‘;
         Flag := 1;
         While Flag <> 0 do begin        {поки не кінець рядка}
            repeat     {перевірка на пробіл спочатку рядки}
                 Poz := Pos(‘ ‘,Str_old);
                if Poz = 1 then
                    Str_old := Copy(Str_old,2,256);    { вилучаємо пробіл спочатку
рядка}
            until Poz <> 1;
```

```
            if poz = 0 then poz := length(Str_old)+1;          {немає більше пробілів}
            if length(Copy(Str_old,1,Poz-1))> 2 then   {додаємо слово в Str_New}
             Str_new := Concat(Str_New, Copy(Str_Old,1,Poz));
             if Poz = length(Str_old)+1 then
                Flag := 0;                    {останнє слово}
             Str_old := Copy(Str_old,Poz+1,256);  {вилучаємо слово з Str_Old}
          end;
          writeln(New_file,Str_new);    {запис обрабленого рядка у новий текстовий
файл}
  end;
  Close(New_file);
  {readln}
End.
```

Приклад 15.4.4.

```
{ Обробка тексту: Підрахунок кількості слів у тексті }
{ На вході –  текст, на виході –  кількість слів в тексті                }
const Alpha : set of char = ['A'..'Z','A'..'П','Р'..'Я','a'..'z','a'..'п','р'..'я'];
var s: string;
     i: integer;
     wc :integer;
begin
    writeln('Введіть текст');
    readln(s);
    i:=1;
    wc:=0;
    Repeat
      while NOT(s[i] in Alpha) and (i <= length(s)) do inc(i);
      if (i<=length(s)) then inc(wc);
      while (s[i] in Alpha) and (i <= length(s)) do inc(i);
    Until (I > length(s));
    writeln('Кількість слів у цьому тексті = ',wc);
end.
```

Приклад 15.4.5.

```
{ Обробка тексту: Виділення слів з тексту }
{ На вході –  текст, на виході –  список слів      }
Const  Alpha : set of char=['A'..'Z','A'..'П','Р'..'Я','a'..'z','a'..'п','р'..'я'];
var     s,t:string;
        i:integer;
begin
    writeln('Введіть текст'); readln(s);
    writeln('Список слів в тексті:');
    i:=1;
```

```
    Repeat
      while NOT(s[i] in Alpha) and (i<=length(s)) do inc(i);
      t:=‘‘;
      while (s[i] in Alpha) and (i<=length(s)) do begin
           t:=t+s[i];
           inc(i);
      end;
      if length(t) <>0 then writeln(t);
    Until (i>length(s));
     Readln;
end.
```

Приклад 15.4.6.

```
(* Обробка тексту: вилучення з тексту коментарів типу {...}          *)
{ На вході –  текст з коментарями, на виході –  текст без коментарів }
var s,r:string;
   state,i:integer;
begin
    writeln(‘Введіть будь-який текст з коментарями’); readln(s);
    r:=‘‘; state:=0; {нормальний стан}
    for i:=1 to length(s) do begin
      case s[i] of
        ‘{‘: if state=0 then state:=1;  {тепер ми всередині коментаря}
        ‘}’: if state=1 then state:=0   {тепер ми вийшли з коментаря}
            else r:=r+s[i];           {ми не в коментарі}
        else if state=0 then r:=r+s[i]; {ми не в коментарі}
      end;
    end;
    writeln(‘новий текст:’); writeln(r);
    readln;
end.
```

Приклад 15.4.7.

Перевірити, чи є в лінійному записі заданої математичної формули баланс відкриваючих та закриваючих дужок.

```
Program Balance;
 Uses Crt;
 Var S:  String;
          Dlina, Flag, i : Integer;
BEGIN
   ClrScr;
   WriteLn(‘Введіть лінійний запис математичної формули:’);
   ReadLn(S);
    i:= 1; Flag:= 0; Dlina:= Length(S);
   While (Flag>=0) and (i<=Dlina) do begin
      If S[i] = ‘(‘ then Flag:= Flag + 1;
      If S[i] = ‘)’ then Flag:= Flag –  1;
      i:= i+1
   end;
   If Flag = 0 then
      Write(‘Є баланс ‘)
```

```
   else
     Write(‘Немає баланса ‘);
   WriteLn(‘відкриваючих та закриваючих дужок’);
   ReadLn
END.
```

Для перевірки програми можна ввести наступну систему тестів:

Номер тесту	Перевіряє мий випадок	Дані	Результат
1	При перегляді лінійного запису зліва направо першою зустрічається закриваюча дужка	‘‘a)b+1(‘‘	‘‘Немає балансу’’
2	Першою зустрічається відкриваюча дужка, але число відкриваючих і закриваючих дужок не збігається	‘‘(a+b))’’	‘‘Немає балансу’’
3	Є баланс дужок ‘‘(a+b/(c*d))’’	‘‘(a+b/(c*d))’’	‘‘Є баланс’’

Приклад 15.4.8.

Визначити, чи є задане слово "перевертишем" (слово називається "перевертишем", якщо збігається з собою після перевертання).

```
Program TurnOver;
  Uses Crt;
  Var Slovo    : String;
      Dlina, i : Integer;
      Flag     : Boolean;
BEGIN
  ClrScr;
  Write(‘Введите слово : ‘);  ReadLn(Slovo);
  Dlina:= Length(Slovo);
    { Порівнюються пари букв: перша буква з останньою, }
    { друга буква з передостанньою і так далі }
  i:=1;  Flag := TRUE;
  While (i <= Dlina/2) and Flag do  begin
       Flag := (Slovo[i]=Slovo[Dlina-i+1]);
       i := i+1
   end;
  WriteLn; Write( ‘Відповідь : слово ‘, Slovo);
  If Flag then WriteLn(‘ – перевертиш. ‘)
  else WriteLn(‘ – не перевертиш’);
  ReadLn
END.
```

Приклад 15.4.9.

Задану послідовність слів перевпорядкувати у порядку за абеткою (тобто виконати лексикографічне впорядкування).

```
Program LexOrder;
  Uses Crt;
  Var    Words       : Array[1..10] of String; {масив слів}
           Tmp         : String;      {Tmp – допоміжна змінна }
           i, j, NWords : Integer;     {NWords – кількість слів }
BEGIN
  ClrScr;
  Write('Кількість слів у тексті – ');
  ReadLn(NWords);
  For i := 1 to NWords do  begin
        Write(i, '-е слово : ');
         ReadLn(Words[i])
  end;
  For i := 1 to NWords-1 do { лексикографічне впорядкування слів }
     For j := i+1 to NWords do
         If Words[i]>Words[j] then  begin
            Tmp := Words[i]; Words[i]:=Words[j]; Words[j]:=Tmp
         end;
  WriteLn;  WriteLn('О т в е т');
  WriteLn('Лексикографічно впорядкований масив слів:');
  For i := 1 to NWords do Write(Words[i], ' ');
  WriteLn; ReadLn
END.
```

Приклад 15.4.10.

Як було сказано раніше, текстові файли зберігають інформацію у символьному вигляді і ця інформація розбита на рядки. В процесі введення (виведення) у текстовий файл можливе перетворення інформації несимвольного типу (наприклад, чисел) у символи. З файлом на магнітному диску пов'язується файлова змінна, тип якої оголошується як text.

Програма Write_text створює текстовий файл, в який може бути записана будь-яка текстова інформація.

```
program Write_text;
var
        FIL : text; {Опис файлової змінної}
        s : String;
begin
   Assign(FIL, 'ASPUSHKN.TXT');
```

{Встановлюється зв'язок файлової змінної FIL з файлом ASPUSHKN.TXT. ASPUSHKN.TXT – ім'я файлу на диску, воно більше ніде у програмі не з'явиться, його заміщатиме ім'я FIL}

```
   ReWrite(FIL);
{Тепер файл ASPUSHKN.TXT відкритий для запису. Якщо у ньому вже було
щось записано, то все буде знищено}
   WriteLn('Введіть вірші по рядках');
   ReadLn(S);
   while S <> '' do
   begin
       WriteLn(FIL, s);
       ReadLn(S)
   end;
   Close(FIL); {Файл закритий}
end.
```

У цій програмі спочатку вводиться змінна файлового типу FILE. Потім за допомогою процедури Assign встановлюється зв'язок між цією змінною і файлом ASPUSHKN.TXT, який створюється (або вже створений) на диску комп'ютера. Після встановлення зв'язку зовнішнього файлу з файловою змінною зовнішній файл треба відкрити для запису або читання. При відкритті файлу виконуються необхідні системні операції, що готують файл до запису або зчитування інформації. У нашому прикладі процедура ReWrite виконує всі підготовчі дії, які необхідні для запису у файл. Кожен рядок, що набраний на клавіатурі, зчитується у змінну S строкового типу, а потім за допомогою файлової змінної FILE передається у файл ASPUSHKN.TXT.

При натисненні клавіші Enter вводяться символи #13 і #10, що завершують рядок. Запис у файл здійснюватиметься доти, доки не буде введений порожній рядок. Процедура Close "закриває" файл. Після закриття файлу пов'язаний з ним зовнішній файл оновлюється, потім файлова змінна може бути пов'язана з іншим зовнішнім файлом.

Запустимо програму, наберемо знаменитий чотиривірш, натискаючи в кінці кожного рядка клавішу Enter. Введення завершимо подвійним натисненням Enter, при якому в програму буде переданий пустий рядок. Після завершення програми на диску з'явиться файл.

Приклад 15.4.11.

Читання з текстового файлу здійснюється процедурами ReadLn або Read. Якщо перед списком введення коштує ім'я файлової змінної, то дані читаються не з клавіатури, а з файлу, який пов'язаний з цією змінною:

ReadLn(FIL,список_ввода);

Черговий рядок створеного нами раніше на диску файлу ASPUSHKN.TXT зчитується у допоміжну змінну списку введення через файлову змінну FIL. От як це робиться у програмі Read_luk, що зчитує файл ASPUSHKN.TXT і що виводить текст на екран:

```
program Read_luk;
var
    FIL : text;
```

```
    S : String; {s –  ім’я допоміжної змінної списку введення}
begin
    Assign(FIL, ‘ASPUSHKN.TXT’);
    Reset(FIL); {Відкрили файл для читання}
    while not EOF(FIL) do
    {Читання продовжується, поки не буде досягнутий кінець файлу}
    begin
        ReadLn(FIL, S);
        {Читання з файлу ASPUSHKN.TXT у S построковий}
        WriteLn(S); {Виведення зчитаного на екран}
    end;
    Close(FIL); {Закрили файл}
end.
```

Текстовий файл ASPUSHKN.TXT повинен знаходитися в тому ж каталозі, що і програма.

Процедура Reset встановлює покажчик поточної позиції файлу на початок першого рядка текстового файлу. Процедура ReadLn зчитує рядок повністю, зберігає прочитане значення у змінній S та пересуває покажчик на початок наступного рядка. Рух допускається послідовно тільки в один бік – зліва направо (адже текстовий файл є файлом послідовного доступу!). Так продовжується доти, доки покажчик не досягне кінця файлу.

Приклад 15.4.12.

“Склеювання” текстових файлів. Завдання полягає у тому, щоб об’єднати два текстові файли в один, зберігши порядок розташування рядків. Для виконання цієї операції призначена програма Cat. Розберіть її самостійно.

```
program Cat;
var
    fil1, fil2, fil3 : text;
    s1, s2, s3 : String;
begin
Assign(fil1, ‘t1.txt’);  Assign(fil2,’t2.txt’);  Assign(fil3,’t3.txt’);
{Відкриваємо файли, пов’язані із змінними fil1, fil2 для читання, а fil3 для
запису}
Reset(fil1);  Reset(fil2);  ReWrite(fil3);
ReadLn(fil1, s1);  ReadLn(fil2, s2);
while not(Eof(fil1)) or not(Eof(fil2)) do
    if s1 < s2 then  begin
        WriteLn(fil3, s1);
        ReadLn(fil1, s1);
    end
    else  begin
        WriteLn(fil3, s2);
        ReadLn(fil2, s2);
    end;
```

```
if Eof(fil1) then
        while not(Eof(fil2)) do begin
            ReadLn(fil2, s2);
            Write(fil3, s2);
        end
else
    while not(Eof(fil1)) do begin
        ReadLn(fil1, s2);
        Write(fil3, s2);
    end;
Close(fil1);  Close(fil2);  Close(fil3);
end.
```

Приклад 15.4.13.

Співставлення із зразком. Є послідовність символів x[1]..x[n]. Визначити, чи є у ній символи "abcd", що йдуть один за одним. Іншими словами, потрібно з'ясувати, чи є у слові x[1]..x[n] підслово "abcd". Природно, що ми не користуватимемося готовою функцією Pascal – POS(SUBST, ST).

Рішення. Є приблизно n (якщо бути точним, n-3) позицій, на яких може знаходитися шукане підслово у початковому слові. Для кожної з позицій можна перевірити, чи дійсно там воно знаходиться, порівнявши чотири символи.

Проте є ефективніший спосіб. Читаючи слово x[1]..x[n] зліва направо, ми чекаємо появи букви 'a'. Як тільки вона з'явилася, ми шукаємо за нею букву 'b', потім 'c', і, нарешті, 'd'. Якщо наші очікування виправдовуються, то слово "abcd" виявлене. Якщо ж якась з потрібних букв не з'являється, ми опиняємося біля розбитого корита і починаємо все спочатку.

Цей простий алгоритм можна описати в різних термінах. Використовуючи термінологію так званих кінцевих автоматів, можна сказати, що при читанні слова *x* зліва направо ми в кожен момент знаходимося в одному з наступних станів: "початкове" (0), "одразу після a (1), "одразу після ab" (2), "одразу після abc" (3) і "одразу після abcd" (4). Читаючи чергову букву, ми переходимо у наступний стан за правилом:

Поточний стан	**Чергова буква**	**Новий стан**
0	a	1
0	крім a	0
1	b	2
1	a	1
1	крім a,b	0
2	c	3
2	a	1
2	крім a,c	0
3	d	4
3	a	1
3	крім a,d	0

Як тільки ми попадемо у стан 4, робота закінчується.

Відповідна програма зрозуміла (ми вказуємо новий стан, навіть якщо він збігається із старим; ці рядки можна опустити):

```
i:=1; state:=0;
{i – перша непрочитана буква, state – стан}
while (i <> n+1) and (state <> 4) do begin
  if state = 0 then begin
        if x[i] = a then begin
           state:= 1;
        end else begin
           state:= 0;
        end;
  end else if state = 1 then begin
        if x[i] = b then begin
           state:= 2;
        end else if x[i] = a then begin
           state:= 1;
        end else begin
           state:= 0;
        end;
  end else if state = 2 then begin
        if x[i] = c then begin
           state:= 3;
        end else if x[i] = a then begin
           state:= 1;
        end else begin
           state:= 0;
        end;
  end else if state = 3 then begin
        if x[i] = d then begin
           state:= 4;
        end else if x[i] = a then begin
           state:= 1;
        end else begin
           state:= 0;
        end;
  end;
end;
answer := (state = 4);
```

Іншими словами, ми кожного разу зберігаємо інформацію про те, який максимальний початок нашого зразка "abcd" є кінцем прочитаної частини. Його довжина і є той "стан", про який йшлося.

Приклад 15.4.14.

