МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

М. І. Безменов

ОСНОВИ ПРОГРАМУВАННЯ В СЕРЕДОВИЩІ DELPHI

Рекомендовано Міністерством освіти та науки України
як навчальний посібник для студентів вищих навчальних закладів,
які навчаються за напрямами галузі знань «Інформатика та
обчислювальна техніка» та «Інформатика»

Харків
НТУ «ХПІ»
2010

ББК 32.973-018.1
Б 39
УДК 004.43 (075)

Р е ц е н з е н т и :

О. І. Пушкар, д-р екон. наук, проф., Заслужений діяч науки і техніки України, Національний економічний університет;

К. О. Соловйова, д-р техн. наук, проф., Харківський національний університет радіоелектроніки;

В. М. Вартанян, д-р техн. наук, проф., Національний аерокосмічний університет ім. М. Є. Жуковського «Харківський авіаційний інститут».

Гриф наданий Міністерством освіти і науки України,
лист № 1.4/18–Г–2770 від 19.12.08

Б 39 **Безменов М. І.** Основи програмування у середовищі Delphi : навч. посіб. – Харків : НТУ «ХПІ», 2010. – 608 с.
ISBN 978–966–593–791–3

Описано можливості однієї з найпопулярніших мов програмування – Delphi 7. Наведено достатньо повний опис базової для Delphi 7 мови Object Pascal, інтегрованого середовища розробника, а також основних компонентів. Подано коротку історію розвитку обчислювальної техніки та мов програмування. Викладання супроводжується великою кількістю прикладів програм, а також контрольними запитаннями й завданнями. Наведені початкові відомості про інтегроване середовище CodeGear Delphi 2009 for Win32.

Призначено для студентів вищих навчальних закладів, які навчаються за напрямами галузі знань «Інформатика та обчислювальна техніка» та «Інформатика». Може бути рекомендовано учням випускних класів.

Іл. 39. Табл. 19. Бібліогр.: 13 назв.

ББК 32.973-018.1

ISBN 978–966–593–791–3

ЗМІСТ

Передмова **8**
1. Комп'ютер і його стисла історія **9**
1.1. Початкові відомості про апаратне і програмне забезпечення комп'ютерів 9
1.2. Алгоритми і розробка програм 13
1.3. Розвиток обчислювальної техніки: події та дати 17
1.4. Розвиток мов програмування 27
1.5. Еволюція Delphi 31
Запитання для контролю і самоконтролю 33
2. Початки програмування у середовищі Delphi **35**
2.1. Найпростіший приклад розв'язання задачі 35
2.2. Два етапи розробки програми 36
2.3. Початкові відомості про середовище розроблювача Delphi 7 38
2.4. Початкові відомості про середовище розроблювача CodeGear Delphi 2009 for Win32 43
2.5. Проект 51
2.6. Модуль форми 54
2.7. Проектування програми 56
Запитання для контролю і самоконтролю 66
Завдання для практичного відпрацьовування матеріалу 67
3. Базові елементи мови Delphi **69**
3.1. Алфавіт мови 69
3.2. Стандартні директиви і визначені імена 70
3.3. Ідентифікатори та змінні 71
3.4. Константи 72
3.5. Система типів 74
3.6. Змінні і їх оголошення 76
3.7. Мітки 77
3.8. Підпрограми 77
3.9. Коментарі та директиви компілятору 77
3.10. Вирази й операції 78
3.11. Оператори. Оператор присвоювання 80
3.12. Розділ опису типів 80
3.13. Скалярні типи 81
3.14. Оголошення констант 96
3.15. Еквівалентність і сумісність типів 99
3.16. Бітова арифметика 104
3.17. Найпростіші введення та виведення 107
Запитання для контролю і самоконтролю 114
Завдання для практичного відпрацьовування матеріалу 117

4. Конструкції керування 119
4.1. Найпростіший оператор перевірки умови 119
4.2. Оператор вибору 125
4.3. Оператор безумовного переходу 130
4.4. Примусове припинення програми 131
4.5. Цикли 131
4.6. Оператори Break та Continue 138
Запитання для контролю і самоконтролю 140
Завдання для практичного відпрацьовування матеріалу 140
5. Складені типи 145
5.1. Масиви 145
5.2. Робота з рядками 158
5.3. Множини 181
5.4. Записи 188
Запитання для контролю і самоконтролю 195
Завдання для практичного відпрацьовування матеріалу 196
6. Файли 199
6.1. Загальні принципи роботи з файлами 199
6.2. Текстові файли 202
6.3. Опрацювання помилок введення/виведення 208
6.4. Типізовані файли 210
6.5. Нетипізовані файли 216
6.6. Інші підпрограми для роботи з файлами 220
Запитання для контролю і самоконтролю 226
Завдання для практичного відпрацьовування матеріалу 226
7. Вказівники і динамічні дані 229
7.1. Посилальні типи і вказівники. Тип Pointer 229
7.2. Адресна арифметика і вказівники 233
7.3. Динамічний розподіл пам’яті 238
7.4. Зв’язані списки 243
Запитання для контролю і самоконтролю 252
Завдання для практичного відпрацьовування матеріалу 252
8. Підпрограми 255
8.1. Функції 255
8.2. Процедури. Види параметрів 263
8.3. Нетипізовані параметри 267
8.4. Випереджальний опис 271
8.5. Рекурсивні підпрограми 271
8.6. Відкриті масиви та рядки. Масиви констант 274
8.7. Процедурний тип 278
8.8. Функції, що повертають вказівник 284
8.9. Підпрограми з параметрами за умовчанням 291
8.10. Перевантажувані підпрограми 292

8.11. Використання стандартних директив, асемблерних вставок, інструкцій машинного коду. Вставка фрагментів, написаних мовою Delphi 294
Запитання для контролю і самоконтролю 296
Завдання для практичного відпрацьовування матеріалу 297
9. Модулі 299
9.1. Призначення модулів та їх структура 299
9.2. Компіляція програм, що використовують модулі 302
9.3. Приклад оформлення модуля і його підключення 304
Запитання для контролю і самоконтролю 309
Завдання для практичного відпрацьовування матеріалу 309
10. Варіанти 311
10.1. Поняття варіантного типу і його подання в пам'яті 311
10.2. Особливості виразів, що використовують варіанти 314
10.3. Варіантні масиви 317
Запитання для контролю і самоконтролю 320
Завдання для практичного відпрацьовування матеріалу 320
11. Елементи сучасного об'єктно-орієнтованого програмування 321
11.1. Визначення класів та основні поняття об'єктно-орієнтованого програмування 321
11.2. Спадкування та перевизначення. Класи і модулі 330
11.3. Віртуальні методи. Конструктори та деструктори 342
11.4. Таблиця віртуальних методів 350
11.5. Динамічні методи 351
11.6. Перевантажувані методи 352
11.7. Абстрактні методи та методи класу 352
11.8. Посилання на класи та вказівники методів 354
11.9. Властивості 361
11.10. Секції класів 369
Запитання для контролю і самоконтролю 370
Завдання для практичного відпрацьовування матеріалу 371
12. Списки 373
12.1. Списки класу TList 373
12.2. Списки класу TStrings 379
12.3. Списки класу TStringList 382
Запитання для контролю і самоконтролю 387
Завдання для практичного відпрацьовування матеріалу 387
13. Обробка особливих (виняткових) ситуацій 389
13.1. Захищені блоки і їх використання як механізму обробки винятків 389
13.2. Використання стандартних класів винятків 393
13.3. Створення власних класів і примусове збудження винятків 403
Запитання для контролю і самоконтролю 407
Завдання для практичного відпрацьовування матеріалу 408

14. Графічні можливості Delphi 409
14.1. Клас TCanvas 409
14.2. Графічні інструменти 419
14.3. Класи для роботи із зображеннями 426
Запитання для контролю і самоконтролю 433
Завдання для практичного відпрацьовування матеріалу 433
15. Поняття компонентів та характеристика їх базових класів 435
15.1. Класи TComponent і TPersistent 435
15.2. Події та їх опрацьовувачі 440
15.3. Клас TControl: загальні властивості та методи візуальних компонентів 441
15.4. Спільні властивості і методи віконних компонентів 451
15.5. Невізуальні та графічні компоненти 460
Запитання для контролю і самоконтролю 461
Завдання для практичного відпрацьовування матеріалу 462
16. Форма та контейнери загального призначення 463
16.1. Форма – компонент TForm 463
16.2. Фрейм – компонент TFrame 478
16.3. Панель – компонент TPanel 480
16.4. Панель групування – компонент TGroupBox 481
16.5. Панель зі скролінгом – компонент TScrollBox 481
16.6. Панель з однобічним прокручуванням – компонент TPageScroller 482
16.7. Панель зі смугами – компонент TCoolBar 483
16.8. Набір закладок – компонент TTabControl 487
16.9. Набір сторінок із закладками – компонент TPageControl 489
16.10. Блокнот із закладками – компонент TNotebook 492
16.11. Крайка – компонент TBevel 492
Запитання для контролю і самоконтролю 493
Завдання для практичного відпрацьовування матеріалу 494
17. Компоненти для введення, виведення та відображення інформації 495
17.1. Компоненти для виведення тексту 495
17.2. Компоненти для введення та виведення тексту 501
17.3. Компоненти для виведення графічної інформації 513
17.4. Таблиці 517
Запитання для контролю і самоконтролю 536
Завдання для практичного відпрацьовування матеріалу 536
18. Компоненти для подачі команд керування 539
18.1. Кнопка – компонент TButton 539
18.2. Графічна кнопка – компонент TBitBtn 540
18.3. Кнопка швидкого виклику – компонент TSpeedButton 543
18.4. Головне меню – компонент TMainMenu 545
18.5. Контекстне меню – компонент TPopupMenu 549
18.6. Спарені кнопки – компонент TSpinButton 550

18.7. Панель інструментів та інструментальні кнопки – компоненти TToolBar та TToolButton 550
18.8. Таймер – компонент TTimer 551
Запитання для контролю і самоконтролю 555
Завдання для практичного відпрацьовування матеріалу 555
19. Компоненти для відображення та керування значеннями величин 557
19.1. Смуга прокручування – компонент TScrollBar 557
19.2. Повзунок – компонент TTrackBar 559
19.3. Лічильник – компонент TUpDown 562
19.4. Редагований рядок з лічильником – компонент TSpinEdit 564
19.5. Індикатор процесу – компонент TProgressBar 564
19.6. Індикатор величини – компонент TGauge 567
Запитання для контролю і самоконтролю 568
Завдання для практичного відпрацьовування матеріалу 568
20. Компоненти вибору та установки 569
20.1. Прапорець – компонент TCheckBox 569
20.2. Перемикач – компонент TRadioButton 571
20.3. Група перемикачів – компонент TRadioGroup 572
20.4. Простий список – компонент TListBox 575
20.5. Комбінований список – компонент TComboBox 578
20.6. Група прапорців – компонент TCheckListBox 579
20.7. Комбінований список з розширеними можливостями – компонент TComboBoxEx 582
20.8. Список вибору кольору – компонент TColorBox 583
20.9. Список зображень – компонент TImageList 584
Запитання для контролю і самоконтролю 585
Завдання для практичного відпрацьовування матеріалу 586
21. Компоненти для організації діалогів 587
21.1. Загальні принципи роботи з компонентами для організації діалогів 587
21.2. Вікно відкриття файлу – компонент TOpenDialog 588
21.3. Вікно збереження файлу – компонент TSaveDialog 595
21.4. Вікна відкриття та збереження зображень – компоненти TOpenPictureDialog і TSavePictureDialog 597
21.5. Вікно вибору шрифту – компонент TFontDialog 597
21.6. Вікно вибору кольору – компонент TColorDialog 600
21.7. Вікно пошуку – компонент TFindDialog 601
21.8. Вікно пошуку та заміни – компонент TReplaceDialog 603
21.9. Діалогові вікна для керування друком – компоненти TPrintDialog, TPageSetupDialog і TPrinterSetupDialog 604
Запитання для контролю і самоконтролю 605
Завдання для практичного відпрацьовування матеріалу 606
Список літератури 607

ПЕРЕДМОВА

Видання даного навчального посібника обумовлено змінами навчальних планів і навчальних програм на кафедрі системного аналізу і управління Національного технічного університету «Харківський політехнічний інститут». Справа в тому, що раніше ази програмування на спеціальності «Соціальна інформатика» у НТУ «ХПІ» студенти починали освоювати, орієнтуючись на мову програмування Turbo Pascal 7.0 з наступним переходом на Delphi. Відповідно до цього і був випущений навчальний посібник з мови Turbo Pascal 7.0 [1]. Але прийшов час, коли кафедра ухвалила рішення про те, щоб здійснювати підготовку студентів, орієнтуючись відразу на середовище візуального програмування, у зв'язку з чим і виникла необхідність у даному виданні.

Посібник містить матеріал, орієнтований на вивчення та освоєння як базової для Delphi мови Object Pascal, так і компонентів, що найчастіше використовуються при написанні нескладних програм. Викладання матеріалу супроводжується прикладами програм, що пройшли налаштування і тестування. Варто врахувати те, що програмні коди не претендують на те, щоб їх можна було назвати як найкращими, – вони лише ілюструють можливості окремих операторів і компонентів Delphi.

Контрольні запитання і завдання для програмування, які вміщені після кожного розділу, дозволяють організувати самотестування і практичне відпрацьовування розглянутого матеріалу. Розділи, що присвячені розгляду компонентів Delphi, можуть бути використані як довідковий матеріал.

Для освоєння матеріалу не потрібні жадних особливих додаткових знань, за винятком хіба що початкових навичок у роботі з файлами в операційній системі Windows.

Виражаю вдячність рецензентам за зауваження, що дозволили внести зміни в текст посібника.

Я вдячний також моїм дружині й дочці за їхнє розуміння і терпіння, які вони проявляли, коли їхній чоловік і батько замість того, щоб приділяти їм більше уваги, займався написанням даного навчального посібника.

1. КОМП'ЮТЕР І ЙОГО СТИСЛА ІСТОРІЯ

1.1. Початкові відомості про апаратне і програмне забезпечення комп'ютерів

Комп'ютери в сучасному світі проникнули в усі сфери діяльності людини. Насамперед, звичайно, сказане стосується наукової діяльності. Це передбачав вже розробник першого обчислювального пристрою, який можна було програмувати, – Чарльз Беббідж. Ось його слова: «Тепер є можливість виконувати весь процес розробки і аналізу за допомогою машин. З появою Аналітичної Машини стало ясно, що вона неминуче стане фактором, що визначає шляхи розвитку науки».

Функціонування комп'ютерів засноване на такому понятті, як програма.

Програмою називається набір інструкцій, які призначені для їхнього виконання комп'ютером.

Множина фізичних складових комп'ютера називається його ***апаратним забезпеченням*** (hardware), а множина програм, використовуваних ним, – ***програмним забезпеченням*** (software).

При розгляді будови комп'ютера найчастіше виділяють п'ять основних складових: пристрій введення, пристрій виведення, центральний процесор (central processing unit – CPU), основну (оперативну) пам'ять і вторинну (зовнішню, допоміжну) пам'ять (рис. 1.1).

Центральний процесор – це пристрій, призначений для виконання різного роду арифметичних і логічних операцій і видачі сигналів керування іншими пристроями комп'ютера.

Основна пам'ять – це пристрій, що служить для оперативного (поточного) зберігання інформації. Ця пам'ять є енергозалежною – інформація у ній зберігається до відключення живлення. Саме в цій пам'яті зберігається виконувана процесором програма, і саме в неї записується опрацьовувана центральним процесором інформація. Центральний проце-

сор й основну пам'ять можна вважати єдиним блоком, у зв'язку з чим повинен бути забезпечений швидкий доступ до основної пам'яті.

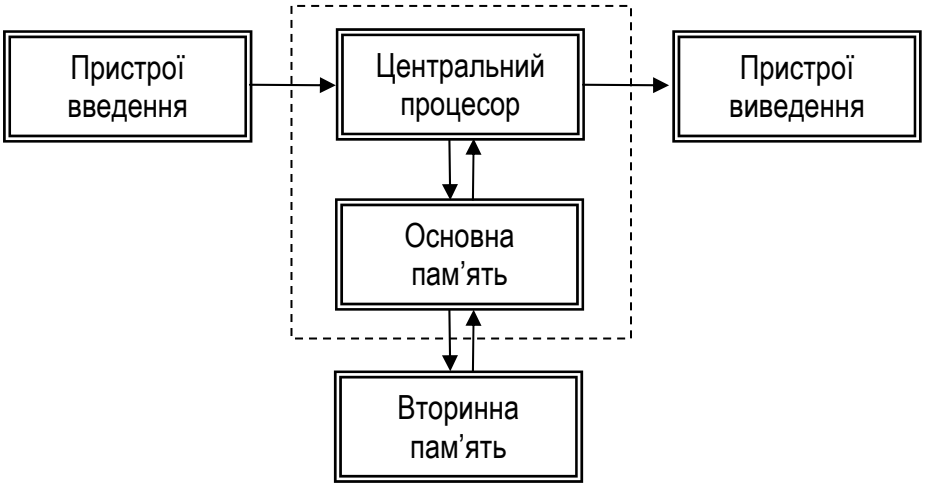


Рис. 1.1. Узагальнена схема комп'ютера

Основна пам'ять являє собою послідовність пронумерованих комірок, називаних ***комірками пам'яті***. Інформація в кожній з них зберігається у вигляді послідовності нулів та одиниць, тобто у двійковій системі числення. Порція інформації, що може набувати лише двох значень – нуль або одиниця, називається двійковою цифрою або ***бітом*** (від англ. bit, binary digit – «двійкова цифра»). Практично в усіх комп'ютерів комірки пам'яті складаються з вісьмох бітів або числа, що кратне восьми. Це можна пояснити наступними причинами. По-перше, число 8 – це ступінь числа 2 і, по-друге, для кодування будь-якого символу достатньо (точніше, раніше було достатньо) восьми бітів ($2^8 = 256$).

Ділянка пам'яті ємністю у вісім бітів називається ***байтом***, і саме цей термін використовується для іменування пронумерованих комірок пам'яті. Число, що ідентифікує кожен байт, називається його ***адресою***. Вміст окремих байтів пам'яті може змінюватися, а їх адреси служать для доступу до них центрального процесора. При цьому в сучасних мовах програмування програмісту найчастіше немає необхідності знати безпосередні адреси комірок пам'яті, що використовуються в його програмі, – він працює не з адресами, а з іменами змінних. Доступ же до потрібних адрес здійснюється автоматично на підставі типів елементів даних, яким відповідають різні системи кодування. У більшості випадків програмісту можна не замислюватися і про те, що дані в комп'ютері подаються у двійковій системі числення. Він може вважати, що в комірках пам'яті зберігаються числа, літери, інша інформація.

Якщо для зберігання порції даних одного байта недостатньо, здійснюється виділення декількох послідовних байтів. У цьому випадку весь

виділений блок пам'яті сприймається як єдине ціле і також розглядається як комірка пам'яті. Під адресою такої комірки розуміють адресу першого її байта. Якщо умовно припустити, що байти основної пам'яті нумеруються, починаючи від 1, то в цьому випадку співвідношення між номерами байтів та адресами комірок пам'яті, у яких зберігаються дані різного розміру, можна проілюструвати за допомогою рис. 1.2.

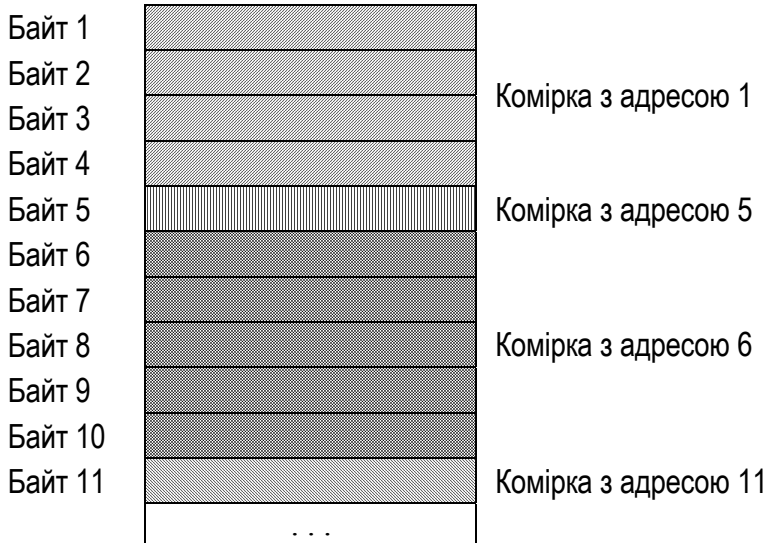


Рис. 1.2. Байти і комірки пам'яті

Пристрій введення – це пристрій, за допомогою якого можна здійснювати введення інформації в комп'ютер. Основними пристроями введення в сучасних комп'ютерах є клавіатура і миша.

Пристрій виведення – це пристрій, призначений для вилучення інформації з комп'ютера. Найпоширенішим пристроєм виведення є монітор, що служить для виведення інформації з метою наступного її візуального сприйняття користувачем. Дуже часто для виведення інформації на папері використовуються принтери.

Найчастіше пристрої введення інформації і пристрої виведення не розділяють і говорять про пристрої введення/виведення інформації. При цьому клавіатуру і монітор поєднують у єдине поняття – ***термінал візуальної інформації*** (video display terminal – VDT).

На відміну від основної пам'яті, що служить для зберігання інформації під час виконання програми, ***вторинна пам'ять*** – це пристрій, що служить для довгострокового (постійного) зберігання інформації з можливістю її (інформації) використання програмами. Одиницею зберігання інформації для користувача в цьому випадку є файл. При цьому інформація зберігається на зовнішніх запам'ятовувальних пристроях, найбільш поширеними з яких є жорсткий диск (вінчестер), дискета (гнучкий диск), лазерний диск, магнітна стрічка, флеш-пам'ять. Такого роду пристрої є знім-

ними і можуть переноситися (разом з інформацією, що зберігається на них) з одного комп'ютера на інший.

Основним елементом програмного забезпечення комп'ютера є операційна система.

Операційна система – це програма, що розподіляє ресурси комп'ютера між різними виконуваними ним програмами. Мабуть, найпоширенішою операційною системою на разі є операційна система Windows – розробка компанії Microsoft. Інші популярні операційними системи – це UNIX, Linux, OS/2, DOS, Macintosh, VMS.

Програма звичайно пишеться ***мовою високого рівня***, однією з яких є мова Delphi. Особливістю мов високого рівня є те, що вони багато в чому схожі на ті мови, якими спілкуються люди. Мови високого рівня містять значно складніші інструкції, ніж ті коди, які можуть бути виконані центральним процесором.

Мову, зрозумілу комп'ютеру, називають ***мовою низького рівня***. Врешті-решт, вихідний (початковий) текст програми повинен бути перекладений на так звану ***машинну мову***, у якій програма подається послідовністю команд, причому ці команди записуються у вигляді двійкових кодів. Перетворення початкового тексту програми, який інакше називається ***кодом програми*** або ***вихідним (початковим) кодом***, у машинний код здійснює спеціальна програма, що називається ***компілятором***.

Компілятор – це особлива програма, для якої як вхідні дані виступають програми, написані мовою високого рівня, і на виході якої також утворюються програми, але вже написані машинною мовою.

Програми, отримані на виході компілятора, називаються ***об'єктними програмами*** або ***об'єктними кодами***. Особливістю таких програм є те, що вони не можуть бути відразу виконані, оскільки до того необхідно здійснити підключення до них деяких службових підпрограм.

Остаточне опрацювання програми, точніше об'єктного коду, здійснює так звана ***програма-компонувальник*** (інакше ***редактор зв'язків***). Саме компонувальник створює ***машинний код***, який і виконується комп'ютером при розв'язанні задачі.