```
{ У заданому тексті одне задане слово скрізь замінити на інше задане слово }
Program Replace;
  Uses Crt;
  Var Text, Slovo1, Slovo2 : String;
        i, DlinaSlova, P     : Integer;
BEGIN ClrScr;
  Write('Введіть рядок: '); ReadLn(Text);
  Write('Яке слово замінити ? '); ReadLn(Slovo1);
  Write('На яке слово замінити? '); ReadLn(Slovo2);
  WriteLn; WriteLn('В і д п о в і д ь: ');
  WriteLn('Початковий текст: ', Text); DlinaSlova:=Length(Slovo1);
  DlinaSlova:=Length(Slovo1);
  P:=Pos(Slovo1,Text); {номер позиції, з якої у рядку Text     }
                       {вперше зустрічається підрядок Slovo1  }
  While P>0 do         {цикл продовжується доти, доки підрядок}
                       {Slovo1 зустрічається у рядку Text          }
   begin
     Delete(Text, P, DlinaSlova); {вилучення підрядка Slovo1, шо починається}
                                  {з позиції P, з рядка Text  }
     Insert(Slovo2, Text, P);  {вставка підрядка Slovo2  }
                               { у рядок Text з позиції P}
     P:=Pos(Slovo1, Text); {номер позиції, з якої підрядок Slovo1}
                           {зустрічаєтбся у рядку Text черговий раз}
   end;
  WriteLn('Новий текст: ', Text);
  ReadLn
END.
```

Приклад 15.4.15.

1.Створити файл даних, кожен запис якого складається з наступних полів:

Поля запису:

Назва книги Автор Рік видання Кількість сторінок

2. Зчитати з файлу інформацію.

```
program knigi;
uses crt;
const n=5;
type
        kniga=record {тип запису}
        avtor:string[20];
        god:integer;
        str:integer;
end;
kn = array[1..n] of  kniga; {масив записів}
```

```
var     sp:kn;
        i,j:integer;
        f: file of  kniga; {типизованний файл записів}
BEGIN
    clrscr;
    for i:=1 to n do  begin  { вводимо дані за умовою }
        write(‘avtor: ‘); readln(sp[i].avtor);
        write(‘god: ‘); readln(sp[i].god);
        write(‘str: ‘); readln(sp[i].str);
    end;
    assign(f, ‘kniga.txt’);
    rewrite(f);
    for i:=1 to n do   begin
        write(f, sp[i]); { записуємо їх у файл }
    end;
    close(f);
    reset(f);
    i:=1;
    while (not eof(f)) and(i<=n) do  begin
        read(f,sp[i]); { зчитуємо дані з файлу в масив }
        inc(i);
    end;
    writeln;
    writeln(‘Spisok knig >100 str, fam avtora na A:’);{ виводимо список книг, в яких >100 стр}
    for i:=1 to n do { і прізвище автора починається на А }
        if (sp[i].avtor[1]=‘A’) and (sp[i].str>100) then  begin
            writeln(sp[i].avtor,’ ‘,sp[i].god,’ god ‘,sp[i].str,’ str’);
        end;
    readln;
END.
```

15.5. Алгоритми сортування та пошуку

Традиційно розрізняють внутрішнє сортування, яке оброблює дані в оперативній пам'яті, і зовнішнє сортування, що оперує з даними, які знаходяться на дисках. Проблеми оптимізації істотно розрізняються для цих випадків. При внутрішньому сортуванні намагаються зменшити число порівнянь ключів і переміщень записів файлу (масиву). У зовнішньому сортуванні вирішальним фактором ефективності відповідного алгоритму стає кількість звернень до дискових пристроїв.

Надалі, для зручності, будемо вести мову тільки про внутрішнє сортування і розглядати файли, що являють собою одновимірні числові масиви – вектори. Записами і ключами таких файлів будемо вважати значення відповідних

елементів масивів. Це не є яким-небудь істотним обмеженням. Справа в тому, що обговорювані алгоритми легко переносяться і на інші випадки внутрішнього сортування.

За усталеною традицією всі різноманітні методи сортування поділяються на три основні групи: сортування вибором, сортування включеннями і обмінні сортування. Цей розподіл проводиться по домінуванню у відповідному алгоритмі типу дії: вибір, включення або обмін.

Нижче будуть розглянуті наступні три повільних алгоритми сортування: простим вибором, простими включеннями і простими обмінами (метод бульбашки), а також їх поліпшені модифікації і більш ефективні алгоритми – швидке сортування та пірамідальне сортування. Перші три з них для великих масивів мають погані характеристики часу. Їх трудомісткість дорівнює $O(n^2)$ операцій порівнянь і обмінів, де n – довжина сортованого масиву.

Швидке сортування, хоча і має в гіршому випадку трудомісткість $O(n^2)$, в переважній більшості реальних завдань дає кращі характеристики часу в порівнянні з іншими методами (приблизно $O(n \times log_2(n))$).

Існує також традиційний поділ алгоритмів на дві категорії: чисельні і нечисельні. Чисельні алгоритми призначені для математичних розрахунків: обчислення за формулами, вирішення рівнянь, статистичної обробки даних тощо. У таких алгоритмах основним типом оброблюваних даних є числа. Нечисельні алгоритми мають справу з найрізноманітнішими типами даних: символьними, графічними, мультимедійною інформацією. Для нечисельних алгоритмів більш характерна сортування масиву по полю, яке називається ключем сортування.

Для програмних продуктів другої категорії найбільш часто використовуються алгоритми сортування даних, тобто впорядкування інформації за деякою ознакою. Від ефективності цих алгоритмів, і насамперед від швидкості їх виконання, багато в чому залежить ефективність роботи всієї програми.

15.5.1. Обмінна сортировка. Сортування методом "бульбашки"

Метод "бульбашки" – це один з найпопулярніших методів сортування, і використовує метод обмінного сортування. Він заснований на тому, що елементи масиву з меншими (у разі сортування за зменшенням) або з більшими (у разі сортування за збільшенням) поступово переміщаються до кінця масиву, подібно до бульбашки повітря що переміщається до поверхні води. При цьому кожен елемент порівнюється не з усіма елементами, а тільки з сусіднім елементом.

Різні обмінні сортування відрізняються один від одного порядком проходу масиву і вибором елементів для обміну. Бульбашковеа сортування – це найпростіша з обмінних сортировок і, напевно, сортировок взагалі.

Ідея бульбашкового сортування – у визначенні неправильної пари. Неправильна пара – це пара поруч стоящих елементів, таких що перший

елемент пари більше другого. Повністю впорядкований масив – це масив, в якому немає жодної неправильної пари. Звідси ідея сортування. Необхідно організувати перебір пар масиву і щораз, коли буде виявлена неправильна пара, її елементи міняти місцями. Важливим є питання про кількість проходів масиву. Ясно, що одного проходу може виявитися недостатньо.

Розглянемо приклад. Нехай дано масив (5, 4, 3, 2, 1). Прохід масиву полягає в переборі 4 пар елементів. Виконаємо прохід покроково:

1. Порівнюються (5, 4). Результат (4, 5, 3, 2, 1).
2. Порівнюються (5, 3). Результат (4, 3, 5, 2, 1).
3. Порівнюються (5, 2). Результат (4, 3, 2, 5, 1).
4. Порівнюються (5, 1). Результат (4, 3, 2, 1, 5).

Зауважимо, що після першого проходу число 5 (найбільше в масиві) встало на своє законне місце. По всій видимості, після кожного проходу одне число буде займати призначене для нього місце. Виняток становитиме останній прохід, його виконання призведе до впорядкування відразу двох чисел. Таким чином, кількість проходів дорівнює N-1, якщо N – це кількість чисел.

Поясним більш детально походження назви сортування, тобто пояснимо, звідки взявся термін «бульбашка». Поставте сортований масив вертикально, так щоб початок виявився вгорі. Тоді виявиться, що числа, великі за значенням (важкі), опускаються вниз (тонуть), а менші (легкі) піднімаються вгору, як бульбашки.

Нижче показана найпростіша форма програми сортування методом бульбашки. Ця програма відрізняється від програми обмінного сортування тільки процедурою SortArray.

Приклад 15.5.1.

```
{ сортування бульбашковим методом}
program Sort_Bubble;
uses Crt;
var
      A: array[1..10] of byte;
      i, j, buf: byte;
{Виведення елементів масиву}
procedure ShowArray;
begin
      for i := 1 to 10 do Write(A[i]:4);
end;
{Сортування масиву. Якщо ByAsc = True, то –  за збільшенням.
Якщо ByAsc = False, то –  по убуванню. }
procedure SortArray (ByAsc: boolean);
begin
   for i := 1 to 9 do
      for j := 1 to 10-i do
        if (ByAsc and (A[j]> A[j+1])) or
```

```
          (not ByAsc and (A[j]< A[j+1])) then
         begin
            buf := A[j];
            A[j]:= A[j+1];
            A[j+1]:= buf;
         end;
end;
begin
   ClrScr;
   {Заповнюємо масив випадковими числами}
   Randomize;
   for i := 1 to 10 do A[i]:= Random (255 );
   Writeln ( ‘Масив до сортування: ‘ ) ;
   ShowArray;
   SortArray (True) ; {Сортування за збільшенням}
   WriteLn;
   Writeln ( ‘Масив відсортований за збільшенням:’);
   ShowArray;
   SortArray (False) ; {Сортування за зменшенням }
   WriteLn;
   Writeln ( ‘Масив відсортований за зменшенням:’);
   ShowArray;
   ReadKey;
end.
```

Приклад 15.5.2.

Ця версія сортування бульбашковим методом може сортувати символьний масив у порядку зростання значень елементів. Наприклад, наступна коротка програма сортує рядок, який зчитується з дискового файлу "test.dat".

У цьому випадку данне "item" є масивом елементів "DataItem", який сортується, а данне "count" містить число елементів у масиві. Сортування бульбашковим методом має два цикли. Оскільки число елементів масиву задається змінною "count", зовнішній цикл викликає перегляд масиву count – 1 раз. Це забезпечує знаходження кожного елементу у своїй позицій після завершення процедури. Внутрішній цикл призначений для фактичного виконання операцій порівняння і обміну.

```
 program SortDriver;
{ця  програма зчитуватиме 80 або менше символів з дискового файлу "test.dat",
відсортує їх та видасть результат на екран дисплея }
type
  DataItem = char;
  DataArray = array [1..80] of char;
var
  test: DataArray;
```

```
   t, t2: integer;
   testfile: file of char;
  { сортування бульбашковим методом}
procedure Bubble(var item: DataArray; count:integer);
var
  i, j: integer;
  x: DataItem;
begin
    for i := 2 to count do
    begin
       for j := count downto i do
          if item[j-1]> item[j]  then
          begin
             x := item[j-1];
             item[j-1]:= item[j];
             item[j]:= x;
          end;
       end;
   end;
begin
   Assign(testfile, 'test.dat');
   Reset(testfile);
   t := 1;
   { зчитування символів, які сортуватимуться}
   while not Eof(testfile) do begin
        read(testfile, test[t]);
        t := t+1;
   end;
   t := t-2; {скорегувати число лічених елементів }

   Bubble(test, t); { сортувати масив }
   { видати відсортований масив символів }
   for t2 := 1 to t do write(test[t2]);
   WriteLn;
end.
{ Для ілюстрації роботи сортування бульбашковим  методом  нижче
   наведені результати кожного етапу сортування масиву "dcab":
            – початкове положення: d c a b;
            – перший прохід:      a d c b;
            – другий прохід:      a b d c;
            – третій прохід:      a b c d.  }
```

15.5.2. Сортування відбором

Сортування відбором (або **вибором, обміном).** При сортуванні відбором (обміном) послідовно є видимим весь масив, визначається найбільший (у разі сортування за зменшенням) або найменший (у разі сортування за збільшенням) елемент, який потім міняється місцями з першим елементом масиву. Потім є видимими елементи масиву, починаючи з другого, і виконується аналогічна операція за визначенням найбільшого або найменшого елементу та його обміну з першим елементом поточної послідовності (наприклад, при третьому перегляді елементів масиву першим елементом поточної послідовності є третій елемент масиву і так далі). Цей цикл повторюється N-1 разів, де N – кількість елементів масиву, а першим елементом поточної послідовності елементів, яку потрібно проглянути, є N-1 елемент масиву.

У якомусь сенсі цей алгоритм також можна вважати обмінним. Тут черговий елемент обмінюється значенням з знайденим. Сенс алгоритму – в механізмі пошуку «правильного місця». На відміну від бульбашки, тут на кожному кроці черговий елемент зміщується, можливо, більше, ніж на одну позицію. Вірогідність цього «можливо, більше» сильно залежить від того, наскільки невпорядковані вихідний масив: якщо він майже впорядкований, то ефект може бути не надто великий. Якщо масив сильно не впорядкований, то ефект виявиться значним.

Приклад 15.5.3.

```
program Sort_Otbor;
uses Crt;
var
      A: array[1..10] of byte;
      i, j, buf: byte;
{Виведення елементів масиву}
procedure ShowArray;
begin
      for i := 1 to 10 do Write(A[i]:4);
end;
{Сортування масиву. Якщо ByAsc = True, то –  за збільшенням.
Якщо ByAsc = False, то –  за зменшенням. }
procedure SortArray (ByAsc: boolean);
begin
   for i :=1 to 9 do
      for j := i+1 to 10 do
         if (ByAsc and (A[i]> A[j])) or
            (not  ByAsc  and  (A[i]< A [ j ] ) ) then
         begin
            buf := A[i];
```

```
                A[i]:= A[j];
                A [ j ] := buf;
            end;
end;
begin
   ClrScr;
   {Заповнюємо масив випадковими числами}
   Randomize;
   for i := 1 to 10 do A[i]:= Random (255 );
   Writeln ( ‘Масив до сортування: ‘ ) ;
   ShowArray;
   SortArray (True) ; {Сортування за збільшенням}
   WriteLn;
   Writeln ( ‘Масив відсортований за збільшенням:’);
   ShowArray;
   SortArray (False) ; {Сортування за зменшенням}
   WriteLn;
   Writeln ( ‘Масив відсортований за зменшенням:’);
   ShowArray;
   ReadKey;
end.
```

15.5.3. Метод шейкер-сортування

Розглянемо складніші в реалізації, але ефективніші методи сортування. Зробимо порівняльний аналіз **шейкер-сортування** та **квадратичної вибірки**.

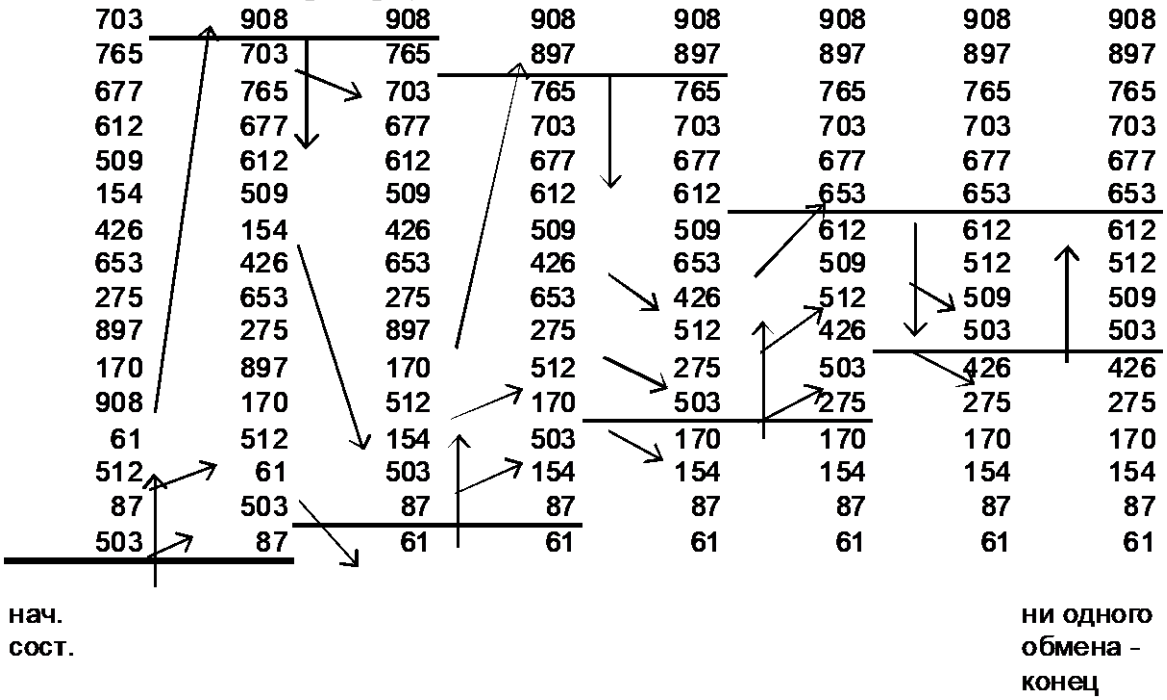
Метод шейкер-сортування. Метод шейкер-сортування є вдосконаленим варіантом методу бульбашки. Алгоритм передбачає наступні дії: набір записів є видимий від початку до кінця, кожні два сусідні елементи порівнюються і, якщо їх порядок порушує заданий порядок сортування, ці елементи міняються місцями. В результаті після першого перегляду максимальний (або мінімальний – залежно від заданого порядку сортування) елемент виявиться на останньому (тобто належному йому) місці.

Після цього масив є видимим у зворотному напрямі (на відміну від методу бульбашки, де напрям перегляду не міняється); при цьому здійснюються аналогічні дії (обміни). Після другого перегляду на перше місце в масиві також буде виставлений правильний елемент. А це означає, що надалі ані перший, ані останній елемент масиву можна вже не переглядати, що скорочує кількість порівнянь по відношенню до такого для методу бульбашки. Перегляд масиву продовжуються, поки при черговому з них не буде жодного обміну. Тобто, будемо виконувати наші проходи масиву: непарні – зліва направо і парні – справа наліво. При цьому на непарних проходах буде займати своє місце найбільший елемент (з решти), а при парних – найменший (також з решти). Такий спосіб називається ***шейкерним сортуванням***.

Метод квадратичного вибору. Метод квадратичного вибору є вдосконаленням методу простого вибору і був вперше опублікований Е.Х. Френдом. Якщо кількість сортованих елементів N є квадратом цілого числа (наприклад, $16 = 4^2$), то сортований файл може бути роздільний на $\sqrt{N}$ груп по $\sqrt{N}$ елементів у кожній. Будь-який вибір, окрім першого, вимагає не більше ніж $\sqrt{N} - 1$ порівнянь всередині групи раніше обраного елементу плюс $\sqrt{N} - 1$ порівнянь серед «лідерів груп». Наприклад, 16 елементів розбиваються на 4 групи по 4 елементи в кожній. Для кожної групи знаходиться максимальний (або мінімальний – в залежності від порядку сортування) елемент (для цього потрібно 3 порівняння усередині кожної групи). Серед цих «лідерів груп» визначається максимальний елемент всього файлу. Для пошуку наступного елементу спочатку знаходиться «новий лідер» тієї групи, в якій розташовувався тільки що відібраний елемент (2 порівняння), а потім серед оновленого складу «лідерів» знаходиться другий за величиною ключа у файлі (3 порівняння). Процес триває до закінчення елементів.

Приклад 15.5.4.

Ручний розрахунок. Розглянемо пробний набір з 16 записів. Представивши ключі записів у вигляді стовпчика, виконаємо сортування цих ключів методом шейкер-сортування:



Таким чином, процес впорядкування зажадав семи переглядів масиву (чотири в прямому напрямі і три – в зворотному).

Для впорядкування того ж набору методом квадратичного вибору масив ключів розбивається на чотири групи, в кожній з яких шукається максимальний елемент, і з цих максимальних елементів вибирається найбільший елемент масиву. Після цього в групі, з якої він був узятий, шукається наступний по

величині елемент, і знову виробляється порівняння максимальних елементів груп. Таким чином, потрібно правильно вибрати 15 елементів (невибраним залишається мінімальний елемент). Процес відбору показаний нижче:

703	703	703	703	703	703	703	703	703	703	703	703	703	703	703
765	765	765	765	765	765	765	765	765	765	765	765	765	765	765
677	677	677	677	677	677	677	677	677	677	677	677	677	677	677
612	612	612	612	612	612	612	612	612	612	612	612	612	612	612
509	509	509	509	509	509	509	509	509	509	509	509	509	509	509
154	154	154	154	154	154	154	154	154	154	154	154	154	154	154
426	426	426	426	426	426	426	426	426	426	426	426	426	426	426
653	653	653	653	653	653	653	653	653	653	653	653	653	653	653
275	275	275	275	275	275	275	275	275	275	275	275	275	275	275
897	897	897	897	897	897	897	897	897	897	897	897	897	897	897
170	170	170	170	170	170	170	170	170	170	170	170	170	170	170
908	908	908	908	908	908	908	908	908	908	908	908	908	908	908
61	61	61	61	61	61	61	61	61	61	61	61	61	61	61
512	512	512	512	512	512	512	512	512	512	512	512	512	512	512
87	87	87	87	87	87	87	87	87	87	87	87	87	87	87
503	503	503	503	503	503	503	503	503	503	503	503	503	503	503

Тут максимальні елементи кожної групи виділені сірим кольором, а максимальний з них – жирним шрифтом.

Експериментальні результати.

На невпорядкованих даних:

Метод	**Порівнянь**	**Обмінів**	**Пам'яті (у розмірах ключа)**
шейкер-сортування	71	41	5
квадратичний вибір	58	16	26 + копія масиву

На даних, впорядкованих в правильному порядку:

Метод	**Порівнянь**	**Обмінів**	**Пам'яті (у розмірах ключа)**
шейкер-сортування	15	0	5
квадратичний вибір	48	16	26 + копія масиву

На даних, впорядкованих в зворотному порядку:

Метод	**Порівнянь**	**Обмінів**	**Пам'яті (у розмірах ключа)**
шейкер-сортування	120	120	5
квадратичний вибір	48	16	26 + копія масиву

Проведені дослідження показали, що в середньому (на неврегульованих даних або даних, впорядкованих у зворотному порядку) метод квадратичного вибору вимагає меншої кількості порівнянь і обмінів, хоча вимагає великих резервів пам'яті. У разі ж, коли початкові дані вже впорядковані у правильному порядку, кращий результат дає шейкер-сортування (N – 1 порівнянь і 0 обмінів).

```
program Sort_SH_KV;
const n = 16;
    sqrtn = 4;
type data = array [1..N] of integer;
```

```
var ain, ash, aqu: data;
procedure Exch(var a, b: integer);   {обмін місцями}
var c: integer;
begin
        c := a;
        a := b;
        b := c;
end;

procedure Shake(var t: data; var cmpcnt, xchcnt: integer);
var i1, i2, it, i, k: integer;
begin
        cmpcnt := 0;
        xchcnt := 0;
        i1 := 1;
        i2 := N;
        repeat
            k := 0;
        for i := i1 to i2-1 do  begin
            inc(cmpcnt);
            if t[i] > t[i+1] then  begin
                exch(t[i], t[i+1]);
                inc(xchcnt);
                inc(k);
                it := i + 1;
              end;
        end;
        if k = 0 then exit;
        i2 := it –  1;
        k := 0;
        for i:= i2 downto i1+1 do  begin
            inc(cmpcnt);
              if t[i] < t[i-1] then  begin
                exch(t[i], t[i-1]);
                inc(xchcnt);
                inc(k);
                it := i –  1;
             end;
        end;
        i1 := it + 1;
        until k = 0;
end;
procedure Quad(var t: data; var cmpcnt, xchcnt: integer);
var     sel: array [1..sqrtn, 1..sqrtn] of integer;
```

```
        max, imax: array [1..sqrtn] of integer;
        i, j, l, m, im, jm: integer;
        res: data;
begin
        for i:=1 to sqrtn do
            for j:=1 to sqrtn do sel[i, j] := t[(i-1) * sqrtn + (j)];
        cmpcnt := 0;
        xchcnt := 0;
        for i:=1 to sqrtn do  begin
            max[i] := sel[i, 1];
            imax[i] := 1;
           for j := 2 to sqrtn do  begin
                inc(cmpcnt);
                if sel[i,j] > max[i] then  begin
                            max[i] := sel[i, j];
                            imax[i] := j;
                        end;
            end;
        end;
        for l := N downto 1 do  begin
             i := 1;
            while max[i] = 0 do inc(i);
             m := max[i];
            im := i;
            jm := imax[i];
           for j := i+1 to sqrtn do if max[j] > 0 then  begin
                inc(cmpcnt);
                if max[j] > m then  begin
                    m := max[j];
                    im := j;
                            jm := imax[j];
                        end;
           end;
           res[l] := t[(im-1)*sqrtn + (jm)];
           inc(xchcnt);
           sel[im, jm] := 0;
           jm := 1;
          while (sel[im, jm] = 0) and (jm <= sqrtn) do inc(jm);
          if jm > sqrtn then begin
                max[im] := 0;
                 imax[im] := 0;
         end
         else    begin
                max[im] := sel[im, jm];
```

```
                imax[im] := jm;
                for j:=jm + 1 to sqrtn do if sel[im, j] > 0 then  begin
                        inc(cmpcnt);
                                if sel[im, j] > max[im] then  begin
                            max[im] := sel[im, j];
                            imax[im] := j;
                        end;
                end;
          end;
    end;
    t := res;
end;
procedure PrintData(var dat: data);
var i:  integer;
begin
        for i:=1 to N do writeln(dat[i]:6);
end;

var     fname: string;
        fl: text;
        b: char;
        i: integer;
        ecsh, ccsh, ecqu, ccqu: integer;
BEGIN
writeln('Звідки вводити інформацію: k –  клавіатура, f –  файл?');
readln(b);
if upcase(b) = 'F' then   begin
    writeln('Введіть ім'я файлу');
    readln(fname);
    assign(fl, fname);
    reset(fl);
    for i := 1 to N do  begin
        readln(fl, ain[i]);
    end;
    close(fl);
end
else    begin
    writeln('Введіть початкові данні');
    for i := 1 to N do  begin
        write('k: '); readln(ain[i]);
    end;
end;
writeln('Початкові данні');
PrintData(ain);
```

```
writeln('Натисніть Enter');
readln;
ash := ain;
Shake(ash, ccsh, ecsh);
writeln('Результат шейкер-сортування');
PrintData(ash);
writeln('порівнянь –  ', ccsh, ';  обмінів –  ', ecsh);
writeln('Натисніть Enter');
readln;
aqu := ain;
Quad(aqu, ccqu, ecqu);
writeln('Результат сортування квадратичним вибором');
PrintData(aqu);
writeln('порівнянь –  ', ccqu, ';  обмінів –  ', ecqu);
writeln(' Натисніть Enter');
readln;
writeln('Таблиця експериментальних даних');
writeln('Метод          ':30, 'Порівнянь':10, 'Обмінів':10, 'Пам'яти (у розмірах ключа)':29);
writeln('шейкер-сортування':30, ccsh:10, ecsh:10, 5:10);
writeln('квадратичний вибір':30, ccqu:10, ecqu:10, N+sqrtn+6:10, ' + копія масиву');
writeln(' Натисніть Enter');
readln;
END.
```

15.5.4. Сортування вставкою

Ідея алгоритму полягає у наступному. Масив проглядається, починаючи з першого елементу до останього. Вважається, що елемент розташований на своєму місці, якщо всі елементи, що знаходяться лівіше його, меньші за значенням. І елемент стоїть не на своєму місці, якщо зліва від нього є елементи, які більші за значенням. Тоді, якщо у процесі перегляду масиву зустрівся елемент, який стоїть не на своєм місуі, необхідно визначити для нього «правильну позицію» і виконати вставку.

При сортуванні вставкою спочатку упорядковуються два елементи масиву. Потім робиться вставка третього елементу у відповідне місце по відношенню до перших двох елементів. Потім робиться вставка четвертого елементу в список з трьох елементів. Цей процес повторюється доти, доки всі елементи не будуть впорядковані. Наприклад, для масиву "dcab" сортування вставкою проходитиме таким чином:

- початковий стан: d c a b;
- перший прохід: c d a b;
- другий прохід: a c d b;

– третій прохід: a b c d.

Нижче наводиться алгоритм сортування вставкою. Ця програма відрізняється від програми обмінного сортування або методу бульбашки тільки процедурою SortArray.

Приклад 15.5.5.

```
{ сортування вставкою }
program Sort_Insert;
uses Crt;
var
    A: array[1..10] of byte;
    i, j, buf: byte;
{Виведення елементів масиву}
procedure ShowArray;
begin
    for i := 1 to 10 do Write(A[i]:4);
end;
{Сортування масиву. Якщо ByAsc = True, то – за збільшенням.
Якщо ByAsc = False, то – за зменшенням. }
procedure SortArray (ByAsc: boolean);
begin
  for i := 2 to 10 do  begin  { Перегляд невідсортованої частини}
     n := A[i];  {Обираємо елемент з невідсортованої частини}
     {Визначаємо позицію вставки }
     j := 1;
     while (ByAsc and (n > A[j])) or
          (not ByAsc and (n < A[j]))
        do Inc(j);
     {Зрушуємо елементи для вставки елементу n}
     for x := i-1 downto j do A[x+1]:= A[x];
     {Вставка елементу n}
     A[j]:= n;
  end;
end;
begin
  ClrScr;
  {Заповнюємо масив випадковими числами}
  Randomize;
  for i := 1 to 10 do A[i]:= Random (255 );
  Writeln ( 'Масив до сортування: ' ) ;
  ShowArray;
  SortArray (True) ; {Сортування за збільшенням}
  WriteLn;
  Writeln ( 'Масив відсортований за збільшенням:');
```

```
   ShowArray;
   SortArray (False) ; {Сортування за зменшенням }
   WriteLn;
   Writeln ( 'Масив відсортований за зменшенням:');
   ShowArray;
   ReadKey;
end.
```

15.5.5. Сортування Шелла

У 1959 році Дональд Шелл опублікував вдосконалений алгоритм сортування вставками, який згодом отримав його ім'я. Ідея даного методу полягає в порівнянні розділених на групи елементів деякої послідовності.

Мета алгоритму – змусити кожен сортований елемент за крок вставки переміщатися великим стрибком. Для цього вхідний масив розбивається на групи спеціальним чином. Спочатку, порівнюються елементи, які знаходяться один від одного на відстані *d* (перше знечення *d* дорівнює *N/2* , де *N* - кількість всіх елементів) . Поступово відстань між елементами зменшується, і при *d = 1* сортування відбувається востаннє.

Нижче можна спостерігати реалізацію сортування Шелла на прикладі послідовності чисел : 5, 3, 8, 0, 7, 4, 9, 1, 6, 2. Тут як проміжних значень взяті значення 5, 2, 1.

Початковий	5	3	8	0	7	4	9	1	6	2
d=5	4	3	1	0	2	5	9	8	6	7
d=2	1	0	2	3	4	5	6	7	9	8
d=1	0	1	2	3	4	5	6	7	8	9

Так элементи поступово стають на свої позиції, і на останньому кроці (*d=1*) число перестановок буде зовсім невелике.

Різні автори прпонують різні варіанти коду, що лежить в основі сортування Шелла. Одні визнчають даний алгоритм як «удосконалений варіант сортування вставками» (відповідно підгоняючи код під це визначення), інші вважають за доцільне використовувати «бульбашковий метод». Далі показані два способи реалізації алгоритму, які найбільш часто застрічаються.

Приклад 15.5.6.