Таким чином, компонувльник поєднує об'єктний код програми з об'єктними кодами службових підпрограм, які використовуються даною програмою. У багатьох випадках (зокрема, це характерно для Delphi) компонувальник викликається автоматично відразу ж по завершенні роботи компілятора.

Процес розв'язання задачі за допомогою комп'ютера ілюструє рис. 1.3. При цьому разом з програмою на вхід комп'ютера повинна надходити ***вхідна інформація*** (***вхідні дані***). Результати роботи програми називають ***вихідними даними***.

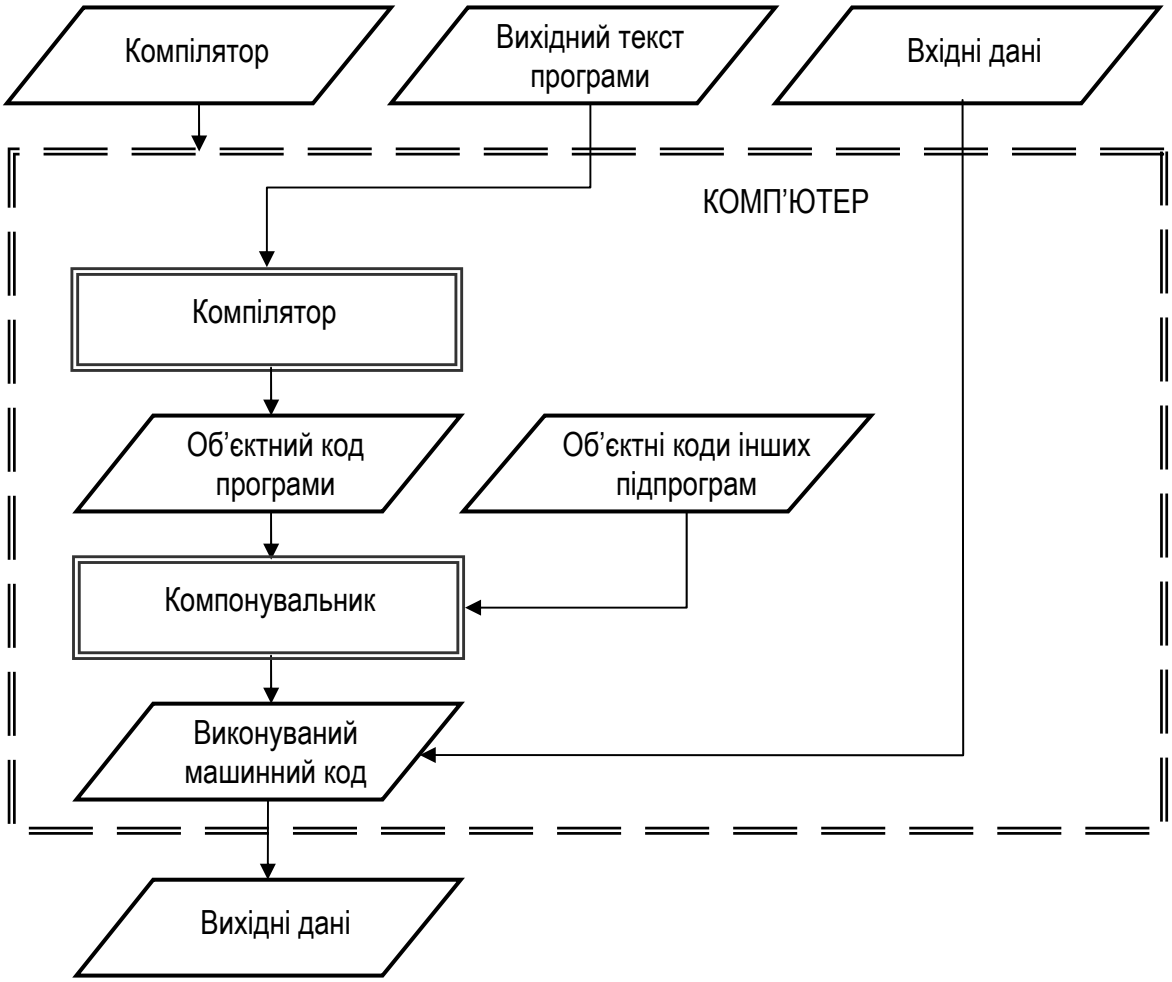


Рис. 1.3. Процес розв'язання задачі

1.2. Алгоритми і розробка програм

Якщо потрібно написати програму для розв'язання за допомогою комп'ютера деякої достатньо складної задачі, більш-менш досвідчений програміст в жодному разі не починає одразу писати текст програми. Перед тим як написати програму, її розробник повинен проаналізувати задачу, виділити основні етапи її розв'язання й, якщо це вбачається за необхідне, записати їх звичною для нього мовою (англійською, українською, російською). Якщо послідовність дій, яку слід виконати, визначена, то написання програми багато в чому стає формальністю, яка зводиться до виконання рутинної роботи. Якщо ж метод розв'язання поставленої задачі не визначений, то тут комп'ютер не допоможе. Про це добре сказала дочка поета Байрона Ада Аугуста Кінг, графиня Лавлейс, яку вважають першою програмісткою в історії обчислювальної техніки, оскільки саме вона займалася розробкою програм для аналітичної машини Беббіджа (щоправда, ця машина так і не була створена). Ось її слова: «Аналітична

машина зовсім не претендує на створення чого б то не було. Вона може виконувати тільки те, що ми їй скажемо, – якщо ми знаємо, як це зробити. Вона може аналізувати, але не здатна висувати гіпотези яких-небудь аналітичних залежностей або законів. Її сфера – зробити більш доступним те, із чим ми вже знайомі».

Кінцева послідовність дій, виконання яких приводить до перетворення початкових даних у шуканий результат, називається ***алгоритмом***.

Частковим випадком алгоритму є програма, написана на деякій мові програмування.

Слово «алгоритм» утворилося від імені перського математика й астронома IX століття аль-Хорезмі, що написав відомий підручник з перетворення чисел і рівнянь. Книга називається «Кітаб аль-джабр валь-мукабала», що можна перекласти як «Книга про відновлення і протиставлення». Від арабського слова «***аль-джабр»***, що зустрічається в назві книги, відбулося слово «***алгебра»***, що, у свою чергу, породило слово «***алгоритм»***.

Для того щоб набір команд міг бути віднесений до розряду алгоритмів, він повинен повно й однозначно задавати, які дії та в якому порядку потрібно виконати. Людина або машина, що виконує алгоритм, точно ідуть за його приписами – і не більше того.

Якщо ж алгоритм визначений недостатньо чітко, то можливі найбільш непередбачені наслідки за аналогією з тим, про що розповідається в наступному анекдоті:

> Коваль учить свого учня, як кувати підкови:
> – Спочатку кладеш її у вогонь, щоб розігрілася. Коли буде червоною, поклади на ковадло, а коли я кивну головою, вдар по ній молотком.
> Учень уважно вислухав майстра й усе зробив, як йому було сказано.
> Від цієї науки майстер лежав тиждень з хворою головою, а потім ще тиждень ходив з перев'язаною.

Несподіваний ефект, що з'являється при видаванні нечітких інструкцій, досить часто називають ***ефектом мавпячої лапи***. Цей ефект описав основоположник кібернетики Норберт Вінер. Суть його можна визначити так: при недостатньо чітко сформульованому алгоритмі програма може повестися найбільш непередбаченим чином, причому за найгіршим з можливих сценаріїв.

Вінер у своїй роботі «Творець і робот»[1], аналізуючи повість англійського письменника початку XX століття Джекобса «Мавпяча лапа», порів-

[1] Винер Н. Творец и робот. Обсуждение некоторых проблем, в которых кибернетика сталкивается с религией. Пер. с англ. – М.: Прогресс, 1966.

нює дії машини, що може навчатися, з магічними діями: магічні дії виконуються буквально, тобто ми одержуємо те, що просили, а не те, що мали на увазі, проте не змогли точно сформулювати[1].

Створення програми є творчим і часто досить складним процесом, для якого не існує повного набору правил, що вказують, як його виконувати.

Умовно цей процес можна розбити на два етапи – етап розробки алгоритму розв'язання задачі та етап його програмної реалізації.

На першому етапі, насамперед, формулюється розв'язувана задача, причому всі формулювання повинні бути повними і точними. Необхідно визначитися з вхідними даними, які мають бути опрацьовані, а також з тим, що має бути отримано в результаті роботи програми. Далі слід написати алгоритм розв'язання задачі, причому його написання здійснюється або природною мовою, або деякою мовою, спеціально розробленою для написання алгоритмів, або за допомогою деяких умовних позначок (зокрема, за допомогою структурних схем). Алгоритм потрібно піддати тестуванню, що виражається в перевірці можливості одержання з його допомогою шуканого розв'язку. Якщо це необхідно, за результатами

---

[1] Вінер посилається на повість англійського письменника Джекобса, у якій «…описується англійська робоча родина, що зібралася до обіду на кухні. Син незабаром іде на фабрику, а старі батьки слухають розповідь свого гостя, старшого сержанта, що повернувся з Індії. Гість розповідає про індійську магію й показує талісман – висушену мавпячу лапу. За його словами, індійський святий наділив цей талісман магічною властивістю виконувати по три бажання кожного з трьох своїх послідовних власників. Гість говорить, що це підходящий випадок випробувати долю. Він не знає перших двох бажань попереднього власника талісмана, але йому відомо, що останнім бажанням його попередника була смерть. Сам він був другим власником, але те, чого він зазнав, занадто страшно переказувати. Гість уже має намір кинути чарівну лапу в камін, але хазяїн вихоплює її й, незважаючи на застереження, просить у неї двісті фунтів стерлінгів.

Через деякий час роздається стукіт у двері. Входить дуже важливий пан, службовець того підприємства, на якому працює син хазяїна. З усією м'якістю, на яку він здатен, пан повідомляє, що в результаті нещасного випадку на фабриці син хазяїна загинув. Не вважаючи себе жодною мірою відповідальним за те, що трапилося, підприємство виражає родині загиблого своє співчуття і просить прийняти допомогу у розмірі двохсот фунтів стерлінгів. Збожеволілі від горя батьки благають, щоб талісман повернув їм сина – і це їхнє друге бажання... Раптово все поринає в лиховісну нічну пітьму, піднімається буря. Знову стукіт у двері. Батьки якимсь чином дізнаються, що це їхній син, але, на жаль, він безтілесний, як примара. Історія кінчається тим, що батьки виражають своє третє й останнє бажання: вони просять, щоб примара сина відійшла.

…Магічне виконання заданого здійснюється найвищою мірою буквально, й ... якщо магія взагалі здатна дарувати що-небудь, то вона дарує саме те, що ви попросили, а не те, що ви мали на увазі, але не зуміли точно сформулювати.

Якщо ви просите двісті фунтів стерлінгів і не обумовлюєте при цьому, що не бажаєте їх одержати ціною життя вашого сина, ви одержите свої двісті фунтів незалежно від того, чи залишиться ваш син у живих, чи помре!

Не виключено, що магія автоматизації й, зокрема, логічні властивості автоматів, що самонавчаються, будуть проявлятися настільки ж буквально».

тестування проводиться уточнення постановки задачі або здійснюється корекція алгоритму.

Другий етап розробки програми у випадку правильного алгоритму зводиться до кодування його обраною мовою програмування з наступним тестуванням отриманої програми. При цьому тестування має бути досить повним з відстеженням усіх можливих окремих випадків початкових даних, з якими може зіткнутися програма при її функціонуванні, і перевіркою правильності одержуваних при цьому результатів. Природна річ, виправлення тільки тих помилок, які виявляє компілятор на етапі опрацювання тексту програми, недостатньо, оскільки як у самій програмі, так і в розробленому алгоритмі можуть зустрітися помилки, які компілятор виявити не в змозі. Якщо в результаті тестування програми виявляються помилки, то доводиться повертатися до фази її кодування, а досить часто і до першого етапу, аж до уточнення постановки задачі.

Процес усунення помилок у програмі називається ***налаштуванням*** (debugging), а самі помилки в програмі досить часто називають ***жучками*** (bugs). Згідно з однією з історій про походження цього терміна, він виник на зорі розвитку комп'ютерної техніки, коли апаратне забезпечення було вкрай чутливим до всякого роду зовнішніх впливів. Одним з програмістів, що працювали під керівництвом професора Говарда Ейкена на першому у світі великому цифровому обчислювальному комп'ютері Harvard Mark I, була Грейс Мюррей Хопер, яка стала надалі контр-адміралом. Під час роботи в реле комп'ютера потрапив тарган і викликав збій у його роботі. Хопер та інші програмісти внесли у вахтовий журнал такий запис про загиблого жука: «First actual case of bug being found» («Перший випадок виявлення жучка»). Під час перерви зайшов професор Ейкен з запитанням про те, що вже полічене. Програмісти відповіли, що вони «усували жучків» (debugging) у комп'ютері. Це перший документований випадок комп'ютерного «жучка», а вахтовий журнал зараз перебуває в експозиції Морського музею в Далгрені (Dahlgren), штат Вірджинія (США).

Серед помилок, які допускаються в програмі, виділяють синтаксичні помилки, помилки часу виконання і логічні помилки.

Синтаксичні помилки – це помилки, пов'язані з порушенням ***синтаксису*** (тобто правил граматики) мови програмування. Прикладом такої помилки, що часто зустрічається у написаних у середовищі Delphi програмах, є пропуск символу «крапка з комою», який є роздільником між операторами. При виявленні синтаксичної помилки компілятор видає ***повідомлення про помилку***, вказуючи її передбачуване місце розташування і пояснюючи можливий її зміст. Звісно, природа помилки може відрізнятися від тієї її інтерпретації, яку робить компілятор, тим більше що одна синтаксична помилка може призвести до того, що компілятор видасть

декілька повідомлень про помилки, зумовлені, проте, наявністю лише першої з них.

Досить часто компілятор виводить ***попереджувальні повідомлення*** про деякі дещо незвичні конструкції у програмі. У багатьох випадках такі повідомлення все-таки свідчать про помилки, у зв'язку з чим на них варто обов'язково звертати увагу і по можливості вносити відповідні виправлення. В ідеальному випадку попереджувальні повідомлення відсутні.

Деякі помилки проявляються тільки під час виконання програми, у зв'язку з чим вони носять відповідну назву – ***помилки часу виконання*** (run-time errors). З появою таких помилок програма завершується аварійно з видачею пояснювального повідомлення. Прикладами таких помилок є ділення на нуль, одержання великих числових значень, які не можуть бути записані в комірку пам'яті (переповнення).

Часто найбільш неприємними помилками є ***логічні помилки*** – помилки в самому алгоритмі, а також помилки, спричинені елементарною неуважністю (наприклад, використання в програмі операції * замість операції +, задавання неправильного числового значення, використання одного імені змінної замість іншого). Такі помилки компілятор найчастіше виявити не може (за винятком випадків, коли вони призводять до порушення синтаксису). Мало того, логічні помилки можуть не проявити себе і під час виконання програми як помилки часу виконання.

Для виявлення логічних помилок здійснюється ***тестування*** програми, що виражається в її запуску з кількома характерними наборами вхідних даних і перевірці відповідних їм вихідних даних. При цьому в жодному разі не можна обмежуватися одноразовою перевіркою програми – повинні бути відслідковані всі окремі випадки вхідних даних, з якими програма може зіткнутися. Тільки після перевірки правильності функціонування програми на багатьох характерних наборах даних можна отримати високу (але не абсолютну) гарантію її правильності.

1.3. Розвиток обчислювальної техніки: події та дати

Розвиток цифрової обчислювальної техніки в сучасному її розумінні, а також науки про принципи її побудови і проектування бурхливими темпами почався з 40-х років XX століття, хоча її історія нараховує кілька століть.

Першим з відомих рисунків обчислювальних пристроїв визнають ескіз тринадцятирозрядного десяткового підсумовувального пристрою на основі коліс з десятьма зубцями, що був зроблений Леонардо да Вінчі в

одному з його щоденників ще в 90-х роках XV – початку XVI століть (рис. 1.4, *а*). Відтворення цього пристрою у XX столітті засвідчило можливість його використання.

У 1623 році німецький учений Вільгельм Шіккард створив свій «годинник для рахунку» – шестирозрядний десятковий обчислювальний пристрій, призначений для додавання і віднімання чисел з можливістю інформування за допомогою дзвоника про переповнення.

Як винахід Леонардо, так і винахід Шіккарда, були загублені й, будучи знайденими тільки в другій половині XX століття, ніяк не позначилися на розвитку обчислювальної техніки (в 1960 році за знайденими кресленнями машину Шіккарда побудували, і вона виявилася працездатною, див. рис. 1.4, *б*).

У 1642 році був створений перший з цифрових обчислювальних пристроїв, що став відомим сучасникам і зробив вплив на розвиток обчислювальної техніки. Його творцем був Блез Паскаль, у майбутньому великий французький учений. Свою обчислювальну машину він назвав «арифметичною машиною» (рис. 1.4, *в*). Машина Паскаля являла собою 6-розрядний десятковий підсумовувальний пристрій, який у подальшому був удосконалений з доведенням кількості розрядів до восьми.

Якщо всі названі вище обчислювальні пристрої призначалися для роботи з десятковими числами, то першим недесятковим підсумовувальним пристроєм була винайдена в Англії сером Семюелом Морлендом недесяткова машина, призначена для роботи з англійською валютою (1667 рік).

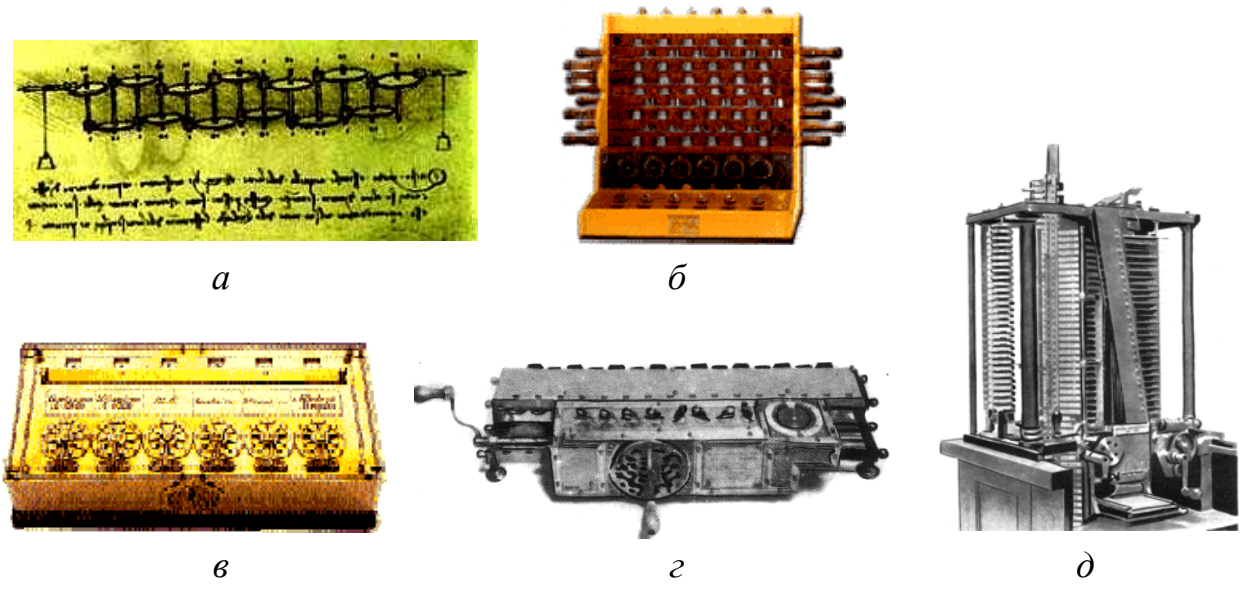
У 1673 році Готфрід Вільгельм Лейбніц створив свій «арифметичний прилад», що міг виконувати всі чотири арифметичні операції над дванадцятирозрядними десятковими числами з результатом до 16 знаків (див. рис. 1.4, *г*). Лейбніц же займався дослідженням властивостей ***двійкової системи числення***, що надалі стала базовою системою числення в обчислювальній техніці.

У 1723 році німецький математик та астроном Християн Людвіг Герстен на основі робіт Лейбніца сконструював свою арифметичну машину. У цій машині була передбачена можливість контролю за правильністю введення даних.

У 1775 і 1777 роках були побудовані аналогічні машині Лейбніца множувальні калькулятори англійця Чарльза Ерла Стенхоупа III і Метьюса Хана з Німеччини, а в 1786 році німецький військовий інженер Йоганн Мюллер висуває ідею «різницевої машини» – спеціалізованого калькулятора для табулювання логарифмів, обчислюваних різницевим методом (проект був забутий через відсутність коштів на його реалізацію). Цей пристрій у XX столітті назвали диференціальним аналізатором.

У 1820 році француз Шарль Ксав'є Тома де Кольмар винайшов ***арифмометр*** – перший калькулятор, що випускався масово. Його виробництво здійснювалося до початку XX століття.

Перший обчислювальний пристрій, який можна було програмувати, також було винайдено в XIX столітті. Цим пристроєм була розроблена англійським математиком і фізиком Чарльзом Беббіджем машина, названа ним аналітичною (її модель наведена на рис. 1.4, *д*).



а *б* *в* *г* *д*

Рис. 1.4. Перші обчислювальні пристрої: *а* – ескіз обчислювального пристрою Леонардо, *б* – машина Шіккарда, *в* – машина Паскаля, *г* – машина Лейбніца, *д* – модель аналітичної машини Беббіджа

При роботі над своєю машиною Беббідж використав ідеї французів Жозефа Жакарда і Гаспара де Проні. Перший з них наприкінці XVIII століття реалізував керування ткацьким верстатом за допомогою перфокарт, а другий запропонував розділяти числові обчислення на три етапи – розробку чисельного методу розв'язання задачі, складання програми послідовності арифметичних дій і проведення самих обчислень шляхом арифметичних операцій над числами відповідно до складеної програми.

До 1833 року Беббідж сконструював механічний пристрій для обчислення таблиць величин, різниці N-го порядку яких є сталими, а в 1834 році почав роботу над універсальним обчислювачем. Архітектура аналітичної машини Беббіджа практично збігається з архітектурою сучасних комп'ютерних систем. Машина Беббіджа мала «склад», або «сховище», для запам'ятовування даних і проміжних результатів (у сучасній термінології – пам'ять) і «млин» (тобто центральний процесор). У машині передбачалася наявність пристрою керування і пристроїв введення та виведення даних. У запам'ятовувальному пристрої передбачалося зберігання 1000 десяткових

чисел, розрядність яких повинна була досягати 50. Для введення даних і програми використовувалися перфокарти – аркуші щільного паперу з отворами, що кодують інформацію. Реалізовувалася машина Беббіджа за допомогою зубчастих коліс і штифтів, а приводити її в дію передбачалося за допомогою парової машини.

Беббідж був першим, хто стверджував, що можна побудувати механічний пристрій, здатний виконувати ***послідовність*** взаємозалежних обчислень. Якщо застосовувати ***різні набори інструкцій***, що задаються машині, то вона зможе служити для ***різних цілей***. Переклавши сказане на сучасну мову, зробимо висновок, що йдеться про складання програми. Відкриттям Беббіджа можна визнати і те, що саме він запропонував здійснювати перетворення інформації в числа з метою їхнього наступного опрацювання машиною.

Програма – це набір правил, за допомогою яких машину інструктують, як вирішувати те або інше завдання. Для введення таких інструкцій звичайна людська мова не підходить, і повинен бути розроблений зовсім новий тип мови. Беббідж винайшов таку мову з цифр, літер, стрілок та інших символів, яка дозволяла програмувати аналітичну машину довгими серіями умовних інструкцій, що, у свою чергу, дозволило б машині реагувати на зміну ситуації.

Як про це вже говорилося вище, багаторічну допомогу в його роботі над аналітичною машиною Беббіджу надавала Ада Аугуста Кінг (уроджена Байрон), графиня Лавлейс. Перекладаючи на прохання Беббіджа одну зі статей, вона додала до перекладу коментарі, які дозволили в майбутньому назвати її першим програмістом планети (у її честь була названа мова програмування Ada). У своїх коментарях Ада Лавлейс повідомила Беббіджу, що склала ***план операцій*** для аналітичної машини, за допомогою яких можна розв'язати рівняння Бернуллі.

У матеріалах Беббіджа і коментарях Лавлейс «накреслені» такі поняття, як ***підпрограма*** і ***бібліотека підпрограм***, ***модифікація команд*** та ***індексний регістр***, які стали вживатися тільки в 50-х роках XX сторіччя. Саме Беббідж увів термін «***бібліотека***», а терміни «***робоча комірка***» і «***цикл***» запропонувала Ада Лавлейс. Вона ж продумала ідею Лейбніца про ***двійкову систему числення*** далі і виявила, що ця система числення дуже практична для впровадження на обчислювальних машинах.

Безпосередньо машина Беббіджа ніколи не була реалізована через її складність для техніки того часу (був побудований прототип тільки одного з її сегментів, що міг оперувати шестирозрядними числами та диференціалами другого порядку, а в 1906 році син Чарльза Беббіджа Генрі побудував процесор батьківської аналітичної машини). Але вже в 1834 році житель Стокгольма Джордж Шойтц, прочитавши стислий опис проекту

Беббіджа, виконав з дерева модель диференціального аналізатора, а в 1853 році він же реалізував перший повнорозмірний диференціальний аналізатор, що працював з п'ятнадцятирозрядними числами та диференціалами четвертого порядку і виводив результати на друкувальну матрицю за принципом Беббіджа. Трохи пізніше Лондонською фірмою Brian Donkin була побудована друга така машина, яку купив британський уряд і яка тривалий час експлуатувалася.

У 1854 році в історію математики й обчислювальної техніки ввійшов математик Джордж Буль, який створив алгебру логіки. Він увів операції (команди керування) І, АБО, НЕ, що здійснюють зв'язки в логічному виразі та дають можливість формувати нові вирази. Цю формальну логіку називають алгеброю Буля (булевою алгеброю), вона і є основою електронного опрацювання даних, будучи пов'язаною з двійковим поданням інформації.

У 1870 році англійський математик Вільям Стенлі Джевонс сконструював «логічну машину», що дозволяла механізувати найпростіші логічні висновки. У 1878 році житель Нью-Йорка Рамон Веріа винайшов калькулятор з убудованою таблицею множення, а в 1885 році став масово випускатися більш компактний, ніж арифмометр множильний калькулятор (його одночасно і незалежно один від одного винайшли американець Френк Болдуїн і швед з Росії Т. Одднер).

У 1886 році Дорр Фелт створив свій Comptometer – перший калькулятор, у якому значення вводилися шляхом натискання клавіш, а в 1889 році він же винайшов перший настільний друкувальний калькулятор.

В історії обчислювальної техніки досить помітним є 1890 рік, коли вперше була здійснена обробка результатів загальноамериканського перепису населення за допомогою обчислювальних машин – перфокарткових табуляторів Германа Голеріта, причому інформація з перфокарт уперше зчитувалася за допомогою електричних машин. У 1896 році була заснована компанія Forms Tabulating Company – одна з перших фірм з виробництва рахункових машин (у 1911 році ця компанія була перетворена в компанію Computing-Tabulating-Recording Company, яка у 1924 році була перейменована в International Business Machines – IBM).

У 1892 році Вільям С. Барроуз створив машину, яка була аналогом машини Фелта, але більш надійною. З цієї машини почалася індустрія офісних калькуляторів.

У 1893 році про роботу Джевонса стало відомо в Росії, коли професор університету в Одесі І. Слешинський опублікував статтю «Логічна машина Джевонса» («Вестник опытной физики и элементарной математики», 1893 р., № 7). Першими розроблювачами логічних машин у дореволюційній Росії стали Павло Дмитрович Хрущов та Олександр Миколайович Щукарьов, які працювали в навчальних закладах України. Спочатку

машина Джевонса була відтворена професором П. Д. Хрущовим в Одесі у 1907 році. Пізніше професором Харківського технологічного інституту О. М. Щукарьовим (він почав працювати в Харкові з 1911 року) ця машина з низкою удосконалень була сконструйована заново. Машини П. Д. Хрущова та О. М. Щукарьова не збереглися.

На початку XX століття були сконструйовані машини для факторизації цілих чисел – машина Юджина Кариссана (1920 рік), що являла собою конструкцію з 14 з'єднаних між собою металевих кілець, і машина Дерика Генрі Леммера (1926 рік), заснована на використанні велосипедних ланцюгів, а пізніше кіноплівки. У 1931–1932 роках у своєму двійковому цифровому лічильнику використав заповнені газом трубки Е. Вінн-Вільямс, а в 1932 році оптичним зчитувачем спорядив свою машину Леммер, що дозволило довести її швидкодію до 5000 операцій у секунду.

Помітним в історії обчислювальної техніки є 1935 рік, коли корпорація IBM презентувала побудовану на реле і здатну виконувати операцію множення за 1 секунду машину IBM 601. Більше півтори тисячі машин цієї моделі використовували вчені та бізнесмени того часу.

Важливу роль у розвитку обчислювальної техніки відіграв німецький учений Конрад Цузе, який ще студентом у 1934 році зайнявся створенням цифрової обчислювальної машини з програмним керуванням і з використанням двійкової системи числення. У 1937 році він записав у своєму щоденнику основну ідею концепції «збереженої програми», а в 1938 році завершив роботу над прототипом електромеханічного двійкового програмувального калькулятора V1, перейменованого пізніше в Z1. Відмінною рисою цієї машини було те, що вона могла працювати з від'ємними та дійсними числами. Машина мала пам'ять на 64 числа й базувалася на системі важелів. Надалі Цузе став одним з провідних конструкторів обчислювальних машин у Німеччині, створивши в 1939–1940 роках машину V2 (Z2) з арифметичним пристроєм на базі релейної логіки і пристроєм введення інформації, який базувався на використанні перфорованої фотоплівки, а в 1941 році – релейний калькулятор Z3, що міг програмуватися і працювати з дійсними числами з семибітовою експонентою і 14-бітною мантисою. У 1945 році він майже завершив роботу над обчислювальною машиною Z4 і розробив мову програмування «Планкалькуль». Робота над машиною Z4 була завершена в 1950 році в Цюріху, причому їх було продано більше 300 штук.

У 1937 році британський математик Алан М. Тьюрінг розвив ідеї Беббіджа та Ади Лавлейс, описавши у своїй роботі теоретичний комп'ютер, відомий зараз як «машина Тьюрінга». Роком пізніше, ще будучи студентом, у своїй магістерській дисертації Клод Шеннон довів, що машина, яка виконує логічні інструкції разом з інструкціями чисто обчислю-

вального характеру, може маніпулювати інформацією як такою, тобто поширив сферу дії обчислювальних машин і на нематематичну форму даних. Шеннон розглянув, як за допомогою електричних кіл обчислювальна машина виконує логічні операції, довівши, що, маніпулюючи всього лише двома станами («коло замкнуте» – одиниця, істина та «коло розімкнуте» – нуль, неправда), можна описувати процес розв'язання будь-якого завдання.

У 1937 році гарвардський математик Говард Ейкен запропонував проект розробки великої рахункової машини, що базується на електромеханічних реле. Проектування цієї машини почалося командою розроблювачів компанії IBM під керівництвом Ейкена в 1939 році, а в січні 1943 року ця машина, що одержала назву MARK I, була побудована (офіційно вона була передана Гарвардському університету в 1944 році). Цей перший з відомих програмувальних калькуляторів називався також Harvard Mark I.

У 1947 році Ейкен і його команда здійснили вдосконалення раніше розробленої машини, завершивши створення Harvard Mark II, а в 1949 році Ейкен завершив роботу над Harvard Mark III. Основним нововведенням було роздільне зберігання даних та інструкцій.

Перша машина, що базується на електронних лампах, була розроблена в США Джоном Вінсентом Атанасовим і Кліффордом Е. Беррі в 1939 році. Машина одержала назву ABC (Atanasoff Berry Computer), але не була запатентована. Початок її розробки відносять до 1937 року, коли Атанасов сформулював основну концепцію своєї обчислювальної машини:

- машина повинна використовувати досягнення електроніки;
- її робота повинна ґрунтуватися не на десятковій, а на двійковій системі числення;
- основою запам'ятовувального пристрою повинні бути конденсатори;
- розрахунки повинні проводитися на основі логічних, а не математичних обчислень.

У 1942–1943 роках у Великобританії за участю Алана Тьюрінга була створена обчислювальна машина Colossus, побудована на 2000 електронних ламп.

У листопаді 1945 року американські вчені Джон Мочлі та Преспер Еккерт разом зі своєю командою завершили роботу над створенням електронної обчислювальної машини ENIAC (Electronic Numerical Integrator and Calculator). Публічна презентація машини була проведена в 1946 році. Це була величезна споруда (більше 30 м у довжину, об'єм – 85 м^3, вага – 30 т), що складалася з 40 панелей, які містили понад 18000 електронних ламп і 1500 реле. Машина споживала близько 150 кВт енергії – потужність,

достатня для невеликого заводу. Обчислювальна машина ENIAC складала числа за 0,2 мс, а множила – за 2,8 мс, тобто у тисячу разів швидше, ніж релейні машини. Введення чисел у машину проводилося за допомогою перфокарт, а програмне керування послідовністю виконання операцій здійснювалося, як у лічильно-аналітичних машинах, за допомогою штекерів і складальних полів. Тут все-таки слід зазначити, що в 1973 році суд віддав пріоритет у створенні електронної обчислювальної машини Атанасову, оскільки Мочлі та Еккерт користувалися його розробками.

Джон Мочлі та Преспер Еккерт є розроблювачами і першого американського серійного комп'ютера UNIVAC (Universal Automatic Computer). Цей комп'ютер був пущений в експлуатацію навесні 1951 року.

У 1945 році Джон фон Нейман описав будову майбутнього комп'ютера EDVAC (Electronic Discrete Variable Automatic Computer), давши детальне визначення концепції збереженої програми. До цього часу введення програми здійснювалося за допомогою спеціальних перемикачів. Зі згаданої ж роботи почалася так звана «архітектура фон Неймана», один з основних принципів якої полягає в тому, що в комп'ютері не доведеться змінювати підключення проводів, якщо всі команди будуть зберігатися в його пам'яті. Повністю робота над комп'ютером EDVAC завершилася в 1952 році. Ця машина мала пам'ять на 1024 44-бітні слова і тактову частоту 1 МГц. Операція додавання на ній виконувалася за 0,864 мс, а операція множення – за 2,9 мс.

Ще раніше (в 1949 році) Моріс Уїлкс і його команда з Кембриджу завершили EDSAC (Electronic Delay Storage Automatic Calculator) – комп'ютер, що повністю відповідає архітектурі фон Неймана. Він мав обсяг пам'яті, що дозволяв зберігати 512 чисел по 17 двійкових розрядів у кожному, додавання виконувалося за 0,07 мс, множення – за 8,5 мс. Для введення даних використовувалася перфострічка, а виведення здійснювалося на друкарську машинку.

У 1951 році Грейс Мюррей Хопер винайшла компілятор, а вже в 1952 році вона написала перший компілятор А-0.

Перший електронний комп'ютер фірми IBM був розроблений у 1952 році, коли вона випустила свою промислову обчислювальну машину IBM 701, яка базувалася на 4000 електронних лампах та 12000 германієвих діодах. У машині IBM 704, яка була вдосконаленим варіантом IBM 701, використовувалися індексні регістри. Дані у цій машині подавалися у формі з плаваючою комою.

Першою електронною обчислювальною машиною (ЕОМ), створеною у нас у країні, визнають МЭСМ (від рос. Малая Электронная Счетная Машина – Мала Електронна Рахункова Машина). Розроблена в 1951 році в Києві під керівництвом С. А. Лебедєва, ця ЕОМ була тоді *самою продук-*

тивною машиною в Європі (6–10 тисяч операцій у секунду) і однією з кращих у світі.

У тому ж 1951 році в Інституті точної механіки та обчислювальної техніки (м. Москва) була спроектована машина БЭСМ (від рос. Большая Электронная Счётная Машина – Велика Електронна Рахункова Машина), що мала швидкодію 800 операцій у секунду. Серійно в 1956 році стала випускатися аналогічна машина, що називалася БЭСМ-2.

У 1951 році під керівництвом І. С. Брука був побудований макет невеликої лампової ЕОМ під назвою М-1, а в 1952 році була побудована ЕОМ М-2 з середньою продуктивністю 2000 операцій у секунду.

У 1952 році під керівництвом С. А. Лебедєва (але вже в Москві) була побудована ЕОМ М-20, швидкодія якої становила до 20000 операцій у секунду. Серійно ця ЕОМ почала вироблятися в 1959 році, а її архітектура надалі реалізовувалася в машинах БЭСМ-3М, БЭСМ-4, М-220, М-222.

У той же час у Мінську почалися розробка і випуск машин сімейства «Мінськ 1» (Мінськ-1, Мінськ-11, Мінськ-12, Мінськ-14), середня продуктивність яких становила 2–3 тисячі операцій у секунду.

У 1958 р. під керівництвом В. М. Глушкова в Інституті кібернетики АН України була створена обчислювальна машина «Київ», що мала продуктивність від 6 до 10 тисяч операцій у секунду.

З лампових обчислювальних машин і почалася ера комп'ютерів.

Приблизно з 1959 року елементною базою ЕОМ стали транзистори, що дозволили здешевити їх, зменшити габарити, збільшити обсяг пам'яті та істотно підвищити їхню швидкодію (до 1 млн. операцій у секунду). Прикладами транзисторних комп'ютерів можуть бути «Стретч» (Велика Британія), «Атлас» (США), БЭСМ-6 (СРСР). У 1965 році фірма Digital Equipment випустила перший міні-комп'ютер PDP-8 розміром з холодильник і вартістю всього 20000 доларів.

Знаковим в історії розвитку комп'ютерів є 7 квітня 1964 року, коли корпорація IBM презентувала свою нову розробку – IBM System/360 (S/360). Обчислювальні машини серії IBM/360 (а з 1970 року нової серії IBM/370) відкрили еру уніфікації, сумісності й стандартизації комп'ютерів.

З 1968 року знову почалося змінювання елементної бази комп'ютерів – на зміну транзисторам прийшли інтегральні схеми, які ще більш здешевили комп'ютери і збільшили їхні можливості.

З появою комп'ютерів, що базуються на надвеликих інтегральних схемах (приблизно з 1973–1975 років), почалася ера мікро-ЕОМ і персональних комп'ютерів. Їхня швидкодія виявилася в 10 разів вищою, ніж в ЕОМ на інтегральних схемах, у 1000 разів вищою порівняно зі швидкодією транзисторних ЕОМ, та в 100000 разів вищою від швидкодії лампових ЕОМ.

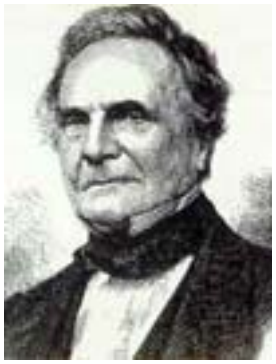

Леонардо да Вінчі


Вільгельм Шіккард


Блез Паскаль


Вільгельм Лейбніц


Джордж Буль


Чарльз Беббідж


Ада Лавлейс


О. М. Щукарьов


Алан Тьюрінг


Джон фон Нейман


Конрад Цузе


Говард Ейкен


Моріс Уїлкс


Джон Моучлі


Проспер Еккерт


С. А. Лебедєв

Рис. 1.5. Історія обчислювальної техніки в обличчях

Перший персональний комп’ютер Alto презентувала в 1973 році компанія Xerox. Він був створений за проектом інженера Алана Кея. В Alto

уперше застосовано принцип виведення програм і файлів на екран у вигляді «вікон».

У 1976 році молоді американці Стів Джобс і Стівен Возняк організували підприємство з виготовлення персональних комп'ютерів «Apple» («Яблуко»), призначених для великого кола непрофесійних користувачів, а в 1981 році зі своїми професійними персональними комп'ютерами IBM PC вийшла фірма IBM, яка займала до цього провідне положення по випуску великих ЕОМ.

З початку 90-х років XX сторіччя термін електронна «обчислювальна машина (ЕОМ)» був практично витіснений з більшості мов словом комп'ютер.

1.4. Розвиток мов програмування

Як про це говорилося в попередньому підрозділі, з розвитком комп'ютерної техніки з'явилася машинна мова, на якій писалися перші програми. За допомогою машинної мови програміст міг задавати команди, оперуючи комірками пам'яті, повністю використовуючи можливості машини. Кожна команда при цьому становить одну елементарну дію для процесора. Однак використання комп'ютерів на рівні машинної мови є досить важкою справою, особливо це стосується введення-виведення. При цьому різні моделі процесорів мають свої набори машинних команд.

Складності, що виникають при програмуванні на машинних мовах, привели до їхньої еволюції убік символічного (мнемонічного) кодування машинних команд, результатом чого стала поява близьких до машинних машинно-орієнтованих мов, що отримали назву асемблерів. Наслідком появи асемблерів, стало те, що відпала необхідність у кодуванні команд на апаратному рівні, що значно полегшило процес програмування. У цілому ж почалася еволюція машинних мов у напрямку природної мови.

У 1954 році в корпорації IBM група розроблювачів на чолі з Джоном Бекусом створила першу мову програмування високого рівня, що отримала назву Fortran (від FORmula TRANslator – перекладач формул). Вона імітувала природну мову, вживаючи деякі слова розмовної англійської мови та загальноприйняті математичні символи. Мова виявилася зручною для розв'язання наукових (читай – математичних) задач, але через відсутність засобів, що дозволяють структурувати програми, використання її для написання великих програм було досить складним. У 1958 році з'явилася друга версія цієї мови (Fortran-II). Однак найбільшою популярністю, мабуть, користувалася мова (Fortran-IV), розроблена в 1964 році. Зараз існують сучасні версії мови Fortran.

Значно зручнішою виявилася мова Algol (від ALGOrithmic Language – алгоритмічна мова). Перша її версія з'явилася в 1958 році (Algol-58), а в 1960 році була розроблена мова програмування Algol-60. Створена групою програмістів на чолі з Петером Науром, ця мова породила групу так званих алголоподібних мов, найяскравішим представником яких є мова Pascal, яку можна назвати праматір'ю Delphi. Мова Algol-60 одержала широке практичне застосування, чого не скажеш про її версію, створену в 1968 році (Algol-68).

У 1960 році була створена мова програмування Cobol (від COmmon Business Oriented Language – ділова орієнтована мова), яка мала значне поширення при рішенні комерційних (економічних) задач.

У 1963 році в Дартмутському коледжі була розроблена мова програмування BASIC (Beginners' All-Purpose Symbolic Instruction Code – багатоцільова мова символічних інструкцій для початківців). Вона розроблялася як засіб навчання, але набула широкого застосування і як звичайна мова програмування. Відтоді розроблено багато версій BASIC. Насамперед, варто сказати про мову Microsoft Visual Basic, що, будучи самостійною мовою програмування, є при цьому і засобом написання макросів у Windows.

У 1964–1965 роках корпорація IBM створила мову PL/1 (від Programming Language One – мова програмування 1), що замислювалася як універсальна мова для розв'язання наукових та економічних задач, яка повинна була замінити Cobol і Fortran. Саме в цій мові вперше з'явилося опрацювання виняткових ситуацій. Дуже великого поширення ця мова не одержала, хоча для машин серії IBM-360 та їхніх аналогів, що вироблялися в колишньому Радянському Союзі, вона був досить характерною.

У міру розвитку обчислювальної техніки необхідність у залученні до процесу написання програм широкого кола фахівців зумовила розробку мови «для навчання програмуванню як систематичній дисципліні». Як таку мову в 1970 році професором Федерального технологічного інституту в Цюріху Ніколаусом Віртом було запропоновано мову Pascal, що стала першою широко розповсюдженою мовою структурного програмування. Розвиток й удосконалення мови привело до розробки в 1979 році її стандартної версії, а також великої кількості діалектів і прикладних бібліотек. Істотний внесок у поширення мови серед найбільшого кола користувачів забезпечила фірма Borland International, яка розширила стандарт мови Pascal і спорядила його потужним компілятором, фактично створивши нову версію мови, що отримала назву Turbo Pascal. У 1985 році ця фірма запропонувала свою версію мови для персональних ЕОМ (версія Turbo Pascal 3.0). Розроблена пізніше версія Turbo Pascal 4.0 характерна тим, що в ній мова програмування була поєднана з текстовим редактором.

Надалі мова удосконалювалася відповідно до вимог сучасного моменту і тенденцій розвитку програмування. Донедавна об'єктно-орієнтована мова програмування Turbo Pascal 7.0 залишалася однією з найпоширеніших мов, поєднуючи в собі великі можливості та зручність інтерфейсу користувача. Її основні конструкції включили в себе такі мови, як Object Pascal і Delphi. Відзначимо, що Pascal є однією з мов програмування, яка використовується на олімпіадах з програмування, в тому числі міжнародних.

У 1972–1973 роках Керніган і Рітчі розробили мову програмування С, що створювалася спочатку для розробки операційної системи UNIX і (після її розвитку) стала однією з основних мов професійного програмування.

До кінця 80-х років минулого століття програми розглядалися як послідовності процедур (або функцій), що виконували певні дії над даними. Процедура являє собою набір певних команд, виконуваних одна за одною. Дані були відділені від процедур, і при складанні програми головним було відстеження того, яка процедура здійснює виклик іншої процедури (якої саме?) і які дані при цьому змінюються. Для внесення ясності в цю потенційно заплутану ситуацію були розроблені принципи структурованого програмування, основна ідея якого полягає в поданні програми у вигляді набору задач. Будь-яка задача, що занадто складна для простого опису, повинна бути розділена на декілька дрібніших складових задач, і цей розподіл необхідно продовжувати доти, доки задачі не стануть досить простими для розуміння. Структуроване програмування і зараз залишається досить успішним способом вирішення складних проблем. Однак з часом стали занадто очевидними деякі недоліки структурованого програмування.

По-перше, дані (наприклад, характеристики деякого переміщуваного об'єкта) і дії над ними (наприклад, змінювання траєкторії прямування) не розглядалися як єдине ціле. Процедурне програмування, навпаки, відокремлювало структури даних від процедур, які маніпулювали цими даними. По-друге, програмісти помітили, що вони постійно перевинаходять нові рішення старих проблем. Бажання мати можливість багаторазового використання рутинних блоків, що повторюються в багатьох програмах, є цілком природним.

Суть об'єктно-орієнтованого програмування полягає в тому, щоб оперувати даними і тими процедурами, які виконують дії над цими даними, як з єдиним об'єктом, тобто самодостатнім елементом, що у чомусь ідентичний іншим таким же об'єктам, але в той же час відрізняється від них певними унікальними властивостями.

У 1983 році Б'ярн Страуструп завершив розробку мови С++ як мови об'єктно-орієнтованого програмування (першою мовою об'єктно-орієнтованого програмування була мова Simula, розроблена ще в 1967 році). Спо-

чатку вона мала назву «С з класами» і розроблялась з 1979 року. Як і інші мови, що одержали значне поширення, С++ має кілька версій. Насамперед, йдеться про версії С++ Builder корпорації Borland і Microsoft Visual C++.