```
{ Сортування Шелла – 1-й варіант }
Var n, i: integer;
mas: array [1..100] of  integer;
function ShellSort(a: array[1..100]  f  integer; leng: integer): integer;
var temp, step, k, i, j, p, d: integer;
gap: array [1..100] of  integer;
begin
```

```
        k:=1;
        gap[1]:=leng div 2;
        while (gap[k] > 1) do
        begin
           k:=k+1;
           gap[k]:=(gap[k-1] div 2);
        end;
        for i:=1 to k  do
        begin
           step:=gap[i];
           for j:=step to leng do
           begin
                 temp:=a[j];
                 p:=j-step;
                 while (p>=1) and (temp<a[p]) do
                 begin
                       a[p+step]:=a[p];
                       p:=p-step;
                 end;
                 a[p+step]:=temp;
            end;
          end;
          writeln;
          for i:=1 to leng do
                 write(a[i], ' ');
          end
{головна область}
Begin
        write('Розмір масиву = ');
        read(n);
        for i:=1 to n do
           read(mas[i]);
        writeln;
        ShellSort(mas, n);
end.
```

Приклад 15.5.7.

```
        { Сортування Шелла – 2-й варіант }
Var  i, j, n, d, ch: integer;
mas: array [1..100] of  integer;
begin
        write('n='); read(n);
        for  i:=1 to  n  do
            readln(mas[i]);
```

```
    d:=n;
    d:=d div 2;
    while (d>0) do
    begin
       for i:=1  to  n-d  do
       begin
            j:=i;
      { поки не досягнуто початок масиву і поки справжній елемент більше
      ніж елемент, що знаходиться на j+шаг}
            while ((j>0) and (mas[j]>mas[j+d]))  do
            begin
                  ch:=mas[j];
                  mas[j]:=mas[j+d];
                  mas[j+d]:=ch;
                  j:=j-1;
            end;
       end;
       d:=d div 2;
    end;
    writeln;
    for  i:=1 to n do
    write(' ', mas[i]);
end.
```

15.5.6. Швидке сортування

При «бульбашковому» сортуванні обміни ключів (елементів масиву) відбувався в сусідніх позиціях. Тут буде розглянуто алгоритм професора обчислювальної математики Оксфордського університету в Англії Ч. Хоара, що використовує дещо інший механізм вибору значень для обмінів. Цей алгоритм називають або сортуванням з поділом, або ***швидким сортуванням***.

У переважній більшості реальних випадків використання «швидкого сортування» дає кращі тимчасові результати в порівнянні з іншими методами. Проте слід мати на увазі, що для нього немає гарантованої трудомісткості типу $O(n*log_2(n))$ операцій порівнянь і обмінів, де n – кількість елементів масиву. Більш того, в інших ситуаціях трудомісткість досягає $O(n^2)$ операцій і не може бути знижена. Виходячи зі сказаного, в особливо критичних ситуаціях, де результат повинен видаватися в жорстко окресленому інтервалі часу, застосовувати швидке сортування слід дуже обачно. Автор мови Паскаль Н. Вирт висловився про цей алгоритм наступне: «... швидке сортування нагадує азартну гру, де слід заздалегідь розрахувати, скільки можна дозволити собі програти в разі невезіння».

Перейдемо до суті методу. Нехай одновимірний масив заданий вектором $v = (v_0\ v_1\ ...\ v_{n-1})$ і x – небудь компонент v. Спочатку обговоримо, як обмінами

елементів досягти розбиття *v* на дві частини *v1* і *v2* так, щоб в *v1* виявилися елементи, що не перевершують *x*, а в *v2* – які більше *x*. Зробити це можна за допомогою наступного простого і ефективного алгоритму.

Переглянемо *v*, рухаючись по компонентах зліва направо, поки не знайдемо елемент $v_i \geq x$. Потім переглянемо *v*, рухаючись по компонентам справа наліво, поки не знайдемо елемент $v_j \leq x$. Якщо $i \leq j$, то поміняємо місцями елементи *vi* і *vj* і застосуємо описану процедуру перегляду з обмінами до частини *v* від *vi+1* до *vj-1*. Продовжуючи описаний процес, поки $i \leq j$, отримаємо необхідний поділ вихідного масиву *v*. Звернемо увагу на наступні обставини. Обраний для порівняння елемент *x* служить бар'єром для двох поточних різноспрямованих переглядів, а простота перевірки умов виправдовує можливі зайві обміни елементів.

Приклад. Нехай у масиві *v*: 6, 23, 17, 8, **14**, 25, 6, 3, 30, 7 зафіксований елемент *x*=14. Переглядаємо *v* зліва направо, поки не знайдемо $v_i \geq 14$. Отримаємо v_1=23 (*i*=1). Далі переглядаємо *v* справа наліво, поки не знайдемо $v_j \leq 14$. Отримаємо v_9=7 (*j*=9). Міняємо місцями v_1 і v_9. Продовжуючи цей процес, прийдемо до послідовності 6, 7, 3, 8, 6 | 25, 14, 17, 30, 23, яка розділена на дві необхідні частини.

Тепер можна безпосередньо перейти до сортування *v*. Нам залишається зробити лише один крок. Після поділу масиву *v* на дві частини – *v1* і *v2* – для завершення розв'язання задачі залишається окремо впорядкувати *v1* і *v2*. Але зробити це можна, наприклад, рекурсивно продовжуючи поділ як *v1*, так і *v2*.

Приклад 15.5.8.

При швидкому сортуванні з розділенням спочатку визначимо елемент, який розташований в центрі масиву. Потім всі значення, які розташовані в лівій частині, і ті з них, які більше (у разі сортування за збільшенням) значення центрального елементу поміняємо з тими елементами, які розташовані в правій частині, значення яких менше значення центрального елементу. Таким чином, по закінченню цього циклу у лівій частині масиву не залишиться жодного елементу більше центрального. Потім кожна з частин у свою чергу розбивається на дві частини і для кожної з них повторюється описаний вище цикл. Таке розбиття виконується доти, доки одна з частин не буде зведена до одного елементу.

Схематично процес швидкого сортування елементів масиву представити складно внаслідок безлічі вкладених рекурсивних викликів процедури SortArray.

```
{ Рекурсивні алгоритми: Швидке сортування }
program Sort_Quick;
uses Crt;
const  N=10;
var
```

```
        A: array[1..N] of byte;
        i, j, buf: byte;
{Виведення елементів масиву}
procedure ShowArray;
begin
        for i := 1 to N do Write(A[i]:4);
end;
{Сортування масиву. Якщо ByAsc = True, то –  за збільшенням.
Якщо ByAsc = False, то –  за зменшенням. }
procedure SortArray (ByAsc: boolean; First, Last: byte) ;
var  mid : byte ;
begin
   i := First;  {Ліва межа}
   j := Last;  {Права межа}
   {Визначається центральний елемент}
   mid := A[(First + Last) div 2];
   repeat
      {Знаходимо в лівій частині позицію елементу
       який необхідно перенести у праву частину}
      while (ByAsc and (A[i]< mid)) or (not ByAsc and (A[i]> mid))
        do Inc(i);
{Знаходимо у правій частині позицію елементу, який можна перенести у ліву
частину}
      while (ByAsc and (A[j]> mid)) or (not ByAsc and (A[j]< mid))
        do Dec(j);
      if i <= j then begin
        {Обмінюємо місцями елементи з лівої та правої частини}
        buf :=  A[i];
        A[i]:= A[j];
        A[j]:= buf;
        Inc(i);   {Зміщуємо позицію у лівій}
        Dec(j);  {та у правій частині до центру}
      end;
   until i > j;  {Повторюємо цикл доти, доки не відбудеться "зустріч" при
зустрічному перегляді}
   {Рекурсивний виклик процедури для розбиття лівої частини}
   if First < j  then  SortArray (ByAsc, First, j) ;
   {Рекурсивний виклик процедури для розбиття правої частини}
   if i < Last  then  SortArray (ByAsc, i, Last) ;
end;
begin
   ClrScr;
   {Заповнюємо масив випадковими числами}
   Randomize;
```

```
  for i := 1 to N do A[i]:= Random (255 );
  Writeln ( ‘Масив до сортування: ‘ ) ;
  ShowArray;
  SortArray (True, 1, N) ; {Сортування за збільшенням}
  WriteLn;
  Writeln ( ‘Масив відсортований за збільшенням:’);
  ShowArray;
  SortArray (False, 1, N) ; {Сортування за зменшенням }
  WriteLn;
  Writeln ( ‘Масив відсортований за зменшенням:’);
  ShowArray;
  ReadKey;
end.
```

15.5.7. Пірамідальне сортування

Майже всі раніше згадані методи сортування послідовності a [1], a [2], ..., a [n] вимагали порівнянь порядку $O(n^2)$. Розглянемо один з найбільш елегантних і ефективних методів сортування складності $O(n*\log_2(n))$, запропонований Флойдом, і отримав назву «***Пірамідальне сортування***» або «***Сортування вспливанням Флойда***». Досі він залишається найоптимальнішим з існуючих методів. В алгоритмі активно використовується упорядковане двійкове дерево, приклад якого представлений на рис. 15.1 [http://abviz.ru/?p=152].

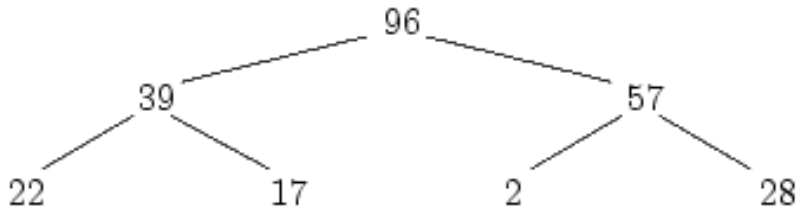


Рис. 15.1. Приклад упорядкованого двійкового дерева

Значення в кожній його вершині не менше, аніж значення в його дочірніх вершинах. Двійкове дерево називається частково упорядкованим, якщо властивість упорядкованості виконується для кожної з його вершин, однак для кореня ця властивість може порушуватися. Приклад частково упорядкованого дерева наведено на рис. 15.2 .

Цікаво відзначити, що в раніше розглянутих методах сортування складності $O(n^2)$ при виборі найбільшого (найменшого) елемента, забували інформацію про інших, забракованих елементах на цю роль, хоча ця перевірка і виконувалася. Структура ж дерева дозволяє зберегти стан процесу сортування послідовності a[1], a[2] , ... , a[n] на кожному його кроці, з метою використання цього стану в подальших розрахунках і зменшення числа операцій порівнянь при пошуку найбільшого (найменшого) з елементів, що залишилися.

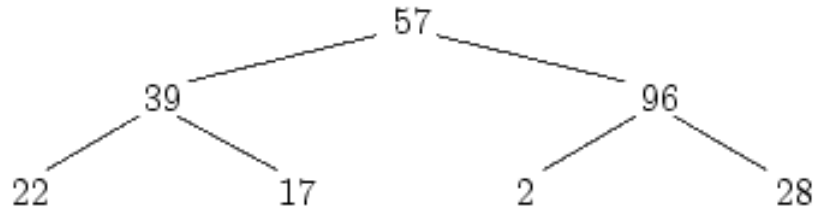


Рис. 15.2. Приклад частково упорядкованого двійкового дерева

Метод сортування Флойда представлений алгоритмом в лістингу прикладу 15.5.9, де початкова послідовність a[1], a[2], ... , a[n] даних представляється у вигляді дерева на суміжній пам'яті (одновимірний масив a [1 .. n]) . У такому дереві ребра присутствую неявно і обчислюються за допомогою арифметичних операцій над індексами елементів масиву – смажений пам'яті. Приклад двійкового дерева показаний на рис. 15.3 . Корінь дерева – a [1] , за кожною вершиною a [k] слідують вершини a [2k] і a [2k +1] . Використання суміжній пам'яті для представлення дерева має й інші переваги, які стають очевидними після аналізу алгоритму 15.5.9.

Основу алгоритму 15.5.9 становить процедура Surface(a [i .. k]) спливання Флойда, яка за O($\log_2$ (n)) порівнянь перетворює майже впорядковане поддерево у впорядкований. Піддерево представляється на одновимірному масиві a[i .. k], рис. 15.4, де a[i] – корінь поддерева, a[k] – максимальний елемент масиву, який ще може належати піддерево.

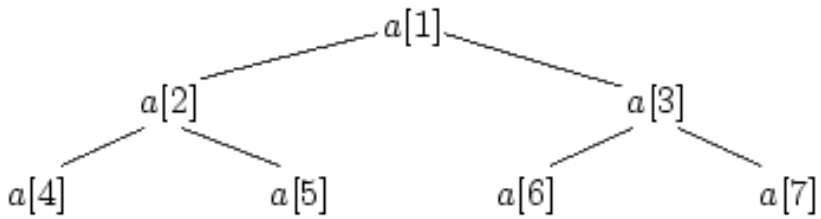


Рис. 15.3. Приклад двійкового дерева на суміжній пам'яті

Визначимо складність процедури Surface (поверхня) спливання Флойда . Процедура полягає в тому, що значення з кореня (тут може порушуватися умова впорядкованості) спливає у напрямку до листка (останній рівень вершин в дереві) доти, доки дерево не перетвориться в упорядковане. Під час спливання на кожному рівні виконується кінцеве число C операцій порівняння елементів. Якщо покласти, що висота дерева (число рівнів у дереві) дорівнює h, то складність одного спливання складе C • h = O(h) . Висота h регулярного двійкового дерева з n вершин легко знаходиться зі співвідношення n ≤ 2 ^ 0 +2 ^ 1 + ... +2 ^ (h - 1), де 2 ^ (i - 1) - кількість вершин на i-му рівні дерева , i = 1,2 , ... h . Звідси висота дерева h = | log2 (n +1) |. Таким чином, складність процедури Surface спливання Флойда становить O($\log_2$(n)).

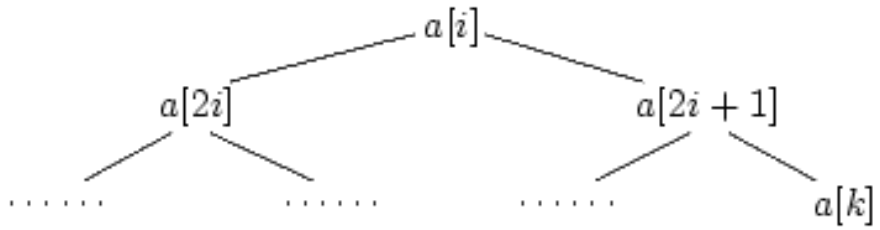


Рис. 15.4. Двійкове поддерево на суміжній пам'яті a[i..k]

Розглянута процедура Surface спливання Флойда дозволяє в майже упорядкованому дереві знайти найбільший (найменший) елемент за число порівнянь $O(\log_2(n))$, перетворюючи дерево до впорядкованого вигляду. У результаті знайдений елемент буде розташовуватися у вершині дерева. Для сортування же безлічі елементів a[1],a[2],…,a[n] з них за алгоритмом 15.5.9 спочатку організується майже впорядковане двійкове дерево за допомогою повторного застосування алгоритму Surface спливання Флойда, спочатку до найдрібніших його піддерев від листків, а потім до усе більш великих.

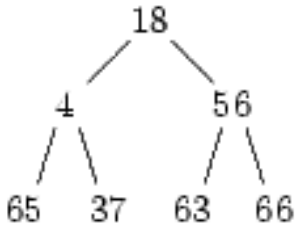


Рис. 15.5. Приклад вхідного дерева сортування

На рис. 15.5 показаний приклад вихідного дерева сортування елементів 18,4,56,65,37,63,66 . Листкі дерева тривіально впорядковані, тому можна почати з мінімальних піддерев, що містять кілька вершин, і укрупнювати їх, кожен раз повністю, застосовуючи алгоритм спливання доти, доки таким чином не буде досягнутий корінь дерева. Зауважимо, що кожне з піддерев, до яких застосовується алгоритм спливання, задовольняє умові майже впорядкованості, оскільки упорядкування проходить від листків до кореня. Саме таким способом в алгоритмі 15.5.9 здійснюється формування вихідного майже упорядкованого дерева.

Після того як дерево впорядковано, найбільший (найменший) елемент виявляється в його корені. За алгоритмом 15.5.9 знайдений елемент міняють місцями з самим останнім листком в дереві (останній елемент аналізованого масиву), дерево зменшується на одну вершину і все готово для визначення нового найбільшого (найменшого) елемента множини за допомогою наступного застосування процедури Surface спливання Флойда. На рис. 15.6 показана повна послідовність перестановок і спливання, які відбуваються після формування з вхідної множини майже упорядкованого дерева і аж до того, як в цьому дереві залишиться всього одна вершина, а вхідна множина виявиться відсортованою.

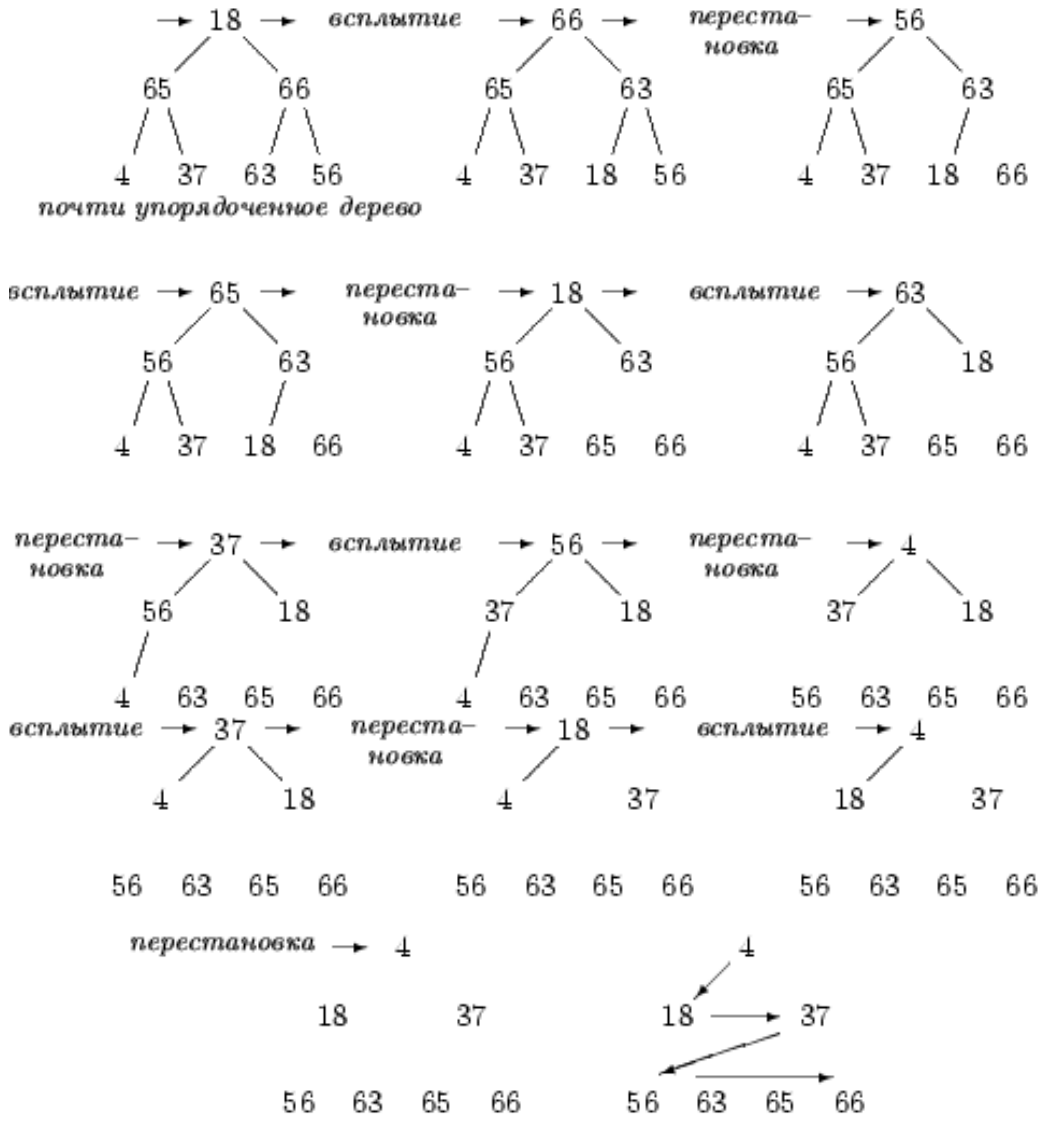


Рис. 15.6. Приклад сортування чисел 18,4,56,65,37,63,66 методом Флойда

<u>Приклад 15.5.9.</u> Пірамідальне сортування