Оскільки спочатку Pascal не містив засоби для розбивки програми на модулі, а також засоби об'єктно-орієнтованого програмування, Н. Вірт в 1978 році запропонував мову Modula-2, у якій ідея модуля стала однією з ключових концепцій. Розроблена в 1986–1988 роках мова Modula-3 уже містила об'єктно-орієнтовані риси.

У 1983 була оприлюднена універсальна мова програмування Ada, у якій підтримуються такі аспекти програмування, як паралелізм та опрацювання винятків, тобто те, що звичайно покладалося на операційну систему. В 1995 році було прийнято стандарт мови Ada 95, що, розвинувши попередню версію, додав до неї об'єктно-орієнтованість. Мова Ada розроблялася на замовлення міністерства оборони США і не здобула значного поширення через складність її конструкцій.

У 1995 році в корпорації Sun Microsystems Кеном Арнольдом і Джеймсом Гослінгом була розроблена мова Java, що успадковувала синтаксис С і С++.

У 2000 році корпорацією Microsoft було анонсовано першу версію мови С#, що розглядалася як альтернатива мові Java. Остаточно вона вийшла у лютому 2002 року. Орієнтована С#, в основному, на розробку багатокомпонентних Internet-додатків. Для опису складного поводження web-сторінок компанією Netscape Communications створено мову JavaScript, що спочатку мала назву LiveScript. Як альтернатива їй компанією Microsoft була розроблена мова VBScript, що є однією з версій мови Visual Basic.

У цілому ж кількість мов програмування дуже велика. Існує багато мов, що мають спеціальну спрямованість, – мови опрацювання рядків (SNOBOL, Icon, Perl), мови виконання операцій над множинами (SETL), мови для опрацювання списків (Lisp, Planner, Scheme, Common Lisp), мови для адміністрування операційних систем (Perl, Python), мови для багатопроцесорних систем (наприклад, мова Occam або модель паралельних обчислень Linda, що може бути додана в будь-яку іншу мову програмування), мови логічного програмування (Prolog, Parlog, Delta Prolog), мови управління базами даних (FoxPro, SQL, Microsoft SQL Server, Oracle) і т. д.

З появою операційної системи Windows мови програмування багато в чому сталі розвиватися убік візуального програмування. Відразу ж за появою Windows 3.1 фірма Borland випустила мову програмування Turbo Pascal for Windows, а в 1992 році її вдосконалену версію – Borland Pascal with Objects 7.0. Ці мови ще не були засобами візуального програмування, але вже були орієнтовані на створення програм, що працюють під керуванням Windows.

Першим візуальним середовищем програмування був Visual Basic, випущений корпорацією Microsoft в 1993 році, після чого в 1995 році фірма Borland презентувала першу версію Delphi. Відтоді відбулися істотні зміни в апаратному забезпеченні комп'ютерів, що зумовило суттєві зміни як операційних систем, так і засобів програмування під них. Були розроблені нові версії Delphi, також одержали свій розвиток інші мови програмування, зорієнтувавшись на програмування під 32-розрядні графічні операційні системи. Такими засобами програмування є версії Microsoft Visual Basic, Microsoft Visual C++, Borland C++ Builder, C#, Java Builder, Microsoft Visual FoxPro, Microsoft SQL Server і т. д. Коли йдеться про останні версії цих систем програмування, вже говорять не про середовище візуального програмування, а про мову, оскільки середовище і базова мова програмування, по суті, уже злилися в єдине ціле.

1.5. Еволюція Delphi

Свою назву система програмування Delphi отримала на честь давньогрецького міста Дельфи, у якому знаходилося головне святилище бога мудрості Аполлона. Його жриці-сивіли вирікали пророцтва бажаючим взнати свою долю. Керівник дослідницької групи по розробці системи Delphi Чак Язджевські відзначив, що назву Delphi запропонував Денні Торп під час однієї з мозкових атак: «Ми бажали, щоб в мені системи відбивалися унікальні здібності продукту до роботи з базами даних, і Delphi як не можна краще перегукується з таким заслуженим ім'ям в цій галузі, як Oracle, принаймні, для тих, кому сполучення "Дельфійський Оракул" про що-небудь говорить».

Як це було зазначене вище, ***перша*** версія Delphi з'явилася в 1995 році. У неї була впроваджена запозичена з C++ модель об'єктів, що відрізняється від тієї, котра була в мові Turbo Pascal 7.0. Як і в C++, цей тип даних назвали класом. Особливостями мови Object Pascal, що стала базовою для Delphi, були такі: практично повністю зняті обмеження на тип значення, що повертає функцією; повернення функцією результату здійснюється через внутрішню змінну `Result`, що спеціально не повинна описуватися; уведено запозичений з C++ механізм опрацювання винятків; уведено відкриті масиви та їхні конструктори. Перша версія Delphi була єдиною версією, що працювала під керуванням Windows 3.1.

У 1996 році була розроблена ***друга*** версія Delphi, що вже призначалася для роботи під керуванням 32-розрядних операційних систем Windows. Її особливості: уведено 2-байтові символи і рядки з таких символів; застосовано новий формат для звичайних рядків; упроваджено типи даних

`Variant` і `Currency`; вжито табличні набори даних і механізм фільтрації; розширено палітру компонентів; додано Репозиторій Об'єктів; розширено перелік утиліт.

У червні 1997 року вийшла ***третя*** версія Delphi, у якій без змін в базовій мові значно розширилися підтримувані технології, поліпшилися властивості редактора коду, розширилася палітра компонентів.

Четверта версія Delphi (липень 1998 року) характеризувалася істотними змінами в базовій мові, а саме, введенням динамічних масивів, замовчуваних параметрів підпрограм, перевантажуваних методів, а також нових типів даних – `Int64, Real48, Real` (аналог типу `Double`). Відбулися також зміни в кодовому редакторі, поліпшилася підтримка баз даних, уведено механізм «причалювання» і механізм «дій», у черговий раз розширилася палітра компонентів.

У ***п'ятій*** версії (1999 рік) внесли істотні зміни в середовище розроблювача, зміни і доповнення в палітру компонентів, а також внесли інші нововведення, зокрема, підтримку технології ADO (Active Data Objects).

Шоста версія (2001 рік) відрізняється, насамперед, тим, що вона здатна створювати програми, які можуть працювати як під Windows, так і під Linux. Крім того, у цій версії були розширенні можливості для підтримки Web-програмування, а також для забезпечення прискореного зв'язку клієнтських місць з деякими серверами баз даних.

Сьома версія, що вийшла в 2002 році, майже повністю відповідає щодо мови та інтегрованого середовища Delphi 6 і характеризується введенням великої кількості технологій, що полегшують створення прикладних програм для баз даних та Internet.

Наприкінці 2002 року корпорацією Borland була анонсована ***восьма*** версія Delphi як новий засіб розробки .NET-додатків, що одержала назву Borland Delphi 8 для Microsoft .NET Framework. Призначення цього продукту – надати користувачам Delphi можливість створювати прикладні програми для платформи .NET Framework, а також переносити на цю платформу успадкований код, створений за допомогою попередніх версій Delphi для 32-розрядних версій Windows. Порівняно з попередніми версіями, що передбачають «класичний» варіант розміщення інструментів середовища з «плаваючим» дизайнером форм і модулів даних, тут було впроваджено нову «багатосторінкову» організацію середовища розробки.

Дев'ята версія Delphi одержала свою назву за роком її випуску – Delphi 2005. У цій версії можливе використання як «класичного», так і «багатосторінкового» середовища розроблювача. Саме середовище розроблювача поповнилося новими інструментами (інструменти для імпорту компонентів .NET у проекти, призначені для платформи Win32, засоби моделювання додатків і т. д.), а також змінився механізм виведення

інтерактивної довідки. До складу цього продукту входять три компілятори – Delphi для платформи Win32, а також Delphi і C# для платформи .NET. Безпосередньо в мові Delphi з'явилася синтаксична конструкція **`for..in..do`**, аналогічна конструкції **`for each`**, відомій користувачам Visual Basic. У цій версії зазнав зміни засоби налаштування і рефакторингу (автоматичного перетворення коду при змінюванні імен змінних класів, параметрів методів і т. д.), засоби створення мережевих прикладних програм і баз даних. Крім того, у середовище Delphi було вбудовано ядро засобу UML-моделювання Borland Together.

У 2006 році була розроблена ***десята*** версія Delphi, що також отримала назву за роком її випуску (Delphi 2006) і була розвитком Delphi 2005.

У 2006 році компанія Borland відокремила свій підрозділ створення засобів розробки у окрему компанію – Code Gear. Ці компанія на початку 2007 року презентувала ***одинадцяту*** версію Delphi (Delphi 2007 for Win32) – першу систему програмування, яка дозволяє розробляти додатки, сумісні з Windows Vista. У вересні 2007 року компанія Code Gear анонсувала вихід об'єднаної системи програмування Code Gear RAD Studio, яка дозволяє створювати Windows- та Internet-додатки для платформ Win32, Microsoft.NET на мовах Delphi та C++.

Останньою (***дванадцятою***) версією Delphi є Delphi for Win32 2009, прес-реліз на яку був опублікований 25 серпня 2008 року компанією Embarcadero, новим власником Code Gear. З багатьох нововведень, привнесених у мову цією версією, можна відзначити повну підтримку Юнікоду за умовчанням в усіх частинах мови (VCL і RTL), введення узагальнених типів (**`generics`**), а також анонімних методів.

Запитання для контролю і самоконтролю

1. З яких основних компонентів складається комп'ютер?
2. Назвіть дані для програми, що виконує множення двох чисел.
3. Що є даними в програмі, яка визначає оцінки всіх студентів (учнів) групи (класу) за результатами контрольної роботи з декількох завдань?
4. У чому відмінність програм, написаних машинною мовою, та програм, що написані мовою високого рівня?
5. Для чого призначений компілятор?
6. Що таке вихідна програма?
7. Що таке об'єктна програма?
8. У чому полягає призначення компонувальника?
9. Що таке операційна система?
10. Яке призначення операційної системи?

11. Назвіть операційну систему вашого комп'ютера.
12. Що таке редагування зв'язків?
13. Які кроки в наведеній нижче послідовності дій неприпустимі в алгоритмі? З якої причини?
    Засипте в міксер 2 чайні ложки цукру.
    Додайте в міксер 1 яйце.
    Додайте в міксер 1 чашку молока.
    Якщо ви не за кермом, додайте 30 мл рому.
    Додайте екстракт ванілі за смаком.
    Збийте до одержання однорідної маси.
    Вилийте суміш у красивий фужер.
    Посипте мускатним горіхом.
14. З чого необхідно починати створення програми?
15. На які фази можна розбити процес створення програми?
16. Перелічите та охарактеризуйте три основні види помилок програмування.
17. Які види помилок виявляє компілятор?
18. До якого виду належить помилка, що полягає у пропущенні в програмі кінцевої закриваючої круглої дужки?
19. Чи потрібно реагувати на попереджувальні повідомлення компілятора або їх можна ігнорувати? Поясніть відповідь.
20. Припустімо, існує програма, призначена для обчислення банківських відсотків, і в ній допущена помилка у розмірі нараховуваних відсотків. Якого виду ця помилка?
21. Дайте характеристику помилок часу виконання.
22. Що таке тестування програми?
23. Чий обчислювальний пристрій визнається першим з тих, які вплинули на подальший розвиток обчислювальної техніки?
24. У чому основна заслуга Чарльза Беббіджа та Ади Лавлейс?
25. Хто є основоположником алгебри логіки?
26. Назвіть розроблювачів перших релейних машин.
27. Яку концепцію заклав у створення обчислювальних машин Джон Атанасов?
28. У чому полягає основний принцип архітектури фон Неймана?
29. Назвіть перші електронні обчислювальні машини.
30. Яка алгоритмічна мова визнається першою мовою програмування високого рівня.
31. Коли і хто розробив мову Pascal?
32. У чому особливості процедурного, структурного та об'єктно-орієнтованого програмування?
33. Дайте коротку характеристику різних версій Delphi.

2. ПОЧАТКИ ПРОГРАМУВАННЯ У СЕРЕДОВИЩІ DELPHI

2.1. Найпростіший приклад розв'язання задачі

Надалі ми будемо орієнтуватися на Delphi 7.

Розгляд матеріалу почнемо з найпростішого прикладу, який ілюструє думку що якщо відомий шлях розв'язання задачі, то написання програми – не така вже й складна справа (приклад 2.1).

```
//Приклад 2.1
//Дане ціле число. Скільки цифр входить у його десятковий
//запис?
```

Послідовність дій, призначена для розв'язання цієї задачі, може виглядати, наприклад, так:

1. Отримати ціле число `N`.
2. Покласти значення деякого лічильника `Count`, призначеного для підрахунку кількості цифр, рівним `0` (поки число `N` ще не опрацьовувалося).
3. Повторювати наступні дії п. 3.1 і 3.2, поки `N` не виявиться рівним `0`:
   3.1. Збільшити `Count` на 1.
   3.2. Відітнути від числа `N` останню цифру (розділити `N` на `10`, взявши як результат цілу частину від ділення).
4. Вивести значення `Count`.

Відкидаючи введення числа `N` і виведення числа `Count`, а також дії, пов'язані з задаванням характеристик змінних, які використовуються в програмі, для реалізації цієї послідовності дій може бути застосовано такий фрагмент програми мовою Delphi:

```
Count := 0;                                    //Покласти Count=0
repeat                                            //Повторювати
  Inc(Count);                                //Збільшення Count на 1
```

```
  N := N div 10;          //N – ціла частина від ділення N на 10
until N = 0;                //поки не виконається рівність N = 0
```

Як це видно із запропонованого тексту, якщо послідовність дій з перетворення даних у шуканий результат (інакше – ***алгоритм***) визначена цілком конкретно, то написання програми близьке до формального перекладу цієї послідовності дій на мову програмування.

У програмі повинні бути задані властивості використовуваних змінних `N` і `Count`, для чого її можна доповнити, наприклад, такими операторами:

```
var
  N: Integer:
  Count: Byte;
```

Ці два оператори вказують, що змінна `N` може набувати тільки цілих значень (можливо, від'ємних), а змінна `Count` – цілих невід'ємних значень.

Крім того, необхідно виконати оформлення програми за правилами мови Delphi і доповнити її засобами введення та виведення даних. Варто також урахувати те, що при використанні бібліотеки візуальних компонентів (Visual Component Library, VCL) створення готової програми здійснюється за принципами, які значно відрізняються від тих принципів, які властиві методиці програмування попередником Delphi – мовою Object Pascal.

2.2. Два етапи розробки програми

Розробка складних програмних продуктів, оформлених відповідно до сучасних вимог, практично неможлива без використання інструментальних засобів швидкої розробки програм (Rapid Application Development, RAD), першим з яких був Visual Basic, створений корпорацією Microsoft. Одним з таких засобів є система програмування Delphi, створена корпорацією Borland і призначена для розробки програм з використанням середовища візуального проектування.

Як і у будь-якому іншому середовищі візуального програмування, у середовищі Delphi процес розробки програми складається з двох етапів, перший з яких орієнтований на роботу з візуальними конструкторами, а другий – на роботу з програмним кодом. Характерним при цьому є те, що в процесі виконання другого етапу (безпосереднього програмування) завжди можна повернутися до першого з метою корекції проекту та наступним продовженням написання програмного коду. Ця особливість забезпечує гнучкість процесу розробки програми.

Візуальним відображенням працюючої програми, що написана у Delphi, є форма, яку і розробляє програміст, починаючи проектування програми. Спеціальне вікно форми створюється автоматично, як тільки програміст у візуальному середовищі вибере команду Application (Додаток). При цьому середовище Delphi автоматично створює оформлений за певними правилами спеціальний файл опису форми, що має розширення .DFM. Крім того, створюється пов'язаний з формою та оформлений спеціальним чином файл (див. підрозд. 2.6), що називається *модулем форми*.

Перший етап проектування програми полягає в наповненні вікна форми елементами керування, які називаються *компонентами*. Такими елементами є різні кнопки, прапорці, списки, вікна редакторів тощо. Програміст оперує компонентами, маючи їхнє візуальне відображення. Можна сказати, що він просто бере їх з так званої палітри компонентів і розміщає на формі.

Якщо програміст поміщає на форму який-небудь компонент, то Delphi автоматично робить відповідні зміни у файлі форми, а також у модулі, вносячи в останній посилання на доданий компонент. При цьому навіть порожній, створеній автоматично формі відповідає програма, що може бути виконана після компіляції.

Компоненти, що входять у бібліотеку візуальних компонентів Delphi, фактично є програмними заготівками для майбутньої програми, оскільки вони містять у собі як програмний код, так і дані, необхідні для його функціонування. Тому і програма в цілому, і розміщені у вікні форми компоненти є працездатними навіть без зміни автоматично створеного програмного коду. Якщо автоматично встановлені налаштування яких-небудь із розміщених на формі компонентів не влаштовують програміста, він може внести зміни без звертання до програмного коду. Для цього в Delphi передбачено засоби, що дозволяють змінювати характеристики компонентів за допомогою мишки або клавіатури вже на етапі проектування.

На другому етапі розробки програми програміст повинен наповнити компоненти програмним кодом, призначеним для розв'язання поставленої задачі. При цьому, природно, йому багато в чому доводиться орієнтуватися на традиційну техніку програмування. У той же час вживання компонентів істотно змінює техніку програмування: сучасне програмування базується на об'єктно-орієнтованому підході з використанням таких понять, як класи, властивості, методи та події. Установлені при проектуванні форми початкові налаштування компонентів можуть змінюватися програмно в процесі виконання додатка. Більш того, у процесі виконання програми компоненти можуть програмно створюватися та знищуватися.

У процесі написання програмного коду програміст завжди може повернутися до етапу проектування, здійснюючи модифікацію форми в

плані не тільки її візуального відображення, але і наповненості компонентами.

Було б дивно, якби в системі програмування Delphi не було передбачено засоби налаштування програм, адже вони є навіть у її досить далекого предка – мови Turbo Pascal. Достатньо потужні засоби налаштування дозволяють виконувати програму за операторами, стежачи за ходом виконання по тексту і контролюючи при цьому поточні значення змінних. Можлива також установка точок припинення (переривання), при досягненні яких програма автоматично переривається свою роботу, переходячи у налаштовувальний режим.

2.3. Початкові відомості про середовище розроблювача Delphi 7

Інтегроване середовище розроблювача (ICP) Delphi візуально реалізуються декількома одночасно відкритими вікнами (рис. 2.1), які програміст може переміщувати по екрану, створюючи собі комфортні умови для роботи.

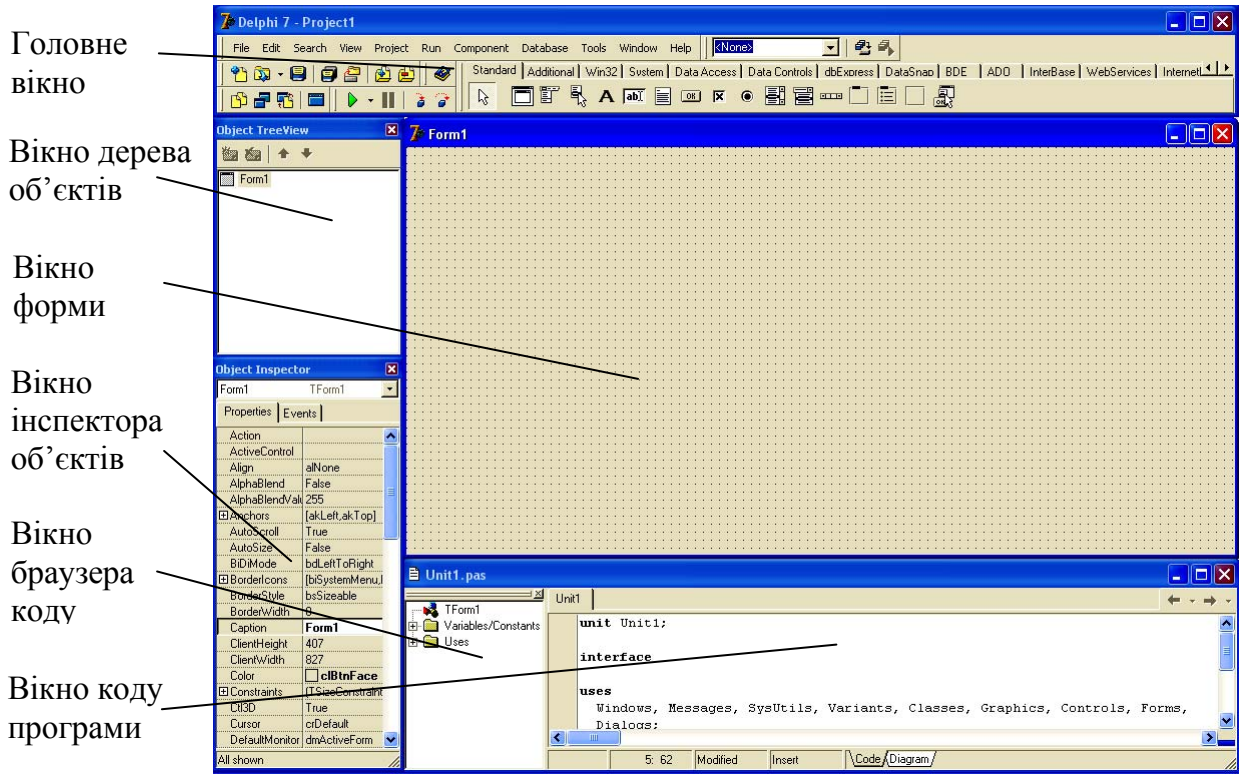


Рис. 2.1. Найважливіші вікна ICP Delphi

Вікна можна розміщати таким чином, щоб вони частково перекривали або повністю закривали одне одного зі збереженням можливості

доступу до них. При цьому кожне вікно призначене для вирішення цілком конкретного завдання. Кожне з вікон, крім головного, може бути закрите. Закриття ж головного вікна приводить до виходу з ICP Delphi.

2.3.1. Головне вікно

Це вікно містить у собі всі засоби з керування проектом (рис. 2.2): головне меню, набір інструментальних кнопок і палітру компонентів.

Усі елементи головного вікна згруповані на окремих панелях, які мають спеціальні вішки, призначені для переміщення цих панелей при їхньому захоплюванні мишкою і перетаскуванні при натиснутій її лівій кнопці. Такі ж вішки з'являються й у вікон при вбудовуванні їх усередину інших вікон (див., наприклад, вікно браузера коду на рис. 2.1). Будь-яку панель головного вікна (крім головного меню) можна прибрати з екрана.

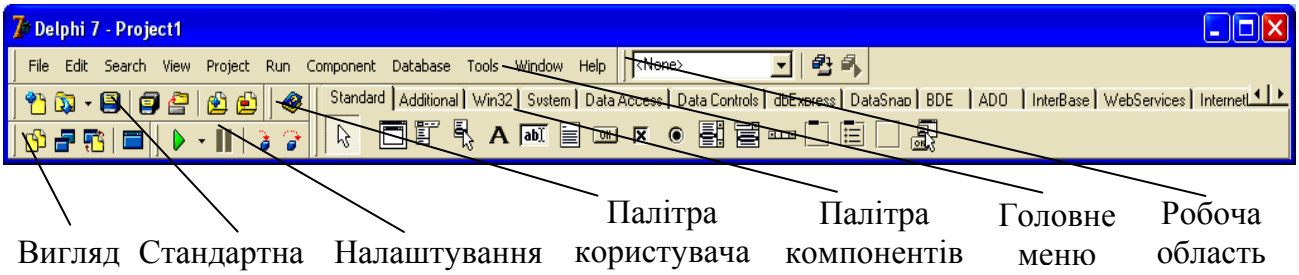


Рис. 2.2. Основні панелі головного вікна ICP Delphi

За допомогою головного меню здійснюється керування можливостями ICP.

Швидкий доступ до багатьох команд головного меню забезпечують так звані інструментальні кнопки, згруповані за функціональним призначенням на окремих панелях головного вікна. Панель Стандартна містить інструментальні кнопки, що забезпечують дублювання тих дій, які можна виконати, звернувшись до пунктів Файл і Проект головного меню. Панель Вигляд поєднує кнопки, що дублюють підпункти пункту View (Вигляд) головного меню, а панель Налаштування полегшує керування процесом налаштування та виконання програми, частково дублюючи підпункти пункту Run (Виконати) головного меню. Палітра користувача за умовчанням вміщує інструментальну кнопку, що забезпечує доступ до довідкової системи Delphi.

Особливе місце займає Палітра компонентів – багатосторінкова панель, на сторінках якої розташовуються інструментальні кнопки, що забезпечують додавання компонентів до активної форми при проектуванні інтерфейсу програми.

2.3.2. Вікно форми

Вікно форми (конструктор форми) являє собою проект вікна створюваної програми. Спочатку воно містить кнопки виклику системного меню, розгортання, згортання і закриття вікна, рядок заголовка і габаритну рамку (ці елементи є стандартними інтерфейсними елементами Windows), а також порожню робочу область, звичайно заповнену крапками координатної сітки, що служить для полегшення позиціювання розташовуваних на формі компонентів і відображається тільки на етапі конструювання програми.

За допомогою команди Tools ► Environment Options (Сервіс ► Опції Середовища) можна викликати вікно налаштувань середовища і, знявши прапорець Display Grid (Показувати) на вкладці (сторінці) Designer (Дизайнер), прибрати координатну сітку з форми.

Будучи візуальним засобом розміщення компонентів на формі, конструктор форми дозволяє безпосередньо за допомогою миші розміщати компоненти на формі з використанням палітри компонентів головного вікна, Інспектора Об'єктів (Object Inspector) або Дерева Об'єктів (Object Tree). Причому, як про це вже говорилося раніше, необхідні зміни у вікні форми можуть бути внесені в будь-який момент під час проектування, наприклад, за допомогою миші можна змінювати положення і розміри компонентів.

2.3.3. Вікно дерева об’єктів

У вікні Дерева Об’єктів (Object TreeView) наочно (у вигляді дерева) відображаються зв’язки між окремими візуальними та невізуальними компонентами, розміщеними на активній формі або в активному модулі даних, а саме, приналежність одних компонентів іншим. Окремі компоненти відображаються в Дереві Об’єктів за допомогою піктограм (маленьких рисунків), поруч із якими містяться імена відповідних компонентів. Клацання (клік) лівою кнопкою мишки на кожній з таких піктограм приводить до активізації відповідного компонента у вікні форми і відображення його властивостей у вікні Інспектора Об’єктів (Object Inspector). При подвійному кліку на піктограмі компонента спрацьовує так званий механізм Code Insight, що вставляє у вікно коду заготівку для опрацьовувача події OnClick (За кліком) або опрацьовувача іншої події (залежно від типу компонента).

Використовуючи Дерево Об’єктів, можна змінити власника компонента, для чого досить клацнути лівою кнопкою мишки над піктограмою і, не відпускаючи кнопку мишки, «перетягнути» її у вікні на піктограму нового власника. Натискання клавіші Delete приводить до видалення компонента, який виділений у Дерево Об’єктів.

2.3.4. Вікно інспектора об'єктів

Вікно Інспектора Об'єктів (Object Inspector) призначене для зміни параметрів (характеристик) розміщених на формі компонентів. За допомогою мишки у вікні форми можна коректувати тільки частину параметрів, які характеризують компоненти, що розміщені на формі. Користуючись же Інспектором Об'єктів, програміст може коректувати всі характеристики компонентів, які доступні на етапі проектування програми (наприклад, колір, шрифт і текст напису, колір компонента). Для здійснення цих змін в Інспекторі Об'єктів використовуються дві вкладки (сторінки) – Properties (Властивості) та Events (Події). За допомогою вкладки Properties (Властивості) програміст може змінювати властивості компонента, а за допомогою вкладки Events (Події) – призначати реакцію компонента на ту або іншу подію (реакцію на натискання клавіш, клік мишкою, зміну розміру вікна, появу компонента на екрані тощо).

Для зміни властивостей компонента необхідно перейти в Інспекторі Об'єктів на першу вкладку, що має вигляд таблиці, у першому стовпчику якої містяться імена властивостей. Значення властивості відображається поруч з її ім'ям у другому стовпчику.

Властивості компонентів можуть відображатися єдиним значенням (числом, рядком символів, True – Істина чи False – Неправда) або множиною значень. У першому випадку говорять про прості властивості, а в другому – про складні. Ліворуч від імені складної властивості відображається значок «+».

Для задавання значення простої властивості необхідно вибрати відповідний рядок і ввести потрібне значення в правому стовпчику. Якщо при виборі простої властивості в правому кінці рядка з'являється кнопка  , то це означає, що для даної властивості визначений список можливих значень, який розкривається при кліку над кнопкою  і служить для подальшого вибору потрібного значення. Для зміни значення складної властивості слід клацнути мишкою над значком «+» поруч з його ім'ям, у результаті чого відкривається список складових цієї властивості. Закриття цього списку здійснюється кліком мишкою над значком «–», у який перетворюється значок «+» при відкритті списку.

Наприкінці рядка для деяких із властивостей при їх активізації може з'явитися кнопка  . Клік над цією кнопкою приводить до появи на екрані діалогового вікна, що служить для установки значення властивості.

Вкладка Events (Події) також має вигляд таблиці з декількох рядків і двох стовпчиків. У першому стовпчику відображається ім'я події, а в другий – ім'я підпрограми для її опрацювання (ім'я опрацьовувача події).

Крім двох вкладок, у вікні Інспектора Об'єктів є список, який може розкриватися і містить імена всіх поміщених на форму компонентів із зазначенням їхніх класів. При кліку мишкою над ім'ям компонента в цьому списку відбувається переключення на відповідні таблиці в нижній частині Інспектора Об'єктів, а також активізація обраного компонента у вікні форми, що позначається виділенням компонента прямокутником.

Налаштування вікна Інспектора Об'єктів можна виконати за допомогою контекстного меню, яке відкривається при кліку правою кнопкою миші.

2.3.5. Вікно коду програми

У вікні коду програми відображається і редагується текст програми (найчастіше текст модуля), написаної мовою Delphi, базою якої є алгоритмічна мова Object Pascal. При розробці нового проекту це вікно містить створюваний автоматично мінімальний початковий текст модуля. Цей текст забезпечує нормальне функціонування порожньої форми. Далі програміст модифікує даний текст відповідно до розв'язуваної задачі, причому зміни виконуються як за допомогою засобів VCL, так і за допомогою «ручної» корекції. При цьому досить часто доводиться переходити з вікна коду у вікно форми і зворотно. Якщо потрібне вікно хоча б частково видно на екрані, перехід може бути здійснений кліком мишкою над видимою частиною вікна.

Якщо вікно коду закрите вікном форми (або навпаки), то для переходу необхідно клацнути над інструментальною кнопкою  , розташованою на панелі Вигляд ICP Delphi або натиснути клавішу <F12>.

2.3.6. Вікно браузера коду

Вікно Браузера Коду (Code Explorer) звичайно активізується разом з вікном коду і служить для полегшення пошуку потрібних елементів у великому тексті програми. Вікно браузера коду містить елементи, що використовуються в програмі.

При подвійному кліку мишею над елементом у вікні браузера коду текстовий курсор у вікні коду автоматично переміщається на опис цього елемента або на рядок, у якому він згадується вперше. За допомогою браузера коду можна також проводити додавання і перейменування елементів програми.

Горизонтальна вішка у верхній частині вікна браузера коду дозволяє перетаскувати його за допомогою мишки, розташовуючи в будь-якому зручному для програміста місці екрана.

Закрити вікно браузера коду можна, клікнувши мишкою над кнопкою ⊠ в його правому верхньому куті. Для відкриття раніше закритого вікна браузера коду треба або клацнути правою кнопкою мишки над вікном коду програми і вибрати опцію View Explorer (Показати Огляд), або скористатися меню, виконавши команду View ► Explorer (Показати ► Огляд Коду).

2.4. Початкові відомості про середовище розроблювача CodeGear Delphi 2009 for Win32

Робота в CodeGear Delphi 2009 for Win32 може виконуватися в середовищі операційних систем Microsoft Windows 2000, Microsoft Windows XP Professional, Microsoft Windows Server 2003, Microsoft Windows Vista. При цьому середовище Delphi 2009 for Win32 може працювати і як окремий інструмент розробки додатків, і як інструмент CodeGear RAD Studio. Розглянемо основні елементи цього середовища (середовище Embarcadero® Delphi® 2010 у тих елементах, що розглядаються нижче, практично не відрізняється від середовища CodeGear Delphi 2009).

2.4.1. Сторінка вітання і початок роботи

Інтегроване середовище розроблювача (ICP) Delphi 2009 for Win32 візуально реалізується декількома одночасно відкритими вікнами, розміщеними у вікні Delphi. При завантаженні Delphi 2009 for Win32 у центральній частині вікна Delphi міститься вікно зі сторінкою вітання (Welcome Page) CodeGear RAD Studio. При подальшій роботі це вікно залишається доступним (якщо його не закрити спеціально) із забезпеченням доступу до нього кліком мишкою над його закладкою. У вигляді окремих сторінок у тому ж самому вікні, у якому розташовується сторінка вітання, містяться вікно дизайнера форми та сторінки з кодами модулів. Праворуч угорі розташовуються дві кнопки, перша з яких (⊡) служить для виклику випадного меню, яке забезпечує перехід до тієї або іншої сторінки (перехід можна виконати кліком мишки над відповідною закладкою), а друга (⊠) – для закриття активної сторінки.

Зокрема, за допомогою сторінки вітання забезпечується швидкий доступ до недавніх проектів, створення нового і відкриття існуючого проекту. Якщо сторінку вітання закрили, то повторно її відкривають командою View ► Welcome Page.

У подальшому вікно Delphi трохи змінюється, у цілому зберігаючи свою структуру. На початку роботи над створюваним у Delphi 2009 for Win32 новим проектом вікно Delphi здобуває вигляд, наведений на рис. 2.3.

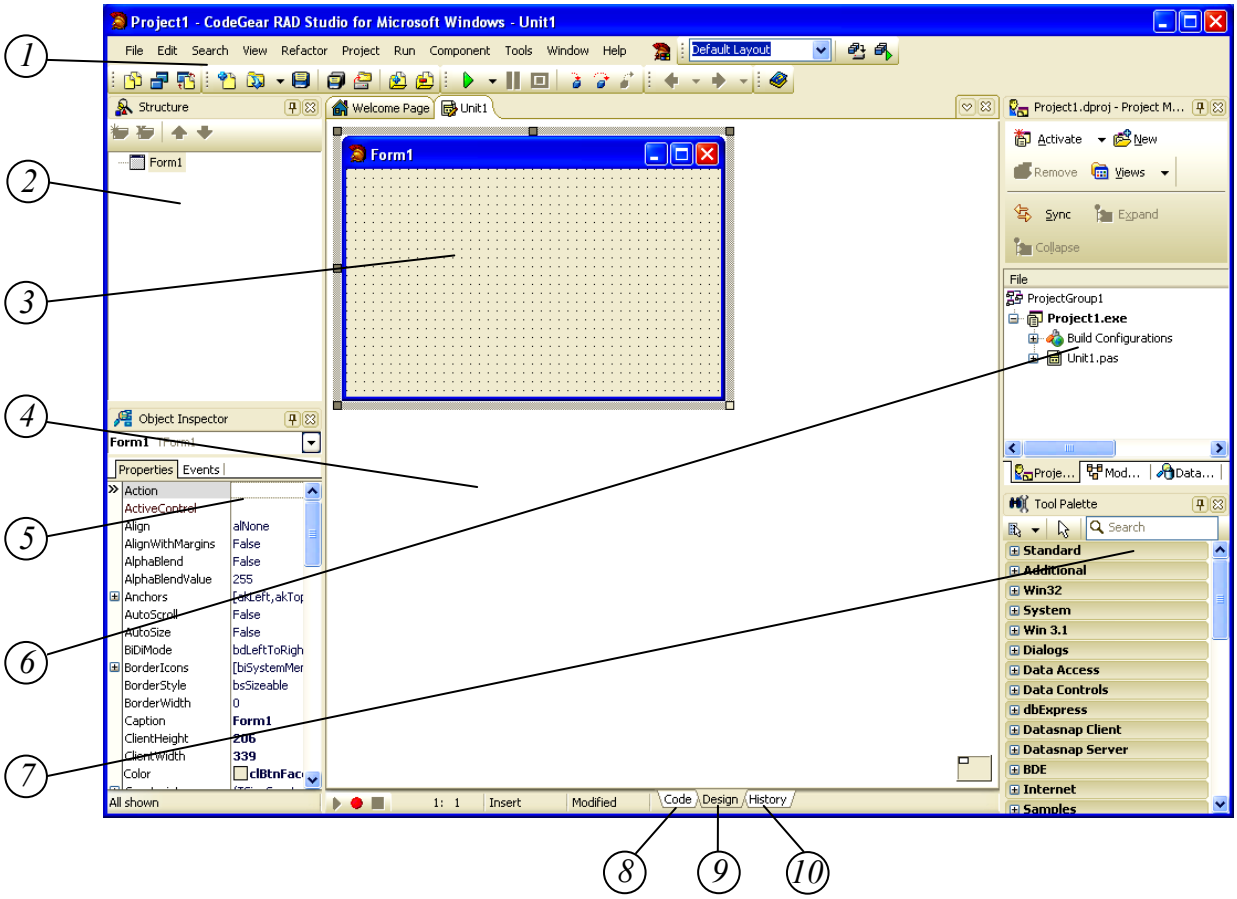


Рис. 2.3. Вікна ICP Delphi 2009 for Win32 на початку роботи над новим проектом: *1* – головне вікно; *2* – вікно структури; *3* – вікно форми; *4* – вікно дизайнера форми; *5* – вікно інспектора об’єктів; *6* – вікно менеджера проектів; 7 – вікно палітри інструменту; *8* – закладка вікна коду; *9* – закладка вікна дизайнера форми; *10* – закладка вікна менеджера історії

У верхній частині вікна Delphi розміщене головне вікно, а в його центральній частині розташовується вікно дизайнера (конструктора) форми. Ліворуч від вікна дизайнера форми – вікно інспектора об’єктів (унизу) і вікно структури (угорі). У правій же частині вікна Delphi також розташовується два вікна – вікно менеджера проектів (угорі) і вікно палітри інструменту (унизу).

2.4.2. Головне вікно

Головне вікно є аналогічним відповідному вікну ICP Delphi 7. При цьому з нього виключена палітра компонентів, що дозволило розмістити набір інструментальних кнопок не у два ряди, а в один. Крім того, додано кілька інструментальних кнопок (наприклад, кнопка – Program Reset).

Головне меню поповнилося опцією Refactor, що реалізує техніку рефакторингу, використовувану для зміни існуючого коду без зміни його поводження. Ця техніка дозволяє розроблювачам спрощувати свою роботу з поліпшення зручності в прочитанні коду. У Delphi 2009 for Win32 забезпечуються такі дії рефакторингу:

- перейменування ідентифікаторів, методів, типів та інших об'єктів;
- оголошення змінних і полів;
- пошук посилань;
- пошук модуля;
- зміну параметрів;
- скасування перейменування.

Забезпечується також ряд інших дій.

2.4.3. Вікно дизайнера форми

На початку роботи над новим проектом вікно дизайнера форми у вигляді окремої сторінки із закладкою Design розташовується над сторінкою вітання CodeGear RAD Studio, за умовчанням будучи відкритим. У правому нижньому куті сторінки дизайнера форми виводиться невеликий затінений прямокутник, який умовно відображає екран при роботі додатка. На зображенні екрана відображається місце розташування і розміри форми відразу ж після запуску додатка. При зміні розміру або місця розташування форми на етапі її конструювання автоматично змінюються розміри і місце розташування її зображення на зображенні екрана. Захопивши мишкою зображення форми на зображенні екрана, можна виконати перетаскування її з автоматичною зміною координат. Клік мишкою над зображенням екрана забезпечує збільшення цього зображення відносно початкових розмірів, причому в цьому випадку можливе захоплення мишкою лівого верхнього кута зображення з метою пропорційного розтягування або зменшення останнього. За наявності у проекті декількох форм дизайнер форми містить кілька сторінок, закладки яких розміщуються у верхній його частині. Кожна з форм у цьому випадку відображується на зображенні екрана, причому зображення форми, з якою здійснює роботу дизайнер форми, залите білим кольором, а зображення інших форм затінені (див. рис. 2.4 для проекту з двома формами).

У лівому верхньому куті вікна дизайнера форми на етапі проектування розташовується вікно форми. Розміри форми можуть змінюватися за допомогою миші, для чого слід захопити праву або нижню її межі або правий нижній кут. З метою полегшення вирівнювання компонентів дизайнер VCL показує напрямні лінії, які висвітлюються при перетаскуванні

компонентів за допомогою миші. У цілому ж робота з дизайнером при проектуванні VCL-форми нічим особливим не відрізняється від роботи в середовищі Delphi 7.

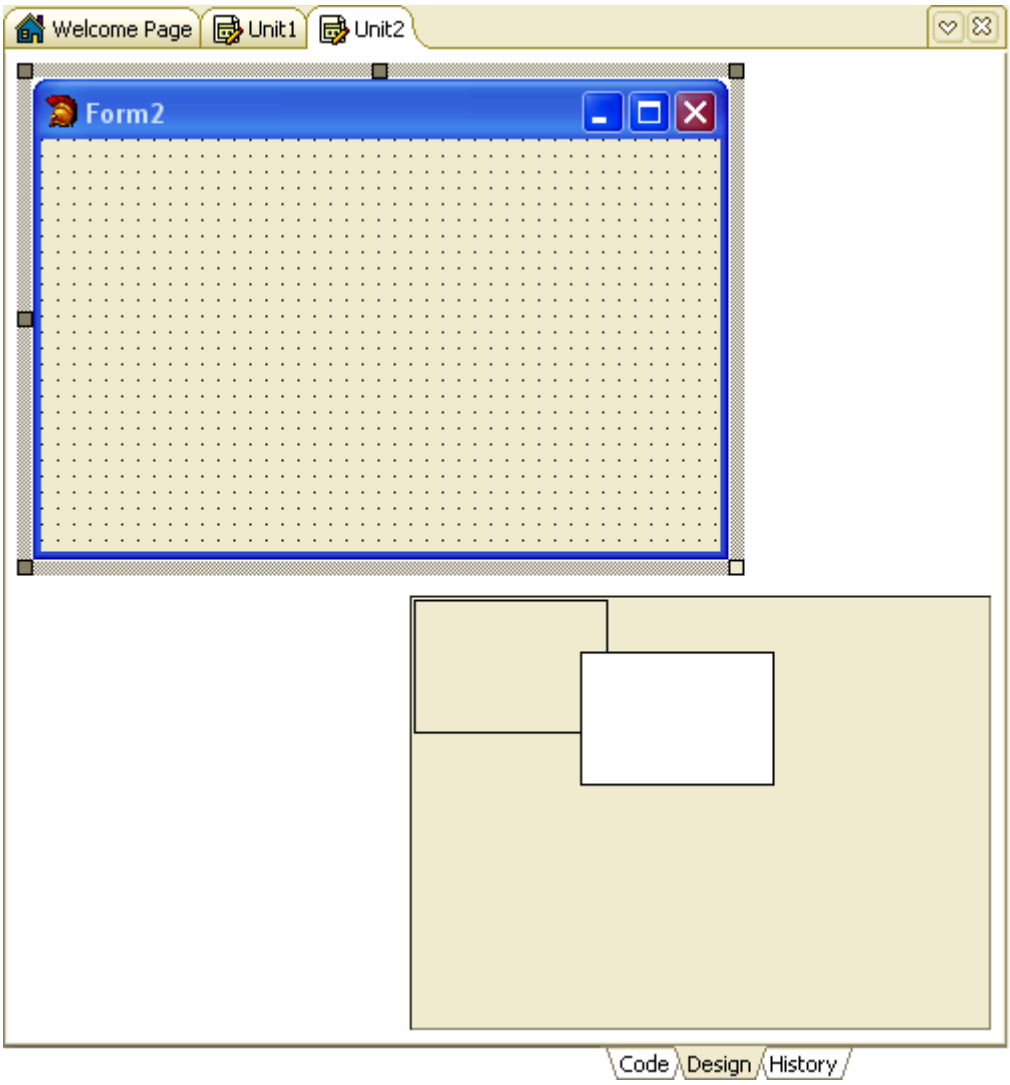


Рис. 2.4. Зображення екрана в дизайнері форми

З формою, крім дизайнера форми, пов'язані ще два вікна, розташовані на окремих сторінках разом зі сторінкою дизайнера, а саме: вікно коду (закладка з ім'ям Code) і вікно менеджера історії (закладка з ім'ям History).

2.4.4. Вікно коду

Доступ до вікна коду (рис. 2.5), як і в Delphi 7, здійснюється або натисканням клавіші <F12>, або кліком над інструментальною кнопкою  , або виконанням команди View ► Toggle Form/Unit. Зворотний перехід у вікно дизайнера форми здійснюється аналогічно. Можна також виконувати кліки мишею над закладками Code і Design.

Редактор коду в ICP Delphi 2009 for Win32 забезпечує нумерацію рядків. За умовчанням пронумеровані всі рядки з номерами, кратними десяти, а також поточний рядок (див. рис. 2.5).