```
Program Floid; { Сортування по зростанню вспливанням Флойда }
uses CRT;
Const n = 1000; { Розмір масиву даних для сортування }
Type  Vector = array[1..n] of Integer;
Var   f :Text; { Текстовий файл для результатів сортування }
{ Заповнити вектор a[1..n] випадковими числами }
procedure Init( var a: vector; n: integer);
var i: integer;
begin
  Randomize;
  for i:=1 to n do a[i]:=Random(100);
end;
{ Процедура спливання Флойда по дереву  a[i..k] }
procedure Surface( var a: Vector; i,k: integer);
```

```
var
j,m,copy :Integer;
begin
  copy:=a[i];
  m:=2*i;
  while m<=k do begin
    if m=k then j:=m
    else if a[m]>a[m+1] then j:=m else j:=m+1;
    if a[j]>copy then begin
      a[i]:=a[j];
      i:=j;
      m:=2*i;
    end
    else break; { вихід з циклу }
  end;
  a[i]:=copy;
end;
{ Сортування  вектора a[1..n] методом Флойда }
procedure Sort( var a: Vector; n: integer);
var
  i,k,w :Integer;
begin
  { Формувати початкове майже упорядковане дерево }
  for i:=n div 2 downto 2 do Surface(a,i,n);
  // Сортувати початкове  ПУДД
  for k:=n downto 2 do begin
    { Сплиття максимального елементу в корінь дерева a[0]}
    Surface(a,1,k);
    { Помістити максимальний елемент a[0] в кінець  a[k]}
    w:=a[k]; a[k]:=a[1]; a[1]:=w;
  end
end;
Var {Main}
  a : Vector; { Вектор вхідних даних для сортування }
  i : Integer;
begin; {Main}
  Assign(f,'sort.out');
  Rewrite(f); { Файл відкритий для запису }
  Init(a,n);
  { Зберегти вхідні дані }
  for i:=1 to n do WriteLn(f,'a[',i:1,']=',a[i]:3);
  Sort(a,n);
  { Зберегти сортовані дані }
  WriteLn(f);
```

```
  for i:=1 to n do WriteLn(f,'s[',i:1,']=',a[i]:3);
  Close(f);
End. {Main}
```

15.5.8. Бінарний пошук у впорядкованому масиві

Для пошуку деякого елементу у впорядкованому масиві можна скористатися простим послідовним перебором його елементів, проте використання такого підходу при роботі з масивами великого розміру займає багато часу. Одним з найбільш ефективних методів пошуку у великих впорядкованих масивах є бінарний пошук. При такому підході масив ділиться навпіл, і визначається його центральний елемент. Якщо центральний елемент дорівнює шуканому значенню, то пошук припиняється. Якщо центральний елемент більше шуканого значення, то ця ж операція розбиття застосовується до нижньої частини масиву. Якщо ж центральний елемент менше шуканого значення, то пошук продовжується у верхній частині.

Приклад 15.5.10.

Для створення, сортування і перегляду масиву скористаємося вже готовими процедурами SortArray, ShowArray з попередніх програм.

```
program  BSearch;
uses Crt;
var
        A: array[1..20] of byte;
        i, j, buf, n: byte;
        Found: boolean;
{Виведення елементів масиву}
procedure ShowArray;
begin
        for i := 1 to 10 do Write(A[i]:4);
end;
{Сортування масиву обміном. Якщо ByAsc = True, то –  за збільшенням.
Якщо ByAsc = False, то –  за зменшенням. }
procedure SortArray (ByAsc: boolean);
begin
   for i :=1 to 9 do
        for j := i+1 to 10 do
           if (ByAsc and (A[i]> A[j])) or
                (not  ByAsc  and  (A[i]< A [ j ] ) ) then
           begin
                buf := A[i];
                A[i]:= A[j];
                A [ j ] := buf;
          end;
```

```
end;

begin
   {Заповнюємо масив випадковими числами}
   Randomize;
   for i := 1 to 10 do A[i]:= Random (255 );
   SortArray (False) ; {Сортування за зменшенням }
   WriteLn;
   Writeln ( ‘Масив відсортований за зменшенням:’);
   ShowArray;
   Writeln;
   {Пошук в масиві, який відсортований за зменшенням }
   Write(‘Введіть число для пошуку: ‘);
   Readln(n);
   buf := 0;
   i := 1; {Ліва межа}
   j := 20; {Права межа}
   Found := False;
   repeat
        buf := (i + J) div 2;  {Номер центрального елементу}
        if A[buf]= n then begin    {Якщо центральний елемент дорівнює
шуканому}
             Found := True;   {Встановлюємо прапорець}
             break;   {Виходимо з циклу}
        end;
        {Якщо шуканий елемент більше центрального, то зміщуємо верхню
межу}
        if A[buf]< n then j := buf –  1;
        {Якщо шуканий елемент менше центрального, то зміщуємо нижню межу}
        if A[buf]> n then i := buf + 1;
   until i > j;
   if Found  then
        Writeln(‘Відповідає ‘,buf,’-му елементу’)
   else
        Writeln(‘Таке число у масиві відсутнє’);
   ReadKey;
end.
```

15.6. Графіка

А зараз приведемо декілька прикладів графічних програм із застосуванням найпопулярніших, на наш погляд, процедур модуля Graph і CRT.

Приклад 15.6.1.

```
{Ця програма демонструє роботу процедур управління текстовим виведенням}
{ на екран дисплея з використанням кольорів}
Program Color_TB;
  Uses Crt;        {підключення до програми бібліотеки Crt}
  Const P = ' ';
  Var i, j : Integer;
BEGIN
   ClrScr; {очистка екрану}
   Window(1, 1, 80, 7); {визначення вікна для заголовочної частини таблиці}
   TextColor(Yellow); {встановлення жовтого кольору символов}
   GoToXY(24, 1); WriteLn('ТЕКСТОВЕ ВИВЕДЕННЯ НА ЕКРАН
ДИСПЛЕЯ');
   GoToXY(30, 2); WriteLn('ТАБЛИЦЯ КОЛЬОРІВ');
   TextColor(LightCyan); {встановлення яркоблакітного кольору символів}
   WriteLn('0-Чорний ',P,'4-Червоний ',P,'8-Темносірий ',P,'12-Розовий ');
   WriteLn('1-Синій ',P,'5-Фіолетовий ',P,'9-Яркосиній ',P,'13-Маліновий ');
   WriteLn('2-Зелений ',P,'6-Коричневий ',P,'10-Яркозелений ',P,'14-Жлвтий ');
   Write ('3-Голубий ',P,'7-Світлосірий',P,'11-Яркоголубий',P,'15-Білий ');
   TextColor(3+128); WriteLn(' i+128-Миготіння'); TextColor(White);
   For i := 0 to 9 do  begin          {цикл по кольорах фону таблици кольорів}
      Window(i*8+1, 7, i*8+8, 25); {визначення вікна для стовчика таблици}
      GoToXY(1, 1); {курсор у верхньому лівому куту вікна}
      TextBackGround(Black); { за зменшенням чорного кольору фону}
      WriteLn(' Фон', i:2);
      WriteLn('----------');
      TextBackGround(i); { встановлення поточного кольору фону вікна }
      For j := 0 to 15 do  begin
         TextColor(j); { встановлення поточного кольору надписів у вікні }
         WriteLn('цвет', j:2);
      end;
    end;
    ReadLn
END.
```

Приклад 15.6.2.

```
{Програма управління світлофором}
PROGRAM Semafor;
USES Graph, CRT;
VAR Device, Mode, x,r, y_red, y_yellow, y_green: Integer;
            klavisha: Char;
BEGIN
      Device:=0;
      InitGraph(Device, Mode, 'c:\tp\bgi');
```

```
    x:=320;              {задаємо центр світлофора по горизонталі}
    r:= 50;                    {задаємо радіус вогнів світлофора}
    y_red:=110;          {задаємо центр червоного світла по вертикалі}
    y_yellow:=240;       {задаємо центр жовтого світла по вертикалі}
    y_green:=370;              {задаємо центр зеленого світла по вертикалі}
    Rectangle(x-100,40,x+100,440);       {малюємо світлофор}
    Circle(x,y_red,  r);
    Circle(x,y_yellow,r);
    Circle(x,y_green, r);
    repeat
      if KeyPressed then begin   {Якщо натиснута яка-небудь клавіша, то:}
        SetFillStyle(1,Black);           {гасимо світло, навіть якщо він не горів:}
        FloodFill(x,y_red,   White);     {верхнє світло}
        FloodFill(x,y_yellow,White);{середнє світло}
        FloodFill(x,y_green, White);     {нижнє світло}
        klavisha:= ReadKey;
        if klavisha='r' then       {якщо була натиснута r, то запалюємо червоний:}
          begin SetFillStyle(1,red);      FloodFill(x,y_red, White)  end;
        if klavisha='y' then               {якщо була натиснута y, то запалюємо жовтий:}
          begin SetFillStyle(1,yellow); FloodFill(x,y_yellow,White)  end;
        if klavisha='g' then               {якщо була натиснута g, то запалюємо зелений:}
          begin SetFillStyle(1,green);  FloodFill(x,y_green, White) end;
      end{if}
    until klavisha='s';        {якщо була натиснута s, то виходимо з програми}
    CloseGraph
END.
```

Приклад 15.6.3.

```
{Гра – постріл снарядом по тарілці, що летить}
PROGRAM  NLO;
USES Graph,CRT;
VAR x,y, Device, Mode, Shag: Integer;
BEGIN
    Device:=0;
    InitGraph(Device,  Mode,  'c:\tp\bgi');
    ReadLn;  {Чекаємо натиснення  клавіші Enter}
    Shag:=1;  {Крок руху тарілки та снаряда}
    x:=750; {Задаємо початкову координату тарілки, що летить}
    repeat          {Тарілка летить поодинці...}
        SetColor(White);
```

```
            Ellipse(x,100,0,360,50,10);
            Delay(20);
            SetColor(Black);
            Ellipse(x,100,0,360,50,10);
            x:=x-shag
        until KeyPressed; {доти, доки не буде натиснута будь-яка клавіша}
                          {після чого тарілка та снаряд летять одночасно:}
        y:=500;           {Задаємо початкову координату снаряда}
        repeat
            SetColor(White);
            Ellipse(x,100,0,360,50,10);      {малюємо тарілку, що літає}
            Ellipse(50,y,0,360,5,10); {малюємо снаряд}
            Delay(200);
            SetColor(Black);
            Ellipse(x,100,0,360,50,10);      {стираємо тарілку}
            Ellipse(50,y,0,360,5,10); {стираємо снаряд}
            x:=x-shag;                {переміщуємо тарілку}
            y:=y-shag     {переміщуємо снаряд доти }
        until y<0;            {доки снаряд не долетить до верху екрану}
        CloseGraph
END.
```

Приклад 15.6.4.