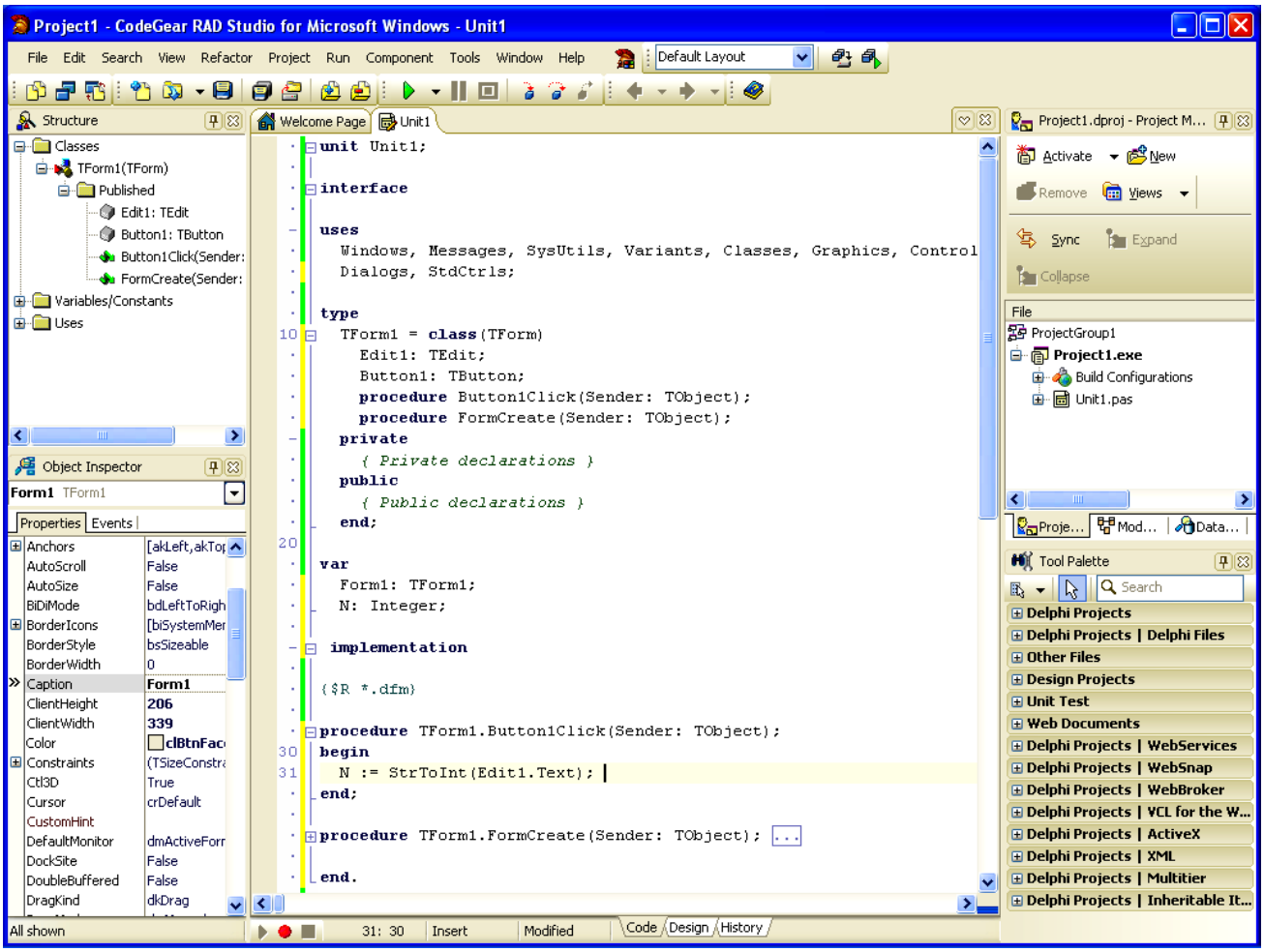


Рис. 2.5. Вигляд ICP Delphi 2009 for Win32 з відкритим вікном коду

Інтелектуальні можливості редактора коду забезпечують полегшення роботи під час набирання програмного коду. Це забезпечується за допомогою автоматичного формування програмних заготовок для деяких стандартних конструкцій:

- якщо виконується набирання заголовка будь-якого циклу (**for**, **while**, **do**), введення пробілу після першого його слова приводить до автоматичного формування заготовки для продовження циклу (наприклад для циклу **for** така заготовка має вигляд **for** I := 0 **to** List.Count - 1 **do**);
- введення пробілу після слів **if** та **case** приводить до автоматичного формування продовження операторів **if** та **case**;
- якщо після службового слова **begin** натискається клавіша <Enter>, то автоматично після порожнього рядка формується рядок з

закриваючою операторною дужкою **end** і символом «крапка з комою».

Окремі секції коду (модуль, його інтерфейсна секція та секція реалізації, описи класів, процедур, функцій тощо) можуть бути «згорнуті», забезпечуючи приховання згорнутого коду. Для згортання секції слід клацнути мишкою над квадратиком із символом «мінус» поруч із заголовком секції. У цьому випадку секція згортається до свого заголовка, ліворуч від якого у квадратику міститься символ «плюс», а праворуч – прямокутник із трьома крапками. Для «розгортання» згорнутої секції достатньо клацнути мишкою або над квадратиком із символом «плюс», або над прямокутником із трьома крапками. Наприклад, на рис. 2.5 згорнутою є секція опису процедури `TForm1.FormCreate`, а інші секції коду модуля `Unit3` розгорнуті (наприклад, секція **unit** охоплює весь код модуля).

2.4.5. Менеджер історії

Менеджер історії (History Manager) розташовується на одній із вкладок вікна дизайнера форми аналогічно вікну коду. Доступ до нього здійснюється кліком мишкою над закладкою History у нижній частині вікна дизайнера форми. Менеджер історії дозволяє побачити і порівняти різні версії файлу, включаючи резервні копії. Використовуючи менеджер історії, можна, зокрема, повернутися до необхідної версії файлу.

2.4.6. Вікно інспектора об'єктів і вікно структури

Вікно інспектора об'єктів (Object Inspector) практично збігається з аналогічним вікном ICP Delphi 7, описаним у п. 2.3.4, у зв'язку із чим немає потреби в окремому його розгляді. Повторне відкриття раніше закритого вікна інспектора об'єктів виконується командою View ▶ Object Inspector (клавіша F11).

Вміст вікна структури (Structure) залежить від того, що завантажено в центральне вікно – дизайнер форми або вікно коду. Якщо в центральному вікні відкритий дизайнер форми, то у вікні структури міститься дерево об'єктів, яке, в основному збігається з деревом об'єктів (Object TreeView) в ICP Delphi 7. Зокрема, подвійний клік мишкою над зображенням компонента в дереві

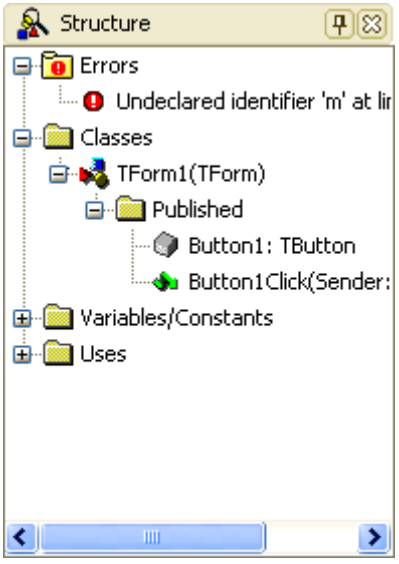


Рис. 2.6. Браузер коду з помилкою в описі

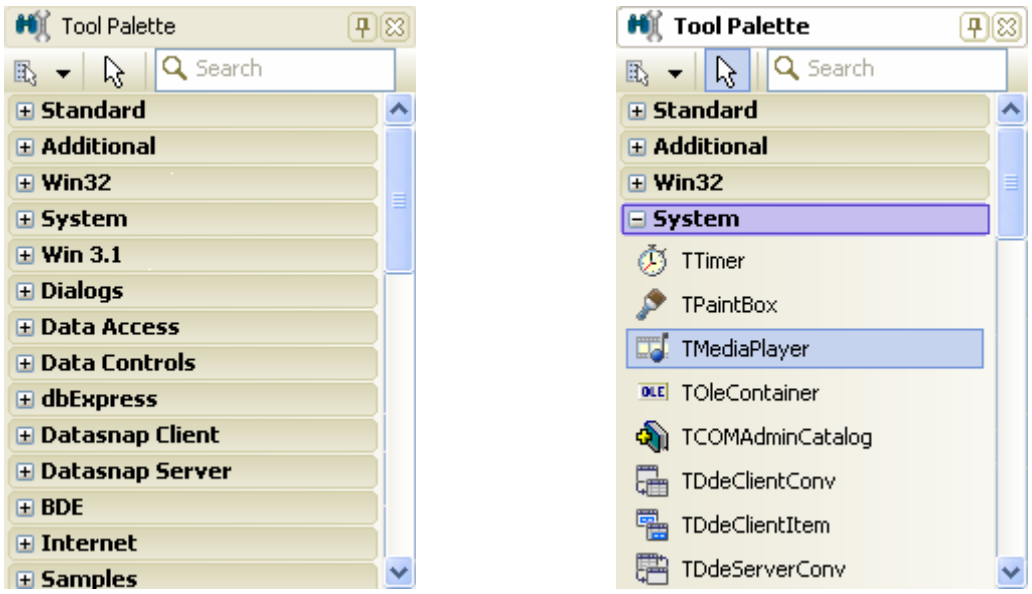
об'єктів приводить до відкриття вікна коду і створення відповідного оброблювача події із розміщенням курсору усередині цього оброблювача.

При відкритому вікні коду у вікно структури завантажується браузер коду. На відміну від браузера коду Delphi 7, у цьому вікні додатково відображаються помилки в описі різних об'єктів, які використовуються в модулі (див., наприклад, рис. 2.6). Кліком мишкою над гілкою з описом помилки можна забезпечити перехід до відповідної позиції у вікні коду.

Повторне відкриття раніше закритого вікна структури виконується командою View ► Structure (Shift+Alt+F1).

2.4.7. Вікно палітри інструменту

Якщо в ICP Delphi 7 палітра компонентів розташовується в головному вікні як окрема панель (правда, її можна «перетягнути» мишею в інше місце, створивши окреме вікно), то в ICP Delphi 2009 for Win32 вона розміщена в спеціальному вікні, що називається палітрою інструменту (Tool Palette) і розташоване праворуч від вікна дизайнера форми. Окремі вкладки палітри інструменту розміщені у вікні у вигляді списку вкладок, кожна з яких розкривається/закривається кліком мишкою над значком ⊞ або ⊟ ліворуч від назви вкладки (рис 2.7, *а*, *б*).



а *б*

Рис. 2.7. Вікно палітри інструменту при відкритому вікні форми:
а – з закритими вкладками; *б* – з відкритою вкладкою System

Компоненти при розкритих вкладках відображаються піктограмами, поруч із кожною з яких зазначається клас відповідного компонента (рис.

2.7, *б*). Піктограми більшості компонентів мають трохи інший вигляд порівняно з піктограмами для ICP Delphi 7. Робота з палітрою інструменту в ICP Delphi 2009 for Win32 аналогічна роботі в ICP Delphi 7. При цьому їх можна розміщати на формі не тільки за кліком мишкою, але й перетаскувати з палітри інструменту при натиснутій лівій клавіші миші.

Істотною особливістю палітри інструменту ICP Delphi 2009 є те, що вона доступна для пошуку з використанням фільтра: можна просто набрати ім'я компонента або частину імені, у результаті чого відкривається список компонентів, імена яких містять як свою частину набраний текст.

2.4.8. Вікно менеджера проектів

Вікно менеджера проектів (Project Manger) розташоване над вікном палітри інструменту і має вигляд, наведений на рис. 2.8. Воно служить для відображення структури проекту, над яким веде роботу розроблювач додатка. За допомогою менеджера проектів можна додавати до проекту файли (модулі), видаляти їх і перейменовувати. Можна також комбінувати проекти, формуючи зв'язану групу проектів.

Власне менеджер проектів завантажується на першу сторінку вікна. На двох інших сторінках розташовані вікно перегляду організації (Model View) і вікно оглядача даних (Data Explorer). Перше з них служить для перегляду логічної структури й організації проекту, а друге – для зміни, видалення, перейменування існуючих і додавання нових зв'язків, а також перегляду бази даних.

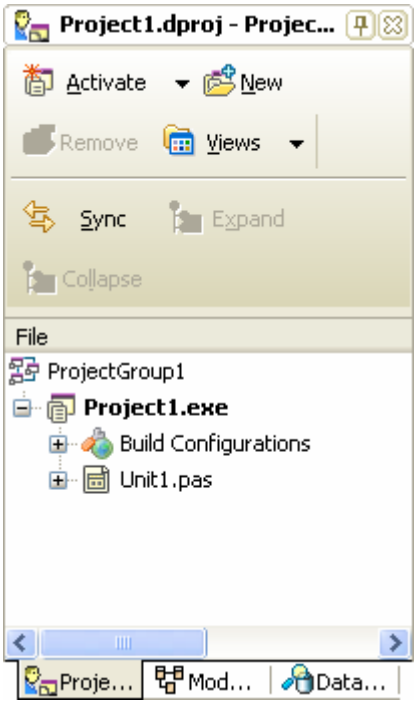


Рис. 2.8. Вікно менеджера проектів

У верхній частині вікна менеджера проектів розміщене меню, за допомогою якого можливі корекція проекту та перегляд його структури.

Повторне відкриття раніше закритого вікна менеджера проектів виконується командою View ▶ Project Manger (Ctrl+Alt+F11). Це ж можна забезпечити виконанням однієї з команд View ▶ Model View і View ▶ Data Explorer, які призначені для переходу в менеджері проектів на відповідні вкладки (сторінки).

Відзначимо, що ICP більш ранньої версії Delphi, а саме Delphi 2007 for Win32, практично не відрізняється від ICP Delphi 2009 for Win32, за ви-

нятком хіба що того, що в ньому присутній ще один пункт головного меню (опція ECO Utils), а також відсутнє меню у вікні менеджера проектів.

2.5. Проект

Мінімальний код, що може нормально функціонувати під керуванням операційної системи, створюється автоматично відразу ж після вибору команди File ► New ► Application (Файл ► Створити ► Додаток).

Якщо говорити про текст програми, написаної у Delphi, то як такий можна розглядати так званий файл проекту, що має розширення .DPR (у середовищах Delphi 2010, Delphi 2009 та Delphi 2007 використовується розширення .DPROJ). Саме цей файл обробляється компілятором, будучи головною програмною одиницею, до якої підключаються одна або кілька інших програмних одиниць, що називаються модулями та зберігаються у файлах з розширенням .PAS.

Програма, написана мовою Delphi (проект), містить три основні частини:

- заголовок програми;
- розділ описів;
- тіло програми.

Заголовок програми починається зі слова **`program`**, після якого зазначається ім'я програми та ставиться крапка з комою.

За заголовком програми розташовується ***розділ описів***, у якому оголошуються так звані ідентифікатори, які позначають різні елементи програми (модулі, константи, типи, змінні, процедури, функції).

Власне ***тіло програми*** починається зі слова **`begin`** і завершується словом **`end`** з крапкою, яке інакше називається ***термінатором***. Тіло програми складається з декількох операторів мови Delphi. У кожному операторі реалізується певна дія, наприклад, змінювання значення змінної, аналіз результату обчислення, звертання до підпрограми тощо.

Раніше згадувався термін «ім'я». Що ж цей термін означає в Delphi?

Ім'я (змінної, програми або якогось програмного об'єкта) – це послідовність допустимих у Delphi символів, яка задовольняє такі правила:

- ім'я може складатися тільки з латинських літер, цифр і символів підкреслення (при цьому великі та малі літери не розрізняються);
- ім'я повинне починатися тільки з латинської літери або символу підкреслення (_).

Стандартний текст проекту для функціонування порожньої форми у разі проектування додатку у середовищі програмування Delphi 7 має такий вигляд:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Зазвичай (якщо не проводилося переналаштування) у вікні коду автоматично виділяються жирним шрифтом так звані зарезервовані слова (див. підрозд. 3.1) і курсивом – коментарі (див. підрозд. 3.9).

Розглянемо текст проекту[1].

У першому рядку розташований заголовок програми з автоматично наданим ім'ям `Project1`. Це ім'я може бути змінене програмістом відповідно до змісту задачі.

Зарезервоване слово **`uses`** (використовується) служить для відкриття розділу опису модулів. У цьому розділі підключаються стандартний модуль `Forms` і створений модуль форми `Unit1`, невідомий Delphi, і тому зазначене ім'я файлу, який містить текст цього модуля (`Unit1.pas`). Крім того, у вигляді коментарю Delphi повідомляє ім'я описаної в модулі форми (стандартне написання коментарів – курсивне).

Для забезпечення нормального функціонування програми до неї повинен бути підключений файл ресурсів Windows. Це виконується на етапі компонування програми, і вказівкою на необхідність підключення файлу ресурсів буде рядок `{$R *.res}`, що є однією з так званих директив компілятору. Файл ресурсів створюється автоматично для кожної

---

[1] Якщо додаток розробляється у середовищі Delphi 2010, Delphi 2009 або Delphi 2007, то основна частина коду проекту має такий вигляд:

```
begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

програми при її компіляції, дістаючи ім'я, що збігається з ім'ям проекту, і розширення .RES.

При виконанні програми завжди автоматично створюється спеціальний об'єкт, який має ім'я `Application` і є об'єктом-програмою. Автоматично створюваний об'єкт `Application` акумулює в собі дані та методи їх обробки, необхідні для нормального функціонування Windows-програми.

Взагалі кажучи, ***об'єктом*** називається оформлений за певними правилами фрагмент програми, який об'єднує дані (вони називаються ***полями*** об'єкта) і підпрограми (***методи***) для їх обробки. У програмах об'єкти розглядаються як єдине ціле і будуються таким чином, щоб повністю забезпечувати розв'язання конкретної задачі. Будучи самодостатніми програмними одиницями, один раз розроблені об'єкти можуть використовуватися при розв'язанні багатьох задач. При цьому процес написання програми багато в чому зводиться до комбінування об'єктів, як тих, що розроблені іншими програмістами та входять до різних бібліотек, так і що створених автором даної програми. Саме на такому підході базується сучасна технологія програмування, яка називається об'єктно-орієнтованим програмуванням.

Стандартне тіло програми, написаної мовою Delphi, містить три оператори, що реалізують звертання до трьох методів об'єкта `Application`. Перший з цих методів

```
Application.Initialize;
```

здійснює виклик спеціальної підпрограми, ім'я якої записане в системній змінній `InitProc`. Ця підпрограма в явному вигляді не виконує ніяких дій.

Оператор

```
Application.CreateForm(TForm1, Form1)
```

служить для створення та показу на екрані вікна головної форми. Закриття цього вікна припиняє виконання програми.

Метод `Application.Run` забезпечує одержання та опрацювання повідомлень, які в ході виконання програми надходять від операційної системи, будучи сигналами про дії користувача або про роботу різного роду пристроїв.

Файл проекту має розширення .DPR і висвітлюється у вікні коду або на запит програміста, або з появою деяких помилок у ході виконання програми. Звичайно у вікні коду він не відображається і змінюється програмістом досить рідко.

2.6. Модуль форми

Модулі – це програмні одиниці, що служать для розміщення окремих частин програм. Будь-який модуль (у тому числі і модуль форми) має таку структуру:

- заголовок, який відкривається зарезервованим словом **unit**;
- секція інтерфейсних оголошень, що відкривається словом **interface**;
- секція реалізацій, що відкривається словом **implementation**;
- термінатор **end** з крапкою.

Після секції реалізації (перед термінатором) **можуть** бути розміщені ініціююча та (або) завершальна секції, що починаються відповідно зі службових слів **initialization** та **finalization**.

Заголовок модуля, як і заголовок проекту, завершується крапкою з комою.

Мінімальний синтаксично правильний варіант модуля має такий вигляд:

```
unit Unit1;

interface

// Секція інтерфейсних оголошень

implementation

// Секція реалізацій

end.
```

У секції інтерфейсних оголошень вміщується опис програмних елементів (констант, типів, процедур і т. д.), які будуть відомі («видні») іншим програмним модулям, що містять посилання на даний модуль. У секції ж реалізацій розкривається механізм роботи цих елементів.

Розділення модуля на дві секції забезпечує зручний механізм доступу до його ресурсів. Щоб отримати доступ до ресурсів модуля достатньо знати тільки особливості його інтерфейсної частини, що містить оголошення елементів. Деталі ж їхньої реалізації (йдеться про процедури і функції) залишаються прихованими.

Модулем форми є модуль, до якого входять елементи програмного коду, які забезпечують створення та функціонування форми та тих компонентів, що розміщуються на ній.

На вкладці Unit вікна коду у випадку стандартної форми міститься код, сформований Delphi, що в обов’язковому порядку повинен бути змі-

нений програмістом (нагадаємо, що код проекту програміст практично ніколи не змінює):

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.
```

В інтерфейсній секції наведеного програмного коду підключені стандартні модулі, а також описані один тип (клас `TForm1`) та один об'єкт (змінна `Form1`).

Об'єкти, що використовуються в програмі, створюються за деякими зразками, які називаються типами. Перед оголошенням типу вказується службове слово **`type`** (тип), що повідомляє компілятор про початок розділу опису типів. Одним з різновидів типів є клас (у цьому випадку він має ім'я `TForm1`).

Все потрібне для створення та функціонування порожнього вікна програми реалізовано в стандартному класі `TForm`. Будь-який клас може бути удосконалений (розвинений). Однак для забезпечення цього програмісти звичайно не змінюють вже існуючий клас, а породжують від нього новий клас, додаючи йому додаткову функціональність і зберігаючи всі можливості батьківського класу.

У наведеному тексті відбито, що клас `TForm1` *породжений* від стандартного класу `TForm`: про це свідчить рядок

```
TForm1 = class(TForm)
```

У цьому випадку, оскільки поки що йдеться про порожню форму, додаткові можливості у класу `TForm1` відсутні.

Важливими елементами програми є ***змінні***. Для того щоб їх можна було використовувати, вони попередньо повинні бути описані. Оскільки об'єкт `Form1` є змінною, він описаний в інтерфейсній частині модуля і буде відомим у проекті, у якому здійснене підключення даного модуля. Перед оголошенням об'єкта вказано зарезервоване слово **`var`** (від англ. *variables* – змінні), що інформує компілятор про початок розділу опису змінних.

Якщо на етапі проектування програміст розміщає на формі новий компонент, то в текст модуля автоматично вставляється опис цього компонента та за додатковою командою створюються заготівки. Справа програміста – наповнити опрацьовувачі подій конкретним змістом у вигляді операторів. Програміст може оголошувати нові змінні, типи, константи і т. д., однак видаляти та змінювати рядки, вставлені автоматично, не рекомендується.

Якщо в інтерфейсній частині модуля необхідно виконати підключення додаткового модуля, то в цьому може допомогти браузер коду. Для внесення змін за допомогою браузера коду потрібно вибрати відповідний розділ (у даному випадку Uses) і клікнути над ним правою кнопкою мишки. У результаті відкриється контекстне меню, у якому треба вибрати пункт New (Додати). Після вибору пункту New (Додати) у дереві, відображуваному в браузері коду, створиться новий елемент, якому необхідно надати ім'я відповідно до імені модуля, що додається.

Аналогічно в інтерфейсній частині модуля можна проводити додавання процедур, функцій, змінних і констант, а також змінювати визначення класів.

2.7. Проектування програми

Спроектуємо програму для розв'язання деякої задачі (приклад 2.2).

```
//Приклад 2.2
//Ввести число. Вважаючи його послідовно довжиною сторони
//квадрата, радіусом кола або кулі обчислити, і вивести
//на екран 1) площу квадрата, 2) площу круга й 3) об'єм кулі.
```

Завантаживши Delphi, виберемо команду File ► New ► Application (Файл ► Створити ► Додаток), у результаті чого буде створена порожня форма, про яку йшлося вище. Захоплюючи по черзі межі форми при натиснутій лівій кнопці миші, підберемо зручні розміри форми.

Далі слід розмістити на формі компоненти відповідно до вимог задачі. Задача передбачає наявність засобів із забезпечення введення числа, виведення результатів обчислень та тих, що безпосередньо реалізують процес обчислень. Природна річ, введення даних завжди повинне супроводжуватися змістовними повідомленнями-підказками.

Виберемо на вкладці Standard компонент Panel (Панель) і клацнемо мишкою над формою. У результаті на формі з'явиться компонент Panel (лівий верхній кут компонента буде розташовуватися в точці, над якою здійснювався клік). Перетягнемо панель мишкою в нижню частину форми, розтягши її на всю ширину форми і задавши висоту, що дорівнює приблизно 0,2 від висоти форми (панель можна розташовувати в будь-якому місці форми). Вона буде служити контейнером, у якому розмістяться інші компоненти проекту (насамперед вікно редактора для введення даних і кнопок для подачі команд на початок обчислень та про завершення роботи програми). У принципі без панелі можна обійтися, розмістивши згадані елементи на вільних місцях форми, що є контейнером для всіх компонентів.

Аналогічним способом виберемо на вкладці Standard компоненти Label, Edit та Button, а на вкладці Additional – компонент BitBtn. Розмістимо ці компоненти на панелі (для чого будемо клікати мишею над панеллю) і встановимо для них потрібні розміри і місце розташування.