```
{Управління рухом крапки на екрані за допомогою клавіш}
PROGRAM  Move_Point;
USES Graph, CRT;
VAR Device, Mode, x, y, d     : Integer;
        klavisha: Char;
BEGIN
        Device:=0;
        InitGraph(Device, Mode, 'c:\tp\bgi');
        x:=320;       {Задаємо початкові координати точки}
        y:=240;
        d:=5;         {Задаємо крок переміщення крапки}
        PutPixel(x,y,White);      {Малюємо крапку у початковому положенні}
        repeat
          if KeyPressed then begin    {Якщо натиснута яка-небудь клавіша, то:}
            PutPixel(x,y,Black);      {стираємо крапку у старому положенні}
            klavisha:= ReadKey;
            if klavisha='d'     then x:=x+d;        {якщо натиснута d, то крок
праворуч}
            if klavisha='a'     then x:=x-d;        {якщо натиснута a, те крок
ліворуч}
            if klavisha='z'     then y:=y+d;        {якщо натиснута z, то крок
донизу}
```

```
            if klavisha='w' then y:=y-d;     {якщо натиснута w, то крок догори}
            if klavisha='p' then d:=d+1;     {якщо натиснута p, то крок
збільшуємо}
            if (klavisha='m') AND (d>0)   {якщо натиснута m і крок ще
позитивний}
               then d:=d-1;              {то крок зменшуємо}
            PutPixel(x,y,White);              {малюємо крапку в новому
положенні}
          end{if}
       until klavisha='s';        {якщо була натиснута s, то виходимо з програми}
       CloseGraph
END.
```

Приклад 15.6.5.

```
{Програма демонструє отримання ефекту руху зображення прицілу під}
{ управлінням клавіш-стрілок клавіатури з виведенням координат центру
прицілу.}
Program PRICEL;
Uses Crt, Graph;
Const Step = 5; {крок зміни координат центру прицілу }
   Instr = 'Management (control) a breech-sight: ->, <-,^. Exit –  ESC'; {Управління
прицілом}
Var        GrDriver, GrMode : Integer; {тип і режим роботи графичного
драйвера}
       X, Y          : Integer; {координати центру прицілу}
       XStr, YStr   : String;
       Ch            : Char;
Procedure MakeSight(X, Y : Integer); {процедура малювання прицілу}
Begin
   SetColor(White);
   Circle(X, Y, 80);
   SetColor(LightGreen);
   Line(X-80, Y, X+80, Y); Line(X, Y-63, X, Y+63); {виведення осей прицілу}
   SetColor(LightRed); Circle(X, Y, 2); {коло в центрі прицілу}
   Str(X, XStr); Str(Y, YStr); { переведення координат в строковий тип}
   SetColor(Yellow);
   OutTextXY(X+5, Y-35, 'x=' + XStr); {виведення координат центру прицілу }
   OutTextXY(X+5, Y-20, 'y=' + YStr)
End;
BEGIN
  GrDriver := Detect;
  InitGraph(GrDriver, GrMode, 'D:\BP\BGI');
  SetColor(LightGray);
  X := GetMaxX div 2; Y := GetMaxY div 2; {координати центру екрану}
```

```
  Rectangle(50, 425, 600, 460); {малювання рамки }
  OutTextXY(120, 440, Instr);
  MakeSight(X, Y); {малювання прицілу в центрі екрану}
  While TRUE do begin  {цикл роботи програми до переривання по клавіші
ESC}
      Ch := ReadKey;
      Case Ch of
        #27: begin CloseGraph; Halt(1) end; {вихід по клавіші ESC}
        #75: X:= X-Step; { зміна координат X, Y натисненням стрілок }
        #77: X:= X+Step; {"ліворуч", "праворуч", "догори", "донизу" }
        #72: Y:= Y-Step;
        #80: Y:= Y+Step
      end;
      ClearViewPort; { очистка графічного екрану }
      SetColor(LightGray); {відновлення рамки з надписом}
      Rectangle(50, 425, 600, 460);
      OutTextXY(120, 440, Instr);
      MakeSight(X, Y) {малювання прицілу у поточних координатах}
   end;
   CloseGraph;
END.
```

Приклад 15.6.6.

```
{ Крапка (або коло) обертається по колу у вертикальній площині. }
{ Використовується алгоритм: виводиться малюнок, у даному випадку точка
(коло)}
{ через певний  час малюнок формується кольором, який співпадає з фоном,}
{ що викликає стирання зображення, потім малюнок виводиться із зсувом}
{первинним кольором, стирається і так далі}
PROGRAM  Move_Point;
uses graph,crt;
Const         r = 50;         {радіус кола (траєкторії)}
Var     i,j,k,x,y: integer;
         t: real;
         grDriver: integer; { драйвер }
         grMode: integer;   { графічний режим }
         grPath: string;    { місце розташування драйвера }
         ErrCode: integer;  { результат ініціалізації граф. режиму }
begin
   grDriver := VGA; grMode:=VGAHi; grPath:='d:\bp\bgi';
   {режим VGA, дозвіл 640x480}
   {драйвер, файл EGAVGA.BGI, знаходиться у каталозі d:\bp\bgi }
   InitGraph(grDriver, grMode,grPath);
   ErrCode := GraphResult;
```

```
   if ErrCode <> grOk then Halt(1);
   while not keypressed do begin
      For i:=1 to 360 do  begin
        t := i*pi/180;
        x := Round(r*cos(t))+Getmaxx div 2;
        y := Round(r*sin(t))+Getmaxy div 2;
        {PutPixel(x,y,15);}        {рух крапки, колір 15 –  білий}
        setcolor(15);
        Circle(x,y,10);            {або рух кола}
        {Delay(100);}
        For  j:=1 to 32000 do      {Робимо затримку за допомогою двох порожніх
циклів}
             for k:=1 to 50 do;
        {PutPixel(x,y,0)}   { рух крапки, колір 0 –  чорний}
        setcolor(0);
        Circle(x,y,10);            { або рух кола }
      End;
  end;
  closegraph;
end.
```

Приклад 15.6.7.

```
{ Програма, яка створює зображення "чоловічка", що махає руками }
{ Спочатку побудуємо елементи, які будуть постійно присутні на екрані:}
{ голова, тулуб, ноги; потім в циклі через затримку виведемо три можливих }
{ положення рук (2 і 4 повторюється). Координати рук задамо у масивах X2 і
Y2.}
PROGRAM  Man;
uses  graph, crt;
const X2 : array[1..4] of integer = (310,310,310,310);
       Y2 : array[1..4] of integer=(230,212,196,212);
var i,j,k: integer;
     Driver: integer; { драйвер }
     Mode: integer;  { графічний режим }
     Path: string;   { місце розташування драйвера }
     ErrCode: integer; { результат ініціювання граф. режиму }
BEGIN
  Driver := VGA; Mode:=VGAHi; Path:='..\bgi';
  {режим VGA, дозвіл 640x480}
  {драйвер, файл EGAVGA.BGI, знаходиться у каталозі d:\bp\bgi }
  InitGraph(Driver, Mode, Path);
  ErrCode := GraphResult;
  if ErrCode <> grOk then Halt(1);
  while not keypressed do   begin
```

```
      setcolor(15);
      circle(250,200,11); {голова}
      line(250,212,250,270); {тулуб}
      line(250,270,230,310); {ліва нога}
      line(250,270,270,310); {права нога}
      {послідовне виведення 3 можливих положень рук}
      for i:=1 to 4 do begin
         setcolor(15);
         line(250,212,X2[i],Y2[i]);
         line(250,212,250+(250-X2[i]),Y2[i]);
         {delay(500);} {затримка}
         for j:=1 to 32000 do      {Робимо затримку за допомогою двох порожніх
циклів}
            for k:=1 to 8000 do;
         {повторний виведення для "стирання" і-ого положення рук}
         setcolor(0);
         line(250,212,X2[i],Y2[i]);
         line(250,212,250+(250-X2[i]),Y2[i]);
      end;
   end;
   readln;
   closegraph;
END.
```

Приклад 15.6.8.

```
{ На більярдному столі (великий прямокутник)  коло }
{(більярдна куля) під кутом літає по столу, відскркуючи від його }
{ країв за законом віддзеркалення. }
{ При натисненні клавіші s програма припиняє свою роботу}
USES Graph, CRT;
VAR x,y, dx,dy,  Device, Mode: Integer;   {dx –  крок кульки по горизонталі
       тобто відстань по горизонталі між двома послідовними
       зображеннями кола. dy –  аналогічно по вертикалі}
       klavisha: Char;
BEGIN
       Device:=0;
       InitGraph(Device, Mode, 'c:\tp\bgi');
       Rectangle(35,35,605,445);       {борти столу}
       x:=320;  y:=240;   {Починаємо рух кульки з центру}
       dx:=1;          dy:=1;       {Напрмок руху –  праворуч донизу}
       repeat
             SetColor(White);  Circle(x,y,10);
             Delay(200);
             SetColor(Black);  Circle(x,y,10);
```

```
            x:=x+dx; y:=y+dy;
            if (x<50) OR(x>590)      then
                dx:= -dx;        {Вдарившись об лівий або правий борт }
                            {кулька міняє горизонтальну складову }
                            {швидкості на протилежну}
            if (y<50) OR (y>430)  then
                dy:= -dy;        {Вдарившись об верхній або нижній борт }
                            {кулька міняє вертикальну складову }
                            {швидкості на протилежну}
            if KeyPressed  then       {Якщо натиснута будь-яка клавіша, то}
                klavisha:= ReadKey;  {запам'ятовуємо її }
        until  klavisha='s';        {якщо була натиснута s, то виходимо з програми}
END.
```

Приклад 15.6.9.

```
{Програма, що малює шахівницю.}
PROGRAM  Chess_Board;
USES Graph;
VAR  i, j, x, y, Driver, Mode :Integer;
BEGIN
        Driver:=0;
        InitGraph(Driver, Mode, 'c:\tp\bgi');
        y:=80;                          {горизонтальні лінії:}
    repeat
        Line(160,y,480,y);
        y:=y+40;
    until y>400;
        x:=160;                  {вертикальні лінії:}
    repeat
        Line(x,80,x,400);
        x:=x+40;
    until x>480;
    Rectangle(155,75,485,405);          {Рамка навколо дошки}
    {Замальовуємо клітини у шаховому порядку:}
    SetFillStyle(1,Yellow);
    y:=100;                          {центр верхнього ряду}
    for i:=1 to 4 do begin                  {чотири пари рядів клітин }
        x:=180;                  {центр найлівішого стовпця}
        for j:=1 to 4 do begin     {замальовуємо непарний ряд клітин }
            FloodFill(x,y,White);
            x:=x+80              {перескакуємо через клітку праворуч}
        end{for};
        y:=y+40;                 {перескакуємо донизу, у парний ряд клітин}
        x:=220;                  {центр другого зліва стовпця}
```

```
      for j:=1 to 4 do begin     {замальовуємо парний ряд клітин }
         FloodFill(x,y,White);
         x:=x+80                 {перескакуємо через клітку праворуч}
      end{for};
      y:=y+40;                   {перескакуємо вниз, в непарний ряд клітин }
   end{for};
   ReadLn;
   CloseGraph
END.
```

15.7. Програмування простих ігр

Приклад 15.7.1.

{Є 3^n монет, серед яких одна фальшива (важча за решту).
Потрібно за допомогою чашкових вагів без гирь рівно за **n** зважувань визначити номер фальшивої монети, загаданої користувачем. Користувач вводить:
0 – якщо ваги урівноважені;
1 – якщо переважила ліва чаша;
2 – якщо переважила права чаша.
}

```
Program Monet;
Uses CRT;
Const
    Nums: Array[1..9] of Integer = (3, 9, 27, 81, 243, 729, 2187, 6561, 19683);
    N = 4;
Var a: Array[1..20000] of Byte;
   Max, i: Integer;
   c: Char;
   Count: Byte;

Procedure Menu;
Begin
  Writeln(#13#10,'[0] – Якщо шальки терезів залишились у рівновазі');
  Writeln('[1] – Якщо ліва шалька терезів перевищила праву');
  Writeln('[2] – Якщо права шалька терезів перевищила ліву');
  Writeln('!!! Врахуйте, що ФАЛЬШИВА монета ВАЖЧЕ решти !!!');
End;

Procedure Find(min, max: Integer);
Var leng, part: Integer;
Begin
  leng := max – min; {Довжина ряду}
  If (Leng < 3) then  Begin
```

```
    Writeln('Останнєе зважування!');
    Writeln('Злева монета ',min,', Зправа монета ',max);
    Repeat
      Writeln('Яка шалька важче? Введіть число 0|1|2');
      Readln(c);
    Until (c='1') or (c='2') or (c='0');
    Write('Фальшива монета –  ');
    Case c of
      '0': Writeln(min+1);
      '1': Writeln(min);
      '2': Writeln(max);
    End;
    Writeln('Кінець гри');
    ReadKey;
    Halt;
  End;

  part := leng div 3; {Одна третина}
  Inc(Count);
  ClrScr;
  Writeln('Зважування номер ',Count);
  Writeln('На лівій шальці вагів лежать монети від ',min,' до ',min + part);
  Writeln('На правій шальці вагів лежать монети від ',min+part+1,' до ',max –
part-1);
  Repeat
  Writeln(#13#10,'Яка шалька важче? Введіть число 0|1|2');
  Menu;
  Readln(c);
  Until (c='1') or (c='2') or (c='0');
    Case c of
      '0': Begin
        Writeln('Рівновага. Фальшива монета десь серед ',max-part,' –  ',max,'
монет');
        ReadKey;
        Find(max-part,max);
      End;
      '1': Begin
        Writeln('Ліва тяжелее. Фальшива монета десь серед ',min,' –
',min+part,' монет');
        ReadKey;
        Find(min, min+part);
      End;
      '2': Begin
        Writeln('Права важче. Фальшива монета десь серед ',min+part+1,' –
',max –  part-1,' монет');
        ReadKey;
        Find(min+part+1, max-part-1);
      End;
  End;
End;
```

```
BEGIN
  ClrScr;
  Writeln('Загадайте монетку від 1 до ',Nums[N]);  ReadKey;
  Writeln('Загадали? Тоді приступимо...');  ReadKey;
  Find(1, Nums[N]);
END.
```

Приклад 15.7.2.

Приклад рекурсивної програми.

{ **Гра "Ханойські вежі"** полягає у наступному. Є три стержні. На перший з них надіта пірамідка з N кілець (великі кільця знизу, менші зверху). Потрібно перемістити кільця на інший стріжень. Дозволяється перекладати кільця із стержня на стріжень, але класти більше кільце поверх меншого не можна. Скласти програму, яка показує необхідні дії.

Рішення. Напишемо рекурсивну процедуру переміщення *i* верхніх кілець з *m-го* стержня на *n-ий* (останні кільця передбачаються великими за розміром і лежать на стержнях без руху).

Спочатку переноситься пірамідка з *i-1* кілець на третю паличку. Після цього *i-е* кільце звільняється, і його можна перенести куди слід. Залишається покласти на нього пірамідку.}