Компонент Label (Мітка) призначений для виведення різних повідомлень (підказок, результатів обчислень і т. д.); ми будемо його використовувати для виведення підказок, що випереджають введення. Для зміни тексту, що міститься в цьому компоненті, досить змінити значення його властивості Caption, що можна робити як на етапі проектування програми, так і програмними засобами.

Компонент Edit (однорядкове редаговане текстове поле) застосовують для введення та (або) відображення досить довгих текстових рядків. Для зміни тексту, що міститься в ньому, достатньо змінити значення його властивості Text, що також можна виконувати як на етапі проектування програми, так і програмними засобами. Ми будемо використовувати цей компонент для введення числа.

Компонент Button (Кнопка) призначений для керування програмою. Ми вдамося до нього для організації обчислень і забезпечення виведення.

Компонент BitBtn (Кнопка з зображенням) є різновидом кнопки Button. Ми застосуємо один з варіантів цього компонента, який служить для видачі сигналу про припинення роботи програми. Ця кнопка не є обов'язковою, оскільки припинення роботи програми забезпечується кліком над кнопкою закриття форми. Замість цього компонента можна використовувати і кнопку Button.

Нарешті, для виведення результату обчислень помістимо на форму багаторядкове редаговане текстове поле, для чого виберемо на вкладці Standard компонент Memo (Текстове поле, від *memo field*) і клацнемо мишкою над вільним місцем форми (поза панеллю). Компонент Memo вживають для введення, редагування й (або) відображення досить довгих текстів, які можуть містити кілька рядків. Цей текст може корегуватися як на етапі візуального проектування, так і програмно, для чого необхідно змінювати властивість Lines (Рядки) цього компонента.

У результаті попереднього проектування програми вікно форми може набути, наприклад, вигляду, наведеного на рис. 2.9.

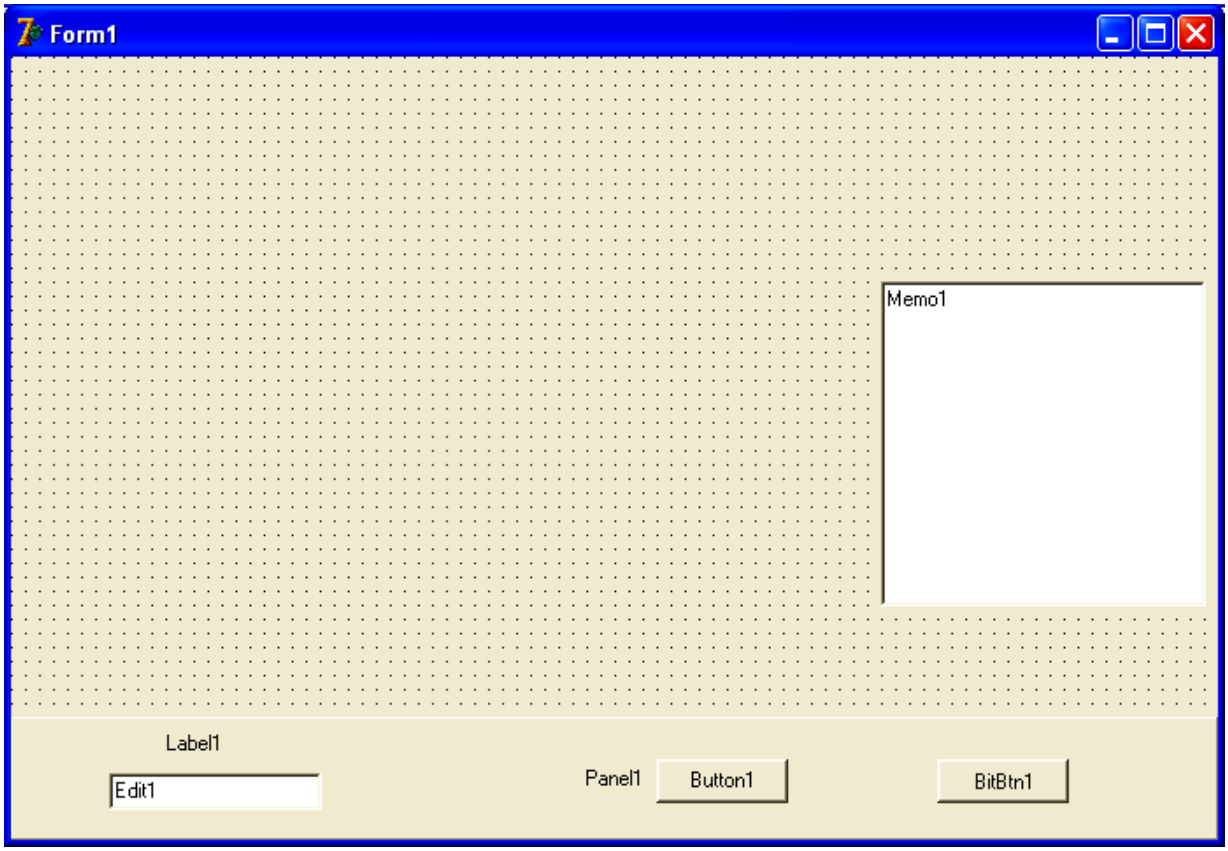


Рис. 2.9. Можливий початковий вигляд форми

Відповідно до специфіки розв'язуваної задачі здійснимо такі зміни властивостей компонентів:

- Форма:
  Position — poScreenCenter
  Caption — Приклад 2.2
- Панель:
  Align — alBottom
  BevelOuter — bvNone

Caption — очистимо

- Мітка:
  Caption — Уведіть число і натисніть "Обчислити"
  Name — lbOutput1
- Поле введення:
  Name — edInput1
  Text — 0
- Багаторядкове поле:
  Align — alClient
  Lines — очистимо
  Name — mmOutput1
  ScrollBars — ssBoth
- Кнопка Button1:
  Caption — Обчислити
- Кнопка Close:
  Kind — bkClose
  Name — bbClose

У результаті форма набуде вигляду, зображеного на рис. 2.10.

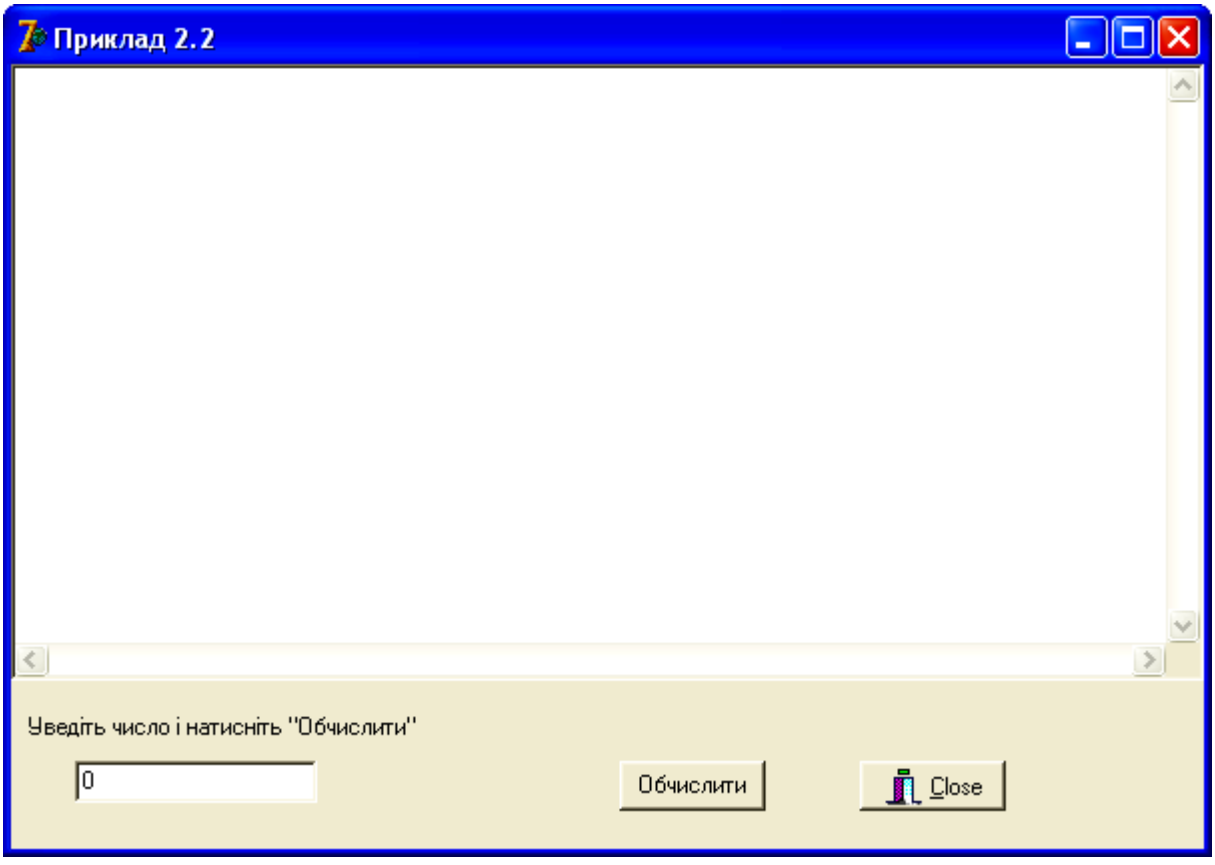


Рис. 2.10. Можливий остаточний вигляд форми

Розглянемо описані вище зміни.

Властивість Position (Положення) у форми визначає її положення щодо меж екрана (див. далі підрозд. 16.1). За умовчанням ця властивість має значення poDesigned й означає розташування вікна так, як воно було розміщене на етапі конструювання. Наведене значення poScreenCenter для властивості Position зумовлює розташування форми в центрі екрана.

Властивість Align (Вирівнювання) визначає спосіб розміщення даного компонента щодо того контейнера, який його містить (див. далі п. 15.3.2). Наприклад, панель розташована усередині форми, а кнопки, мітка і поле введення – усередині панелі. Значення alBottom у цієї властивості, задане для панелі, забезпечує притиснення останньої до нижньої межі форми з розтягуванням її по всій довжині форми. У багаторядкового поля властивість Align дістала значення alClient, що «примушує» компонент зайняти всю частину форми, яка залишилася незаповненою (вона називається клієнтською областю).

Оскільки в компонента може бути зовнішня крайка, її досить часто усувають, щоб компонент не виділявся на фоні свого контейнера. Зміна вигляду зовнішньої крайки забезпечується зміною значення властивості BevelOuter (див. далі п. 15.4.2). У панелі їй надане значення bvNone – немає крайки.

Властивість Name (Ім’я) визначає ім’я, під яким компонент буде існувати в програмі (див. далі підрозд. 15.1). У більшості компонентів ця Властивість змінена з метою додавання імені значеннєвого навантаження (lbOutput1 – мітка (Label) виведення, edInput1 – редактор (Edit) введення, mmOutput1 – поле (Memo) виведення, bbClose – бітова кнопка (BitBtn) закриття).

Властивість Caption (Заголовок) є в багатьох компонентів Delphi і служить для виведення заголовка (див. далі п. 15.3.4). Вона змінена у трьох компонентів: у форми – для виведення інформації про номер приклада, у панелі – для знищення заголовка, у мітки – для виведення підказки .

Властивість Text (Текст) у поля введення (див. далі п. 17.2.1) і властивість Lines (Рядки) у багаторядкового поля (див. далі п. 17.2.4) визначають текст, що буде міститися в однорядковому або багаторядковому полі в момент їхньої появи на екрані. Для компонента Edit у властивість Text записане значення 0, щоб навіть за відсутності введеного числа за нього було взяте значення 0. У багаторядкового поля властивість Lines очищена.

Значення ssBoth (Обидві) властивості ScrollBars у компонента Memo забезпечить розміщення в цьому компоненті двох смуг прокручування (скролінга) з метою можливого перегляду текстів, що вийшли за межі компонента (див. п. 17.2.4).

Значення bkClose властивості Kind (Сорт) у компонента BitBtn означає наявність типового значка і напис над кнопкою, а також зв'язок з кнопкою функції закриття вікна (див. підрозд. 18.2).

Зміна значень властивостей буде автоматично приводити до зміни тексту модуля. В результаті текст модуля прийме наступний вид:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes,Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, ExtCtrls;
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    lbOutput1: TLabel;
    edInput1: TEdit;
    Button1: TButton;
    bbClose: TBitBtn;
    mmOutput1: TMemo;
    private
      { Private declarations }
    public
      { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.
```

Як це видно з наведеного вище тексту модуля, Delphi автоматично вставив у клас розміщені на формі компоненти, надавши їм імена або стандартні (якщо компонент не перейменовувався при проектуванні форми) – Panel1 та Button1, або ті, які були дані їм програмістом (lbOutput1, edInput1, bbClose, mmOutput1). Перелік вставлених нами компонентів і становить більшу частину опису класу. Для завершення проектування програми залишилося доповнити її елементами, що власне і

забезпечують організацію обчислень і виведення даних. Тут також багато в чому допомагає ICP.

Виконаємо подвійний клік мишкою над зображенням форми в панелі Дерево Об'єктів або над будь-яким вільним місцем форми в її вікні. У відповідь на клік мишкою Delphi автоматично вставить у код модуля перед термінатором **end** з крапкою такий код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin

end;
```

Це код так званого опрацьовувача події OnCreate (За створенням) для форми.

Слово **procedure** вказує, що опрацьовувач є так званою підпрограмою-процедурою. Складене ім'я `TForm1.FormCreate` – це ім'я процедури, що складається з двох частин: імені ***класу*** `TForm1` і власне імені процедури `FormCreate`. Після імені процедури в круглих дужках зазначено опис параметра, з яким вона буде викликатися (`Sender: TObject`). У нашій програмі параметр `Sender` використовуватися не буде. У більш складних програмах за допомогою цього параметра програміст може визначити, який компонент згенерував дану подію (у даному випадку подія OnCreate). У загальному випадку у процедури може бути кілька параметрів; вони можуть також бути відсутні. Перший рядок розглянутого фрагмента коду називається ***заголовком процедури*** (він завершується символом «крапка з комою»).

Слідом за заголовком процедури розташовується її ***тіло*** (у даному випадку воно порожнє й містить тільки так звані операторні дужки **begin** та **end**). Щоб опрацьовувач події виконував які-небудь дії, його тіло повинне бути наповнене операторами. Відзначимо, що відразу ж за заголовком може розміщуватися секція описів, у якій програміст описує допоміжні елементи, необхідні для реалізації тіла процедури.

Крім вставки коду опрацьовувача події, Delphi автоматично модифікує опис класу, вставляючи в нього перед словом **private** заголовок опрацьовувача події:

```
procedure FormCreate (Sender: TObject);
```

Модифікуємо код опрацьовувача події OnCreate форми, надавши йому такого вигляду:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Caption := 'Приклад 2.2';
```

```
   DecimalSeparator := '.';
end;
```

Перший оператор цієї процедури змінює стандартний текст заголовка форми на текст `Приклад 2.2`. Цей текст буде виведений у верхній частині форми. Про другий оператор потрібно поговорити особливо. Справа в тому, що якщо використовується русифікована версія Windows, то в ній як роздільник цілої та дробової частин дійсного числа використовується не десяткова точка, як це має місце в мовах програмування (у тому числі і в Delphi), а кома. Результатом цього може бути неправильна робота деяких підпрограм, що перетворюють рядки до дійсних чисел (наприклад, `StrToFloat`). Системна змінна `DecimalSeparator` містить символ, що використовується як роздільник цілої та дробової частин дійсних чисел. Другий оператор опрацьовувача події `FormCreate` на початку роботи програми змінює цей символ на символ «крапка», що використовується в Delphi та у не русифікованій версії Windows.

Клікнемо тепер мишкою над кнопкою Обчислити. У відповідь на клік мишкою Delphi автоматично вставить у код модуля перед термінатором **end** з крапкою такий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Це код так званого опрацьовувача події OnClick (За кліком) для кнопки Обчислити. Взагалі кажучи, при кліку мишею в працюючій програмі виникає подія OnClick, що зв'язується з компонентом, над яким був здійснений клік.

Якщо опрацьовувач події відсутній, то подія ніяк не опрацьовується; за умови його наявності клік мишкою активізує опрацьовувач і забезпечує виконання дій, записаних у цьому опрацьовувачі.

Крім того, в опис класу перед словом **private** буде автоматично вставлений заголовок опрацьовувача події:

```
procedure Button1Click(Sender: TObject);
```

Модифікуємо код процедури `TForm1.Button1Click`, надавши йому такого вигляду:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  X := StrToFloat(edInput1.Text);
  mmOutput1.Lines.Add('X=' + FloatToStr(X));
```

```
  mmOutput1.Lines.Add('Площа квадрата дорівнює '
                      + FloatToStr(Sqr(X)));
  mmOutput1.Lines.Add('Площа круга дорівнює '
                      + FloatToStr(Pi * Sqr(X)));
  mmOutput1.Lines.Add('Об''єм кулі дорівнює '
                      + FloatToStr(Pi * X * X * X / 4));
  lbOutput1.Caption := 'Уведіть число' +
     ' і натисніть "Обчислити" або припиніть обчислення';
  edInput1.SetFocus;
end;
```

Додамо також в інтерфейсну частину модуля після рядка

```
Form1: TForm1;
```

рядок

```
X: Real;
```

Цей рядок описує змінну з ім'ям `X`, що служить для зберігання значень реального (дійсного) типу. Ця змінна буде використовуватися у програмі для зберігання числа, що вводиться. Подібного роду опис дозволяє надалі використовувати ім'я `X` усюди в модулі нижче точки оголошення з можливістю зміни значення змінної, позначеної цим ім'ям.

Розглянемо призначення операторів, що утворюють тіло опрацьовувача подій `TForm1.Button1Click`.

Оператор

```
X := StrToFloat(edInput1.Text);
```

забезпечує перетворення тексту, що міститься у властивості `Text` поля введення, до дійсного числа і запис отриманого значення в змінну `X`.

Наступний оператор виводить у поточний рядок багаторядкового поля `mmOutput1` текст `X=` та перетворене до рядкового вигляду значення змінної `X`.

Третій оператор аналогічним способом виводить текст `Площа квадрата дорівнює` та перетворений до рядкового виду результат обчислення площі квадрата.

Далі виводиться текст `Площа круга дорівнює` та перетворений до рядкового виду результат обчислення площі круга, а також текст `Об'єм кулі дорівнює` та перетворений до рядкового вигляду результат обчислення об'єму кулі. Зазначимо, що для запису в рядок символу «апостроф» потрібно записати два апострофи підряд (безпосередньо в рядок при цьому буде записаний один апостроф).

Наголосимо, що оскільки дані вводяться в рядковому (а не числовому!) виді, введене значення необхідно привести до дійсного числа, для

чого в Delphi визначена функція `StrToFloat`. Аналогічно, оскільки в Delphi можливе виведення тільки текстових повідомлень, числові значення повинні бути перетворені до так званого рядкового виду (для подібного роду перетворення дійсного числа в Delphi визначена функція `FloatToStr`). В обчисленнях використана визначена в Delphi функція піднесення у квадрат дійсного значення (`Sqr`), а також функція без параметрів `Pi`, що служить для обчислення числа π.

Передостанній оператор змінює властивість `Caption` мітки `lbOutput1` з метою зміни підказки для введення значень.

Останній оператор тіла опрацьовувача `Button1Click` за допомогою методу `SetFocus` передає контроль над клавіатурою полю введення `edInput1` з метою забезпечення можливості введення нового числа без додаткових дій, якщо необхідно повторити обчислення без виходу з програми і її повторного запуску.

Таким чином, остаточний текст модуля набуває такого вигляду:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls, Buttons,
  ExtCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    lbOutput1: TLabel;
    edInput1: TEdit;
    Button1: TButton;
    bbClose: TBitBtn;
    mmOutput1: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    private
      { Private declarations }
    public
      { Public declarations }
  end;

var
  Form1: TForm1;
  X: Real;
```

```
implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
   Caption := 'Приклад 2.2';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  X := StrToFloat(edInput1.Text);
  mmOutput1.Lines.Add('X=' + FloatToStr(X));
  mmOutput1.Lines.Add('Площа квадрата дорівнює '
                      + FloatToStr(Sqr(X)));
  mmOutput1.Lines.Add('Площа круга дорівнює '
                      + FloatToStr(Pi  * Sqr(X)));
  mmOutput1.Lines.Add('Об''єм кулі дорівнює '
                      + FloatToStr(Pi * X * X * X / 4));
  lbOutput1.Caption := 'Уведіть число' +
     ' і натисніть "Обчислити" або припиніть обчислення';
  edInput1.SetFocus;
end;

end.
```

Запитання для контролю і самоконтролю

1. На які етапи розділяється процес програмування в ICP Delphi? Дайте їм стислу характеристику.
2. Перелічіть найбільш важливі вікна ICP Delphi.
3. Для чого призначене головне вікно Delphi?
4. Чи можна видалити яку-небудь з панелей головного вікна?
5. Яким способом можна перемістити на нове місце яку-небудь панель головного вікна?
6. Чи можна закрити головне вікно, залишившись у середовищі Delphi?
7. Що таке палітра компонентів?
8. Для чого призначене вікно дерева об’єктів Delphi?
9. Для чого використовується інспектор об’єктів?
10. Опишіть структуру інспектора об’єктів.
11. У чому відмінність простої властивості від складної?
12. Як використовується кнопка  у рядку для простої властивості?

13. Як можна розрізнити просту та складну властивості в інспекторі об'єктів?
14. Що таке вікно коду?
15. Для чого служить браузер коду?
16. Як здійснюється перехід з вікна форми у вікно коду і назад?
17. Опишіть структуру файлу проекту.
18. Дайте визначення терміну «ім'я».
19. Опишіть структуру модуля.
20. У якому файлі зберігається інформація про структуру форми?
21. Які розширення мають файл проекту і файл модуля форми?
22. За допомогою яких засобів у середовищі Delphi можна створити новий додаток (нову програму)?
23. Яким способом можна розмістити компоненти на формі?
24. Що треба зробити, якщо положення компонента або які-небудь його інші характеристики не задовольняють проектувальника додатка?

Завдання для практичного відпрацьовування матеріалу

1. Створіть форму і розташуйте на ній кнопку і панель. Виконайте такі дії:

   а) перемістіть кнопку на панель;

   б) здійсніть зворотне перенесення кнопки.

   Підказка. Попрацюйте з Деревом Об'єктів.

2. Завантажте модуль з прикладу 2.2. Знайдіть, де в коді модуля описана глобальна змінна `X`.

   Підказка. Скористуйтеся Браузером Коду.

3. Завантажте модуль з прикладу 2.2. За допомогою Браузера Коду перегляньте список модулів, які підключені до даного модуля.
4. Створіть новий додаток, що містить порожню форму. Перегляньте текст проекту. **Підказка**. Попрацюйте з пунктом меню Project.
5. Для завдання 3 після завантаження у вікно коду тексту проекту здійсніть перехід до тексту модуля. Закрийте сторінку з текстом проекту.
6. Створіть новий додаток, що містить форму з компонентом Memo, який має встановлені за умовчанням властивості, і запустіть програму. Клацніть лівою кнопкою миші над Memo-компонентом і переконайтеся, що в його вікні можна виконувати набір тексту. Наберіть текст, деякі з рядків якого виходять за межі вікна. Переконайтеся в тому, що в

даному випадку є можливість переглядати і змінювати текст, навіть якщо він виходить за межі вікна. Що відбудеться, якщо вертикальний розмір вікна не дозволяє помістити всі рядки тексту?

7. Виконайте попереднє завдання, змінивши значення властивості ScrollBars на ssBoth.
8. Відкрийте проект для прикладу 2.2 і перевірте, як зміниться робота програми, якщо в остаточний варіант тексту модуля в опрацьовувач події OnClick компонента Button1 безпосередньо перед словом **end** буде вписаний рядок