```
Var   i, m, n, s: integer;
 procedure move(i,m,n: integer); {рекурсивна процедура}
     var s: integer;
begin
     if i = 1 then begin
        writeln ('Зробити хід ', m, ' -> ', n);
     end else begin
        s:=6-m-n; {s –  третий стріжень: сумма номерів дорівнює }
        move (i-1, m, s);
        writeln ('Зробити хід ', m, ' -> ', n);
        move (i-1, s, n);
     end;
end;
begin
   i := 4;       {кількість кілець на першому стріжні}
   writeln('Кілкість кілець на першому стріжні – ',i);
   m := 1;     {номера стріжнів}
   n := 2;
  { s := 3;}
   move(i,m,n);
   readln
end.
```

Приклад 15.7.3.

{ Приклад гри "Хрестики-нолики"

Нагадаємо правила: гра йде на квадратному полі 3х3 клітки. Гравці по черзі ставлять в клітинах хрестики (один) та нулі (інший). Виграє той, хто першим замкнув лінію (вертикальну, горизонтальну, діагональну – неважливо).}

```
uses crt;
TYPE
SquareType = (
        EMPTY , {пусто}
        CROSS , {хрестик}
        NULL    {нолик} );
{масив гри}
PosType = array[0..8] of SquareType;

стартова позиція}
const pos: PosType =
(EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY);
{данна функція повертає true, якщо у масиві
гри є замкнена лінія хрестиків або ноликів}
function EndGame( z:SquareType):boolean;
begin
        {номера клітинок
        0,1,2,
        3,4,5,
        6,7,8 }
        EndGame := true;
        if (pos[0] = z) and (pos[1] = z) and (pos[2] = z) then exit;
        if (pos[3] = z) and (pos[4] = z) and (pos[5] = z) then exit;
        if (pos[6] = z) and (pos[7] = z) and (pos[8] = z) then exit;
        if (pos[0] = z) and (pos[3] = z) and (pos[6] = z) then exit;
        if (pos[1] = z) and (pos[4] = z) and (pos[7] = z) then exit;
        if (pos[2] = z) and (pos[5] = z) and (pos[8] = z) then exit;
        if (pos[0] = z) and (pos[4] = z) and (pos[8] = z) then exit;
        if (pos[6] = z) and (pos[4] = z) and (pos[2] = z) then exit;
        EndGame := false;
end;
{сюди повернемо кращий хід}
var retPos:PosType;

{функція пошуку}
function Search(s:SquareType; {наши фігури} alpha,beta:integer; {мінімум
та максимум}ply:integer{глибина}):integer;
var
        n,tmp:integer;
        f,opS:SquareType;
        findMove:boolean;
begin
{визнпчимо колір фігур суперника}
if s = NULL then opS := CROSS
else opS := NULL;
{якщо є замкнена лінія, повернемо оцінку}
if EndGame(opS) then begin
{суперник у вузлі замкнув лінію; це наш програш}
Search := -1;
```

```
exit;
end;
{поки не знайшли жодної вільної клітини}
findMove := false;
{переберем усі клітини дошки}
for n := 0 to 8 do
begin
f := pos[n];
if f = EMPTY then begin
findMove := true;
{зробили хід}
pos[n] := s;
{обрахунок}
{trap := -Search(opS,-beta,-alpha,ply+1);}
tmp := -Search(opS,-beta,-alpha,ply+1);
if tmp > alpha then begin
alpha := tmp; if ply = 0 then
Move(pos,retPos,sizeof(PosType));
end;
{відновимо позицію}
pos[n] := EMPTY;
if alpha >= beta then break;
end;
end;
if not findMove then Search := 0 {нічья}
else  Search := alpha;
end;

{виводить позицію на екран}
procedure ShowPos;
function Ch(n:integer):char;
begin if pos[n] = CROSS then Ch :='X'
else if pos[n] = NULL then Ch := '0'
else Ch := ' ';
end;
begin
Writeln('-------');
Write ('|', Ch(0), '|', Ch(1), '|', Ch(2), '|') ; Writeln;
Writeln('-------');
Write ('|', Ch(3), '|', Ch(4), '|', Ch(5), '|'); Writeln;
Writeln('-------');
Write ('|', Ch(6), '|', Ch(7), '|', Ch(8), '|'); Writeln;
Writeln('-------');
end;
{повертає true, якщо кінець гри}
function GameOver:boolean;
var
n:integer; begin
GameOver := false;
if not (EndGame(CROSS) or EndGame(NULL)) then
begin
for n := 0 to 8 do
if pos[n] = EMPTY then exit;
end;
GameOver := true;
end;

VAR
ch:integer;
```

```
tmp:integer;
n:integer;
BEGIN
HighVideo;
repeat
Writeln;
ShowPos;
Writeln;
if GameOver then begin
for n := 0 to 8 do pos[n] := EMPTY;
Writeln('Game Over!!!');
Writeln; ShowPos;
Writeln;
end;
repeat
Write('Enter move number [0..8] or -1 (exit): '); Read(ch);
until  (ch = -1) or
((ch >=0) and (ch <= 8) and (pos[ch] = EMPTY)) ;
if ch = -1 then break;
pos[ch] := CROSS;
if not GameOver then begin
tmp := Search(NULL,-2,2,0);
pos := retPos;
end;
until ch = -1;
LowVideo;
Writeln('done');
END.
```

Приклад 15.7.4.

```
{Гра у "пятнашки"}
program Game_Pjat;
uses crt, Graph;
var
  as:array[1..4,1..4] of string; {Двовимірний масив, містить елементи табло}
  bs:array[1..16] of integer; {Масив для заповнення випадковими числами}
  men:array[1..3] of integer; {Масив виводить елементи Головного меню}
  i,j:integer; {Змінні для роботи с масивами}
  strok, stolb:integer; {Координати пустого елементу}
  hod:integer; {Лічильник, зчитує кожний хід, який робить користувач}
  lom:integer; {Змінна для роботи з Головним меню}
  ch:char; {Змінна, якій присвоюється код натиснутої клавіши на клавіатурі}
  prov:boolean; {Перевірка правильності розкладу}
  f_txt: text;  {файлова змінна}

procedure Vivod;
{Процедура виведення на екран табло с цифрами сформоване на момент відображення}
var lx,ly:integer; {Координати виведення двовимірного масиву}
   x,y:integer;   {Координати клітин}
   j1,i1:integer; {Змінні лічильники, для малювання клітинок}
   w1,h1:integer; {Ширина и висота клітин}
begin
    OutTextXY(210,50,'For leaving press ESC');
    w1:=30;
    h1:=30; {Клетка размером 30 на 30}
    for i1:=0 to 3 do {Цикл, промальовування клітин}
```

```
        for j1:=0 to 3 do
        begin
          x:=235+j1*35; {Зсув клітин по x}
          y:=150+i1*35; { Зсув клітин по y}
          setFillStyle(1,1); {Кольор (синий) та стиль клітин (заповненння поточним кольором) }
          Bar(x,y,x+w1,y+h1); {Малювання клітин}
        end;
        lx:=245;
        ly:=162;
        for i:=1 to 4 do {Цикл виведення двовимірного масиву поверх клітин}
        begin
          for j:=1 to 4 do
          begin
            OutTextXY(lx,ly,as[i,j]); {Виведення тексту на екран}
            lx:=lx+35;
          end;
          lx:=245;
          ly:=ly+35;
        end;
        line(220,135,220,300);  {Малювання рамки}
        line(385,135,385,300);
        line(220,135,385,135);
        line(220,300,385,300);
    end;
```

procedure Tablo;

```
    {Формування табло при першому запуску заповнене випадковими та цифрами, що не повторюються }
    var b:integer;   {Змінна, якій присвоюється випадкове число}
       k,z:integer; {Лічильники для операцій с масивами}
    begin
      randomize;
      For z:=1 to 16 do
      begin
        b:=random(15); {Вибір випадкового числа}
        k:=1;
        while k<>17 do {Цикл доки не буде заповнений масив с цілими цифрами}
        begin
          if bs[k]=b then
          begin
            b:=random(17);
            k:=1;
          end
          else k:=k+1;
        end;
       bs[z]:=b;{Присвоєння чергового елементу масива, що не повторяється }
      end;
      z:=1;
      for i:=1 to 4 do {Заповнення двовимірного масиву, замість цифр з одновимірного, присвоюються  строкові елементи}
      begin
        for j:=1 to 4 do
        begin
          case bs[z] of
          1:  as[i,j]:='1 ';
```

```
            2:  as[i,j]:='2 ';
            3:  as[i,j]:='3 ';
            4:  as[i,j]:='4 ';
            5:  as[i,j]:='5 ';
            6:  as[i,j]:='6 ';
            7:  as[i,j]:='7 ';
            8:  as[i,j]:='8 ';
            9:  as[i,j]:='9 ';
            10: as[i,j]:='10';
            11: as[i,j]:='11';
            12: as[i,j]:='12';
            13: as[i,j]:='13';
            14: as[i,j]:='14';
            15: as[i,j]:='15';
            16: as[i,j]:='  ';
            end;
            z:=z+1;
          end;
        end;
        vivod; {Виведення табло на екран}
      end;

      Procedure Pusto;
      {Пошук пустого елементу в табло}
      begin
        for i:=1 to 4 do
        begin
          for j:=1 to 4 do
          begin
            if as[i,j] = '  ' Then {чт дорівнює поточний елемент пробілу}
            begin
              Strok:=i; {Якщо дорівнює, то присвоюються координати пустого
елементу}
              Stolb:=J
            end;
          end;
        end;
      end;

      procedure Perehod;
      {Введення напряму переходу}
      begin
        ch:=readkey; {Змінній присвоюється код натиснутої користувачем
клавіші на клавиатурі}
      end;

      procedure Zamena;
      {Пересування клітин з цифрами в залежності від вибору користувача}
      begin
        Perehod; {Процедура, введення напряму переходу}
        if ord(ch)=75 then {Якщо натиснута клавіша леворуч}
        begin
          if stolb<>4 then {Якщо це не останній елемент, що стоїть у межі
табло}
          begin
            as[strok,stolb]:=as[strok,stolb+1]; {На місце пустого елементу
пересувається елемент, що стоїть праворуч від пустого}
            as[strok,stolb+1]:='  '; {Елементу, що стоїть праворуч від пустого
присвоюється пустий елемент}
```

```
            stolb:=stolb+1; {Нова координата пустого елементу}
            hod:=hod+1; {Черговий зроблений хід}
          end;
        end;
        if ord(ch)=72 then {Якщо натиснута клавіша догори}
        begin
          if strok<>4 then {Якщо це не останній елемент, що стоїть біля межі
табло}
          begin
            as[strok,stolb]:=as[strok+1,stolb]; {На місце пустого елементу
присвоюється елемент, що стоїть знизу від пустого}
            as[strok+1,stolb]:='  '; {Елементу, що стоїть знизу від пустого
присвоюється пустий елемент}
            strok:=strok+1; {Нова координата пустого елементу}
            hod:=hod+1;    {Черговий зроблений хід}
          end;
        end;
        if ord(ch)=77 then  {Якщо натиснута клавіша праворуч}
        begin
          if stolb<>1 then {Якщо це не останній елемент, що стоїть на межі
табло}
          begin
            as[strok,stolb]:=as[strok,stolb-1]; {На місце пустого елементу
присвоюється елемент, що стоїть леворуч від пустого}
            as[strok,stolb-1]:='  '; {Елементу, що стоїть ліворуч від пустого
присвоюється пустий элемент}
            stolb:=stolb-1; {Нова координата пустого елементу}
            hod:=hod+1;    {Черговий зроблений хід}
          end;
        end;
        if ord(ch)= 80 then {Якщо натиснута клавіша донизу}
        begin
          if strok<>1 then {Якщо це не останній елемент, що стоїть на межі
табло}
          begin
            as[strok,stolb]:=as[strok-1,stolb];{На місце пустого елементу
присвоюється елемент, що стоїть зверху від пустого}
            as[strok-1,stolb]:='  '; {Елементу, що стоїть зверху від пустого
присвоюється пустий елемент}
            strok:=strok-1; {Нова координата пустого елементу}
            hod:=hod+1;    {Черговий зроблений хід}
          end;
        end;
        Vivod;
      end;

      procedure Proverka;
      {Перевірка чи правильно розложено табло}
      begin
        prov:=false;
        if (as[1,1]='1 ') and (as[1,2]='2 ') and (as[1,3]='3 ') and (as[1,4]='4 ')
          and (as[2,1]='5 ') and (as[2,2]='6 ') and (as[2,3]='7 ') and (as[2,4]='8 ')
          and (as[3,1]='9 ') and (as[3,2]='10') and (as[3,3]='11') and (as[3,4]='12')
          and (as[4,1]='13') and (as[4,2]='14') and (as[4,3]='15') and (as[4,4]='  ')
then
        begin
          prov:=true; {Якщо табло розложено вірно, то ІСТИНА}
          OutTextXY(230,100,'Congratulate You have won');
        end;
```

```
    end;

    procedure Game;
    {Підключення графіки та перехід у режим ГРИ}
    var grMode:integer; {Режим роботи відеосистеми}
        grPath:string; {Шлях до файлу}
        grDriver:integer;{ Драйвер відеоадаптера, що використовується програмою}
        ErrCode:integer;
    begin
      hod:=0;
      grDriver:=VGA;
      grmode:=VGAHi;
      grPath:='EGAVGA.BGI';
      { initGraph(grDriver, grMode,grPath);} {Ініціалізація графічного режиму}
      InitGraph(grDriver, grMode,'');
      ErrCode := GraphResult;
      if ErrCode <> grOk then begin
        Writeln('Graphics error:', GraphErrorMsg(ErrCode));
        halt
      end;
      Tablo; {Формування табло}
      Pusto; {Пошук пустого елементу}
      repeat {Цикл, поки не натиснута клавіша ESC або поки ігрок не переміг }
        Zamena; {Пересування у масиві}
        Proverka; {Перевірка чи є даний розклад вірним}
      until (ord (ch)=27) or (prov=true);
      closeGraph; {Закриття графічного режиму}
    end;
    procedure Help;
    {Перехід у режим довідки}
    var f_txt: text;   {Файлова змінна}
      g1:string; {Змінна для роботи з рядками у файлі}
    begin
      clrscr;
      assign(f_txt,'fhelp.txt'); {Об'ява файлу}
      reset(f_txt); {Відкриття файлу}
      readln(f_txt,g1); writeln(g1); {Присвоєння змінної рядка файла}
      readln(f_txt,g1); writeln(g1);
      readln(f_txt,g1); writeln(g1);
      readln(f_txt,g1); writeln(g1);
      readln(f_txt,g1); writeln(g1);
      readln(f_txt,g1); writeln(g1);
      readln(f_txt,g1); writeln(g1);
      writeln('For leaving press ENTER');
      readln;
      close(f_txt); {Закриття файлу}
    end;

    {ОСНОВНА ПРОГРАМА}
    begin
    {Виведення на екран головного меню}
    {Елементи Головного меню, один з яких замальован білим кольором, а решта зеленим}
      men[1]:=15;
      men[2]:=2;
      men[3]:=2;
      repeat
```

```
        clrscr;
        strok:=1; {Поточний рядок}
        Lom:=1;
        { Вибір кольорів }
        GoToXY(32,10); Textcolor(men[1]); writeln('Help');
        GoToXY(32,11); Textcolor(men[2]); writeln('Play');
        GoToXY(32,12); Textcolor(men[3]); writeln('Exit');
        ch:=readkey; {Вибір напрямку пересування елементів меню}
        if (ord(ch)=80)  then {Якщо донизу, тоді поточний стає білим, а нижній
стає зеленим}
        begin
          for i:=1 to 3 do
          begin
            if (men[i]=15) and (strok<>3) then
            begin
              men[strok]:=2;
              men[strok+1]:=15;
            end
          else strok:=strok+1;
          end;
        end;
        if ord(ch)=72 then {Якщо догори, то поточний білим, а верхній
зеленим}
        begin
          for i:=1 to 3 do
          begin
            if (men[i]=15) and (strok<>1)then
            begin
              men[strok]:=2;
              men[strok-1]:=15;
            end
            else strok:=strok+1;
          end;
        end;
        if ord(ch)=13 then    {Якщо натиснута ENTER}
        begin
          for i:=1 to 3 do
          begin
            if men[i]=15 then
            begin
              if Lom=1 then begin Help; break; end; {Перехід у режим довідки}
              if Lom=2 then begin Game; break; end; {Перехід у режим гри}
            end
            else Lom:=Lom+1;
          end;
        end;
      until Lom=3 {Доти, доки не натиснутий пункт меню EXIT}
    end.
```

15.8. Рекурсивні методи

Приклад 15.8.1. Ханойська Вежа.

Розглянемо класичну задачу, яка відома в літературі під назвою «Ханойська Вежа». Ханойська Вежа – це головоломка, яку у 1883 р. придумав французький математик Едуард Люка. Для тих, хто не знає, сутність головоломки у наступному. Є три стрижні та вісім дисків (кілець) різних

діаметрів. Спочатку всі диски зібрані на одному стрижні так, що менші диски лежать на великих. Люка пропонував перекласти всі диски з першого стрижня на третій, використовуючи другий. При цьому слід дотримувати наступне правило: диски можна перекладати з одного стрижня на інший, при цьому не можна класти диск поверх диска меншого радіусу.

За тією ж офіційною версією, заради підвищення інтересу до своєї головоломки Люка придумав легенду про башту Брами, збільшену копію Ханойської. Ця Вежа складалася з 64 золотих дисків, а стріжні були вирізані з алмазу. У початковому стані 64 кільця були надіті на першу піраміду та впорядковані за розміром. Ченці одного з буддійських монастирів повинні перекласти всі диски з першої піраміди (стріжня) на другу, виконуючи єдину умову – диск не можна покласти на диск меншого розміру. При перекладанні можна використовувати всі три стрижні.

Ченці перекладають один диск за одну секунду. Як тільки вони закінчать свою роботу, настане кінець світу. А це станеться після закінчення п'яти мільярдів століть. Завдання це не під силу і сучасним комп'ютерам, оскільки кількість ходів (переміщень) в ній дорівнює 2^{64}-1 (близько 10^{21}), а це, як відомо, велике число (18 446 744 073 709 551 615), і комп'ютер, що працює в сотню мільйонів разів швидше за ченців, не впорається з цим завданням в найближчі тисячоліття. Отже апокаліпсис настане нескоро. Спробуємо це завдання вирішити і ми, звичайно, при обмеженій кількості дисків.

Як нам вже відомо, суть рекурсивних методів – звернення завдання до самого себе. Ви також знаєте, що в Pascal існує можливість рекурсивного визначення функцій. Ця можливість є способом програмної реалізації рекурсивних алгоритмів. Проте побачити рекурсивне вирішення задачі (рекурсивний алгоритм) часто дуже непросто.

На майданчику (назвемо його А, Рис. 5.7) знаходиться піраміда, складена з дисків. Цю піраміду у тому ж вигляді потрібно перемістити на майданчик В. Прі виконанні цієї роботи, як нам вже відомо, необхідно дотримуватися наступних обмежень:

- перекладати можна лише по одному диску, який взятий зверху піраміди;
- класти диск можна або лише на підставу майданчика, або на диск більшого розміру;
- допоміжно можна використовувати майданчик С.

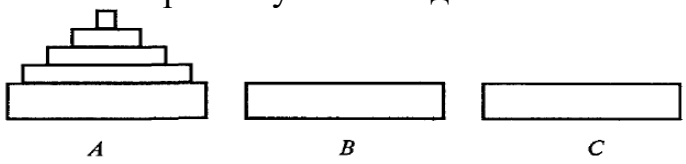


Рис. 5.7. Піраміда з дисками

Неважко вирішити це завдання для двох дисків. Позначимо переміщення диска, наприклад, з майданчика А на В так: А –>В, напишемо алгоритм для цього випадку А–>С; А–>В; С–>В. Всього 3 ходи!

Для трьох дисків алгоритм довший: А–>В; А–>С; В–>С; А–>В; С–>А; С–>В; А–>В. В цьому випадку вже потрібно 7 ходів.

Підрахувати кількість ходів (K) для n дисків можна по наступній рекурентній формулі (використання числових послідовностей, наступні значення яких можна знайти із попередніх, називають **рекурентними**): K(1)= 1; K(n)= 2*K(n-1)+ 1.

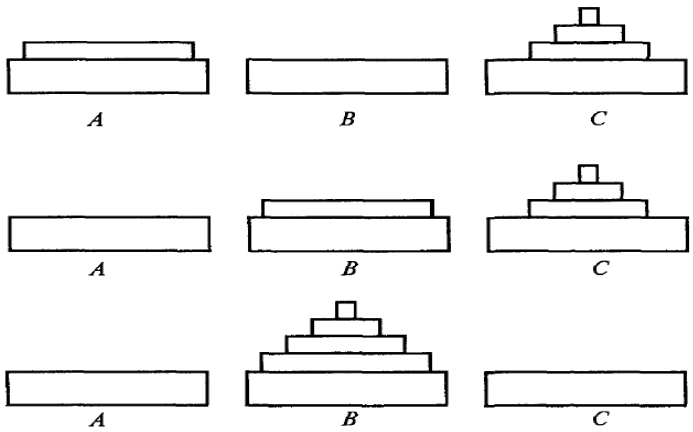
Або, використовуючи метод індукції, доводиться, що K(n)=2*K(n-1)+ 1 = 2^n – 1. Для цього спочатку перевіряється істинність твердження з номером 1 – база індукції, а потім доводиться, що якщо вірне твердження з номером n, то вірно і наступне твердження з номером n + 1 – крок індукції, або індукційний перехід.

Наприклад, K(10)= 2^{10} – 1 = 1023 – мінімальна кількість необхідних переміщень, K(20)= 2^{20} – 1 = 1048575.

Тепер складемо програму, по якій машина розрахує алгоритм роботи ченців і виведе його для будь-якого значення n (кількості дисків). Хай на майданчику A знаходяться n дисків. Алгоритм вирішення задачі буде наступним:

1. Якщо n = 0, то нічого не робити.
2. Якщо n > 0, то перемістити n-1 диск; перемістити диск A на B (A–>B) перемітити n-1 диск із C на B через A.

При виконанні пункту 2 послідовно матимемо три стани.



Опис алгоритму має явно рекурсивний характер. Переміщення n дисків описується через переміщення n-1 диска. Вихід з цієї послідовності рекурсивних заслань алгоритму самого на себе в пункті 1.

А тепер побудуємо програму на Pascal. У ній є рекурсивна процедура Tower, виконання якої закінчується лише при n = 0. При зверненні до процедури використовуються фактичні імена майданчиків, задані їх номерами: 1, 2, 3 (Рис. 5.8). Тому на виході ланцюжок переміщень описуватиметься у такому вигляді: 1) 1=>2 2) 1=>3 3) 2=>3 і так далі

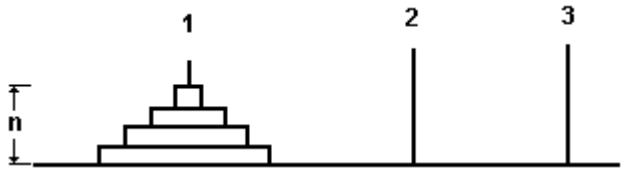


Рис.5.8. Задача про Ханойські вежі

```
Program H_Tower;
Var N: Byte; K: Longint;
```

```
Procedure Tower(N: Byte;A,B,C:Char);
Begin
   If N > 0 Then Begin
            Tower(N-1,A,C,B);
            K:=K+1;
            WriteLn(K,') ',A,' => ',B);
            Tower(N-1,C,B,A)
   End
End;
Begin
   K:=0;      {K –  кількість ходів}
   Write('Вкажить кількість дисків: ');
   ReadLn(N);
   Tower(N, '1','2','3');
   WriteLn('Кількість ходів = ',K);
   ReadLn
End.
```

15.9. Задачі на обчислення безкінечних сум та перебір всіх варіантів

Приклад 15.9.1.

Багато з математичних величин або значень функцій може бути виражені як суми безконечних послідовностей. Чим більше членів ряду бере участь у додаванні, тим більше точним виходить шукане значення. Різниця отриманої суми ряду та дійсної суми називається похибкою додавання. Цю похибку можна оцінити при обчисленні *n*-ого члена. Якщо він досить малий, тобто менше деякого даного числа **eps**, то вважається, що знайдена сума задовольняє вимогам по точності.

Програму для обчислення числа π ми вже наводили в розділі 3 (приклад 3.17). Наведемо програму, яка обчислює косинус кута, використовуючи формулу:

$$\mathbf{Cos(x) = 1-x^2/2!+x^4/4!-x^6/6!+\ldots+(-1)^n x^{2n}/(2n)!+\ldots}$$

```
Program Cosinus;
const eps = le-7;      { точність обчислення }
var x, s, r: real; n: integer;
Begin
    write('Введіть аргумент у градусах:   ');  readln(x);
    x:= x*pi/180;        { переводимо кут у радіани }
    s:= 1; r:= l; n:= 0;
    repeat
        n:= n+2;
        r:= -r*x*x/((n-1)*n);
        s:= s+r;
    until abs(r) < eps;
    writeln('cos(',x*180/pi: 6:2,') = ',s:7:3);
End.
```

Приклад 15.9.2.

Оскільки особливістю комп'ютерних обчислень є надзвичайно висока швидкодія, то деякі задачі, які не мають аналітичного рішення (або мають дуже

складне рішення), комп'ютер без зусиль може вирішити "тупим" перебором усіх можливих варіантів. Тому велику групу циклічних програм складають так звані задачі на повний перебір. Наведемо приклад такого завдання.

Є така старовинна задача. Скільки можна купити биків, корів та телят, якщо плата за бика 10 рублів, за корову – 5 рублів, за теляти – 0,5 рубля, якщо на 100 рублів треба купити 100 голів худоби.

На всяк випадок, якщо рішення потрібно отримати не за допомогою програми, а математичними методами, то слід почитати про так звані Діофантови рівняння – це такі системи рівнянь, в яких кількість невідомих більша кількості рівнянь, але такі системи все ж вирішуються за рахунок того, що змінні цілі і позитивні (якраз наш випадок).

Позначимо через *b* – кількість биків; *k* – кількість корів; *t* – кількість телят. Після цього можна записати два рівняння: 10b+5k+0.5t=100 і b+k+t=100. Перетворимо перше з них, помноживши на 2, аби були цілі коефіцієнти, після чого отримаємо два рівняння з трьома невідомими: 20b+10k+t=200 та b+k+t=100. На 100 рублів можна купити:

- не більше 10 биків, тобто 0 <= b <= 10;
- не більше 20 корів, тобто 0 <= k <= 20;
- не більше 200 телят, тобто 0 <= t <= 200. Таким чином, отримуємо:

```
Program Byki_Korovy_1;
var b, k, t:  integer;
Begin
   for b := 0 to 10 do              {обмеження по биках}
      for k := 0 to 20 do  {обмеження по коровах}
         for t := 0 to 200 do       {обмеження по телятах}
           if(20*b+10*k+t=200) and (b+k+t=100)  then
              writeln('Биків ', b, ', корів ', k, ', телят ', t);
   readln;
End.
```

Скільки разів перевірятиметься умова у даній програмі? Значення змінної *b* змінюється 11 разів (від 0 до 10), значення змінної *k* змінюється 21 раз, а значення змінної *t* змінюється 201 раз. Таким чином, умова перевірятиметься 11*21*201 раз. Але якщо відома кількість биків та корів, то кількість телят можна обчислити за формулою t=100-(b+k) і цикл по змінній *t* можна вилучити.

```
Program Byki_Korovy_2;
var b, k, t: integer;
Begin
   for b := 0 to 10 do              { обмеження по биках}
      for k := 0 to 20 do { обмеження по коровах}
      begin
        t:= 100-(b+k);
         if(20*b+10*k+t=200) then
            writeln('Биків ', b, ', корів ', k,', телят ', t);
```

```
        end;
    readln;
End.
```

При цьому рішенні умова перевіряється 11*21 раз.

Список літератури

1. Немнюгин С. Изучаем Turbo Pascal. / Л. В. Перколаб, С. А. Немнюгин. – СПб: Питер, 2007. – 320 с.
2. Васильев П.П. Turbo Pascal - мой друг. – М., "Компьютер, ЮНИТИ", 1995. – 98 с.
3. Гвоздева В. А. Введение в специальность программиста: учебник. – 2-е изд., испр. и доп. – М: ИД «ФОРУМ»: ИНФРА-М, 2007. – 208 с.
4. Федоренко Ю. Алгоритмы и программы на Turbo Pascal. Учебный курс. – СПб: Питер, 2001. – 240 с.
5. Семакин И.Г., Шестаков А.П. Основы программирования: Учебник. – М.: Мастерство, 2002. – 432 с.
6. Шпак Ю. А. Turbo Pascal 7.0 на примерах /Под ред. Ю. С. Ковтанюка – К.: Юниор, 2003. – 496 с.
7. Лукин С.Н. TURBO PASCAL 7.0. Самоучитель для начинающих. – М.: "Диалог-МИФИ", 1999. – 400 с.
8. Г. Майерс (Надежность программного обеспечения. – М.: «Мир», 1980. – 360 с
9. Ван Тассел Д. Стиль, разработка, эффективность, отладка и испытание программ. – М.: Мир, 1981. – 316 с.
10. Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.: Мир, 1975. – 245 с.
11. Гольденберг В.А. Введение в программирование. – Минск, "Харвест", 1997. – 528 с.
12. Зеленяк О. Практикум программирования на Turbo Pascal. Задачи, алгоритмы и решения. / О. П. Зеленяк. – СПб.: ДиаСофтЮП, ДМК Пресс, 2007. – 320 с.

Навчальне видання

Авраменко Валентин Семенович
Салапатов Володимир Іванович

ВСТУП ДО ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Том 2. Основи програмної інженерії

Навчальний посібник

Загальний редактор *Г. В. Косенюк*

Комп'ютерне верстання *Л. Г. Любченко*

Підписано до друку 11.06.2012. Формат 60x84/16.
Ум. друк. арк. 16,93. Наклад 300 пр. Зам. № 4351

Видавець і виготовлювач
Черкаський національний університет імені Богдана Хмельницького
Адреса: 18000, м.Черкаси, бул.Шевченка, 81, кімн. 117,
Тел. (0472) 37-13-16, факс (0472) 37-22-33,
e-mail: vydav@cdu.edu.ua, http://www.cdu.edu.ua
Свідоцтво про внесення до державного реєстру
суб'єктів видавничої справи ДК №3427 від 17.03.2009 р.