```
Button1Click.Enabled := False;
```

9. Виконайте попереднє завдання 8, але впишіть інший рядок:

   а) `Button1Click.Visible := False;`

   б) `Panel1.Visible := False;`

   в) `Panel1.Enabled := False;`

   г) `Button1.Width := 2 * Button1.Width;`

10. Не змінюючи коду модуля з прикладу 2.2, установіть для властивості Enabled компонента BitBtn значення False і запустіть програму. Що змінилося в її роботі?
11. Для остаточного варіанта форми з прикладу 2.2 (див. рис. 2.2) виконайте вставку ще однієї панелі, яка повинна розміститися у верхній частині форми, розтягшись на всю ширину останньої. Зробіть так, щоб компонент Memo зайняв всю частину форми, що залишилася вільною.

3. БАЗОВІ ЕЛЕМЕНТИ МОВИ DELPHI

3.1. Алфавіт мови

Символи мови Delphi – літери, арабські цифри і спеціальні символи – становлять його алфавіт.

У мові Delphi використовуються великі і малі ***літери*** латинського алфавіту від a до z та від A до Z, а також символ підкреслення (_). При цьому, за винятком коментарів і рядків, великі та малі літери не розрізняються.

Цифри – це арабські цифри від 0 до 9, а також ***шістнадцяткові цифри***, у яких перші десять значень позначають цифрами від 0 до 9, а решту шість – латинськими літерами від A до F або від a до f.

Спеціальні символи – це

а) знаки операцій + − * / = < >

б) обмежувачі . , ( ) [ ] { } : ; '

в) спеціальні знаки $ @ # & ^ ~

Є також складені спеціальні символи. До них відносять такі ***пари*** символів, як <> <= >= := (* *) (. .) //

Застосовується, крім того, символ «пробіл», що є обмежувачем. Як обмежувачі розглядаються також усі символи з десятковими кодами до 31 (код 32 відповідає пробілу), які досить часто називають узагальненими пробільними символами.

Рекомендується активно використовувати пробіли, символи табуляції, а також перехід до нового рядка для написання програм, які будуть легко сприйматися людиною. Вживання пробільних символів у тексті програми (за винятком рядкових констант) не позначається на результатах її компіляції та виконанні.

У деяких випадках (у коментарях і рядках, а також як окремі символи) можуть бути використані літери українського (або російського) алфавіту і всі інші символи, як наявні на клавіатурі, так і відсутні на ній.

Досить часто до алфавіту відносять і зарезервовані слова:

and	**exports**	**mod**	**shr**
array	**file**	**nil**	**string**
as	**finalization**	**not**	**then**
asm	**finally**	**object**	**threadvar**
begin	**for**	**of**	**to**
case	**function**	**or**	**try**
class	**goto**	**out**	**type**
const	**if**	**packed**	**unit**
constructor	**implementation**	**procedure**	**until**
destructor	**in**	**program**	**uses**
dispinterface	**inherited**	**property**	**var**
div	**initialization**	**raise**	**while**
do	**inline**	**record**	**with**
downto	**interface**	**repeat**	**xor**
else	**is**	**resourcestring**	
end	**label**	**set**	
except	**library**	**shl**	

Службові (зарезервовані) слова – це імена (слова мови), які мають спеціальне раз і назавжди закріплене за ними значення. Їх не можна вживати ніяк інакше (наприклад, не можна позначити змінну або константу ім'ям **end**). Зарезервовані слова вказують компілятору на необхідність виконання певних дій.

3.2. Стандартні директиви і визначені імена

У мові Delphi існують також інші імена, що мають початкове спеціальне (заздалегідь задане) значення. Вони спочатку пов'язані з деякими стандартними оголошеннями, що зустрічаються в програмах. Такі імена називаються ***стандартними директивами***.

Програміст може обходити закріплені за стандартними директивами значення і використовувати їх для позначення яких-небудь об'єктів програми (наприклад, як імена змінних). Якщо програміст не визначить явно, для яких цілей застосовується в програмі те або інше ім'я стандартної директиви, воно буде сприйматися у властивому йому значенні. Список стандартних директив такий:

absolute	**external**	**package**	**requires**
abstract	**far**	**pascal**	**resident**
assembler	**forward**	**private**	**safecall**
automated	**implements**	**protected**	**stdcall**
cdecl	**index**	**public**	**stored**
contains	**message**	**published**	**virtual**
default	**name**	**read**	**write**
dispid	**near**	**readonly**	**writeonly**
dynamic	**nodefault**	**register**	
export	**override**	**reintroduce**	

Особливістю стандартних директив **private**, **protected**, **public**, **published** та **automated** є те, що вони вважаються зарезервованими усередині оголошень класів; поза оголошеннями класів вони можуть бути перевизначені.

Службові слова і стандартні директиви у вікні коду автоматично виділяються жирним шрифтом.

Визначеними (***стандартними***) ***іменами*** є й деякі інші імена, наприклад, пов'язані з системою типів (приміром, визначеним є ім'я `Integer`), а також імена констант, змінних, процедур і функцій, визначених у стандартних модулях. На відміну від стандартних директив, ці імена у вікні коду автоматично не виділяються. Значення таких імен програміст може перевизначити.

3.3. Ідентифікатори та змінні

Усі імена, за винятком службових слів, є іменами, обумовленими програмістом, або, інакше, – ***користувальницькими іменами***. Вони повинні бути вичерпно оголошені в програмі.

Як про це говорилося в підрозд. 2.5, для формування імен використовуються латинські літери, цифри і символ підкреслення, причому ім'я не може починатися з цифри.

З поняттям користувальницького імені асоціюється поняття ідентифікатора.

Ідентифікатори – це імена, якими програміст позначає будь-який інший елемент програми, крім зарезервованого слова або коментарю.

Ідентифікатори в мові Delphi – це імена констант, змінних, міток, типів, класів, властивостей, процедур, функцій, модулів, програм і полів у записах і класах. Довжина ідентифікатора може бути довільною.

Хоча це і не обов'язково, але при виборі імені варто виходити з його значеннєвого навантаження. Вдало обране ім'я полегшує людині сприй-

няття тексту програми. Якщо ж ім'я невдале (безлике), то деякий час навіть автору програми буде дуже складно розібратися в її тексті. Що вже говорити про стороннього користувача? Для комп'ютера ж (точніше, для компілятора) абсолютно немає різниці, які імена використовував автор програми.

Досить часто з метою підвищення читаності програми вдаються до імен, що складаються з декількох слів або їхніх скорочень (наприклад, `myage`). Оскільки вживання пробілу усередині імені заборонено, замість нього для розділення складових частин імені може бути поставлений символ підкреслення (`my_age`), завдяки чому ім'я читається легше.

Багато програмістів замість того, щоб використовувати символ підкреслення, просто починають другу частину імені з великої літери (наприклад, `camelHump`). Таку форму запису називають «верблюжим поданням», оскільки одна велика літера усередині слова нагадує верблюжий горб.

Однобуквених безликих імен треба уникати, вживаючи їх тільки для позначення тимчасових змінних.

У програмі, що написана мовою Delphi, повинне бути забезпечене використання комірок пам'яті, для чого служать змінні.

Змінна – це ім'я фізичної ділянки пам'яті, у якій у будь-який момент часу може зберігатися тільки одне значення, причому вміст цієї ділянки пам'яті в ході виконання програми може змінюватися. Для розрізнення змінних удаються до застосування ідентифікаторів.

Щоб можна було користуватися змінною, її необхідно описати.

У деяких (а точніше, у багатьох) випадках змінна йменується не за допомогою ідентифікатора, а більш складним способом. У такому разі іноді говорять про ***уточнене ім'я***. За допомогою уточнених імен здійснюється звертання до елементів масивів, полів записів, об'єктів і класів. Свої особливості має і методика звертання до так званим динамічних змінних (змінних, створюваних у ході виконання програми).

3.4. Константи

Ще одним елементом мови є константи, з якими, насамперед, асоціюються числа.

Константи визначають області пам'яті, які не можуть змінювати свого значення в ході роботи програми. У програмі, написаній мовою Delphi, константи можуть мати свої власні імена. Тип константи визначається способом її запису й легко розпізнається компілятором у тексті програми, тому програміст може не використовувати іменовані константи, тобто не оголошувати їх у програмі явно з наданням імен.

Десяткові числа завжди починаються з цифри, перед якою може стояти знак числа (+ або −).

Дійсні (***реальні***) ***числа*** зображуються у двох форматах. У форматі з фіксованою точкою явно вказують положення десяткової точки (наприклад, `1.002`, `−12.34`, `+1.0`). У форматі з плаваючою точкою (експонентна форма) використовується десятковий порядок, позначуваний великою або малою літерою `E` (`e`), слідом за якою йде ціле число, що показує значення порядку, наприклад, `15e14`, `+12.6e−1`, `1.0E+03`. Значення, що стоїть перед `E` (`e`), називається *мантисою*, а після `E` (`e`) – *порядком* (він може мати знак). Сам символ `E` (`e`) в експонентній формі означає десятковий порядок: число подається значенням, отриманим множенням мантиси на `10` у степні, заданому порядком.

Якщо в записі дійсного числа вжито десяткову точку, то за нею і перед нею обов'язково потрібно вказати хоча б по одній цифрі; аналогічно хоча б одна цифра має бути зазначена після символу `E` (`e`) у випадку використання експонентної форми запису дійсного числа.

Цілі числа в Delphi можуть записуватися як у десятковому, так і у шістнадцятковому вигляді.

У ***десятковому*** вигляді цілі числа записуються звичайним способом (зі знаком або без). Діапазон подання цілих чисел у Delphi – від -2^{63} до $+2^{63}-1$ (від `−9223372036854775808` до `+9223372036854775807`). ***Шістнадцяткове*** число складається з шістнадцяткових цифр, яким передує знак долара `$`. Діапазон шістнадцяткових чисел – від `$0000000000000000` до `$FFFFFFFFFFFFFFFF`. Відзначимо, що шістнадцятковим числам від `$0000000000000000` до `$7FFFFFFFFFFFFFFF` відповідають додатні десяткові числа від `0` до `9223372036854775807`, а числам від `$8000000000000000` до `$FFFFFFFFFFFFFFFF` відповідають від'ємні значення від `−9223372036854775808` до `−1`.

У Delphi визначено дві константи `MaxInt` та `MaxLongInt`, що відповідають цілому числу `2147483647`.

Булівська (***логічна***) константа задається одним з визначених імен – `False` (неправда) або `True` (істина).

Символьна константа – це будь-який символ, укладений в апострофи. Можна також замість символу зазначати його десятковий або шістнадцятковий код з попереднім символом #. Для зазначення символу апострофа його потрібно подвоїти (`''''`).

Текстовим літералом (***рядком***) у Delphi називають послідовність будь-яких допустимих символів, що стоять між апострофами (наприклад,

'Delphi 7'). Якщо як символ рядка необхідно використати власне апостроф, то, як і у випадку символьних констант, його символьне позначення записують двічі (''). Рядок можна задати також у вигляді послідовності, утвореної з символу # з наступним цифровим кодом (наприклад, запис #88#89#90 еквівалентний рядку 'XYZ'), або комбінуванням таких послідовностей і рядків в апострофах:

```
#219'Delphi'#219'7'#219
```

У рядкових даних великі і малі літери розрізняються. Рядок може бути порожнім, тобто не містити символів між апострофами. Максимальна довжина рядкової константи дорівнює 255.

У Delphi визначена також ***константа-вказівник***. Вона задається службовим словом **nil** і позначає відсутню адресу (невизначений вказівник). Константою є також ***конструктор множини***, що задається укладеним у квадратні дужки списком елементів множини.

3.5. Система типів

Типи – це спеціальні конструкції мови, за допомогою яких компілятор створює такі елементи програми, як змінні, константи і функції. Тип визначає можливі значення змінних, констант, результату, що повертає функція, виразів, що належать даному типу, форму подання їх у машині, займаний ними обсяг пам'яті та операції, які можуть виконуватися над ними.

Щодо обсягу пам'яті, який відводиться під дані, і форми їхнього подання, відзначимо, що всі дані в комп'ютері подаються у двійковій системі числення. Як про це говорилося в підрозд. 1.1, мінімальною одиницею пам'яті, що адресується, є байт, який складається з 8 двійкових розрядів – бітів, які нумеруються від нуля. Два байти, що стоять поруч, молодший з яких має парну адресу, називаються *словом*. У байті, слові, подвійному слові перший (крайній) біт з початку області пам'яті є молодшим (номер 0), а останній (крайній) з кінця області пам'яті – старшим (з номером 7, 15 або 31 відповідно). При записі чисел зі знаком старший біт є знаковим.

Якщо програміст використовує нестандартну ідентифікацію (іменування) типу, то відповідний ідентифікатор повинен бути оголошений у розділі опису типів.

Найпростішими стандартними типами даних є Integer (цілий), Real (реальний, дійсний), Boolean (булівський), Char (символьний, літерний).

Система типів мови Delphi досить велика. Базовими є ***прості*** (***скалярні***) типи; складені типи за певними правилами будуються з простих. Стандартні прості типи розділяються на п'ять груп: ***цілі***, ***дійсні***, ***символьні***, ***булівські*** та тип ***дата-час***. До скалярного відносять також перелічені типи, які оголошуються користувачем. Будь-який скалярний тип характеризується його можливими значеннями, серед яких установлений лінійний порядок.

Усі скалярні типи, крім дійсних і типу ***дата-час***, називаються також ***порядковими*** (***дискретними***), оскільки кожному з їх значень можна поставити у відповідність його порядковий номер; у кожного значення є безпосередньо попереднє й наступне значення. Порядковим є також ***перелічений тип***, особливістю якого є те, що всі його можливі значення перелічуються явно й задаються за допомогою ідентифікаторів. ***Обмежені типи*** (***відрізки***, ***діапазони***) формуються з порядкових типів шляхом звуження області допустимих значень. Кількість можливих значень дійсних типів у силу дискретності подання даних у комп'ютері, природно, є скінченною, однак вона настільки велика, що поставити у відповідність кожному дійсному значенню порядковий номер неможливо. Тип ***дата-час*** подається дійсним значенням, у зв'язку з чим він не є порядковим.

До кожного зі значень порядкових типів можна застосовувати функцію `Ord(z)`, що повертає порядковий номер значення виразу `z`, заданого в дужках. Для цілих типів ця функція повертає саме значення, для булівських типів – значення `0` та `1` або `−1` та `0`, для символьного типу в кодуванні ANSI – значення від `0` до `255`, а для переліченого типу – значення від `0` до `65535`. Функція не може повернути значення поза діапазоном типу `Int64` (див. п. 3.13.1)

До даних порядкового типу застосовна також функція `Pred(z)`, що повертає попереднє щодо `z` значення:

```
Ord(Pred(z)) = Ord(z) - 1.
```

Визначена також функція `Succ(z)`, яка повертає попереднє стосовно `z` значення:

```
Ord(Succ(z)) = Ord(z) + 1.
```

Крім простих, у Delphi є ***складені*** типи (масиви, рядки, записи, множини, файли), формування яких засноване на використанні інших типів. З будь-яких типів можуть бути утворені ***посилальні*** типи (***вказівники***). В особливу групу можна виділити ***процедурні типи***. Виділяються також такі типи, як ***об'єкти*** і ***класи***. Крім того, у Delphi є так звані ***варіантні*** типи, у тому числі ***налаштовувані варіанти***. На основі будь-яких типів можна утворити ***користувальницькі*** типи.

3.6. Змінні і їх оголошення

Змінні служать для позначення ділянок пам'яті, вміст яких може змінюватися в ході роботи програми. На відміну від констант, змінні завжди оголошуються в програмі, оскільки для того щоб у програмі можна було використати який-небудь ідентифікатор, його потрібно оголосити до першого застосування. При цьому треба пам'ятати, що одне і те саме ім'я не може бути оголошене двічі усередині однієї програмної одиниці.

Для оголошення змінних служить спеціальний розділ (їх може бути декілька), що починається зі службового слова **var**, за яким ідуть оператори оголошення окремих змінних. При оголошенні змінної вказуються її ім'я і (після двокрапки) тип. Якщо потрібно оголосити декілька змінних одного типу, то їхні імена можуть бути перелічені через кому в одному операторі:

```
var
  number, i, j: Integer;
```

Розділ оголошення змінних подає інформацію для розподілу пам'яті під використовувані в програмі змінні.

Якщо змінна глобальна (описана поза підпрограмою), то в операторі опису можна задати її початкове значення, тобто значення якого прибере ця змінна в момент відведення під неї пам'яті:

```
var
  r: Real = 2.7;
```

Надати початкове значення змінній в операторі опису (здійснити ініціалізацію змінної) можна тільки в тому випадку, коли в ньому описується одна змінна.

За умовчанням глобальні змінні ініціалізуються нулем; початкове значення змінної, описаної усередині підпрограми, не визначено.

Змінну, яка оголошується, можна накласти в пам'яті на іншу раніше оголошену змінну, для чого служить директива **absolute**. У загальному випадку типи (а отже, і розміри) змінних, що накладаються у пам'яті одна на одну, не обов'язково повинні збігатися.

При оголошенні змінної з використанням директиви **absolute** в операторі опису змінної після імені типу оголошуваної змінної вказують службове слово **absolute** і далі ім'я раніше розміщеної в пам'яті змінної.

Наприклад, у наступному фрагменті програми рядок `st` накладається в пам'яті на змінну `StLength`, а оскільки в першому байті змінної типу **string**`[]` зберігається символ, що кодує довжину рядка (див. п. 5.2.1), то це значення завжди можна визначити, скориставшись змінною `StLength`:

```
var
  st: string[100];
  StLength: Byte absolute st;
```

Розміщення змінної в пам'яті за директивою **absolute** зводиться до її накладення на перший байт змінної, котра вже розміщена. Щоб при роботі зі змінними, які накладаються одна на одну, не псувалися значення інших змінних, потрібно накладати коротші змінні на вже розміщені довші змінні, як це зроблено в наведеному вище фрагменті програми.

3.7. Мітки

Мітки відіграють роль імен операторів програми. Активне використання міток, вважається поганим стилем програмування. Мітки, якщо вони вживаються у програмі, в обов'язковому порядку повинні бути оголошені. Розділ оголошення міток починається зі службового слова **label** (мітка).

Про методику опису і використання міток говориться нижче в підрозд. 4.3.

3.8. Підпрограми

Підпрограми – це фрагменти програми, оформлені спеціальним способом. Підпрограми дозволяють структурувати програму; написання великих програм без використання підпрограм говорить про поганий стиль програмування. Програма не повинна залежати від особливостей реалізації підпрограм – підпрограми слід писати так, щоб їх можна було переносити з одних програм в інші і створювати бібліотеки підпрограм.

У Delphi передбачено підпрограми-процедури і підпрограми-функції. При звертанні до підпрограми зазначають її ім'я, а в дужках через кому перераховують параметри, від яких залежить результат обчислення (наприклад, щоб обчислити синус деякого значення, це значення вказується як параметр).

Прикладом підпрограми є функція `MessageBeep`, що має один параметр і служить для видачі звукового сигналу. Вона має один параметр – ціле число, яке звичайно задається іменованою константою, що визначає тип видаваного звуку (`MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION`, `MB_OK`).

Відзначимо, що наявність декількох підпрограм з одним і тим самим ім'ям у програмах, написаних мовою Delphi, є можливою (див. підрозд. 8.10).

3.9. Коментарі та директиви компілятору

У програмі можуть використовуватися коментарі. ***Коментар*** – це текст, розміщений між символами *{* та *}* або між парами символів *(** та **)*. Він розташовується в будь-якім місці, де може стояти пробіл, у тому числі заміняти пробіли. Якщо коментар розташовується наприкінці рядка, то його звичайно починають з символів *//*, не беручи у фігурні дужки (такий коментар називають однорядковим). Відзначимо також, що як коментар можна розглядати й текст, розміщений після термінатора **end** з крапкою, оскільки цей текст ігнорується компілятором.

У вікні коду коментарі звичайно виділяються курсивним написанням.

Фрагменти коду, що мають вигляд коментарів, у яких відразу ж (без пробілу) після відкриваючої фігурної дужки стоїть символ `$`, називається ***директивою компілятору***. Директива компілятору вказує йому на необхідність виконання деяких дій до того, як буде здійснена компіляція програми.

У Delphi існує три види директив компілятору: директиви-перемикачі, директиви з параметрами й умовні директиви. Всі вони оформляються у вигляді рядка коментарів, у якому першим символом (без пробілів) є символ `$` (наприклад, `{$I+}`). ***Директиви-перемикачі*** вказують на те, що компілятор повинен виконати або не виконати деякі дії. Активізація директиви позначається символом «+», вимикання – символом «−» після імені директиви. Директиви-перемикачі можна задавати роздільно або спільно. В останньому випадку вони відокремлюються одна від одної комою, а символ `$` ставиться один раз `{$R+,I-,O+}`. Якщо при задаванні директив-перемикачів де-небудь буде поставлений пробіл, то весь інший текст у фігурних дужках сприймається як коментар. У ***директивах з параметрами*** зазначають параметри (ім'я файлу, розміри пам'яті й т. д.), які передаються компілятору до компіляції. ***Умовні директиви*** використовуються при налаштуванні для перевірки окремих фрагментів програми.

3.10. Вирази й операції

Вираз – це послідовність операндів, роздільників і знаків операцій, яка задає порядок обчислення нового значення. У складному виразі порядок виконання операцій визначається ***пріоритетами*** (***рангами***) операцій. Для зміни порядку виконання операцій, як і у звичайній алгебрі, застосовуються круглі дужки. Операндами у виразах є константи, змінні, звертання до функцій, частині виразів, узяті в дужки. Тип значення виразу визначається типом операндів, що входять у нього, і видом операцій. Якщо

операнди мають різний тип, то здійснюється приведення результату обчислення до більш загального (поглинаючого) типу. Окремим випадком виразу є константа, змінна, звертання до функції.

Прикладами виразів є такі:

`t` – змінна;

`3.1415` – константа;

`Cos(x)` – звертання до функції;

`b * b - 4 * a * c` – алгебраїчний вираз;

`d<0` – вираз-відношення;

`not` `Flag` **`and`** `(F1` **`xor`** `F2)` – булівський вираз;

`(x + y <= b) = (x + y >= a)` – складний вираз.

Щоб визначити обсяг пам'яті, який виділяється під елемент даних, використовують функцію `SizeOf`, що має один параметр і повертає значення типу `Integer`. Цей параметр може бути ім'ям типу або виразом. Наприклад:

```
lbOutput1.Caption := IntToStr(SizeOf(Real));    //Виво-
                                          //диться значення 8
lbOutput1.Caption := IntToStr(SizeOf(n + 1));    //Якщо
          //n має тип Integer, то виводиться значення 4
```

Перелік операцій, які визначені в Delphi, і їхні пріоритети наведені в таблиці 3.1. При цьому менше значення пріоритету є більш високим з погляду порядку виконання операцій.

Таблиця 3.1 – Знаки та пріоритети операцій

Вид операцій	Операції	Пріоритет
Унарні	`-`, `+`, `@`, `^`, **`not`**	1 (вищий)
Мультиплікативні	`*`, `/`, **`div`**, **`mod`**, **`and`**, **`shl`**, **`shr`**, **`as`**	2
Адитивні	`+`, `-`, **`or`**, **`xor`**	3
Відношення	`=`, `<>`, `<`, `>`, `<=`, `>=`, **`in`**, **`is`**	4 (нижчий)

Операції одного пріоритетного рівня виконуються зліва направо. Остання фраза не зовсім правильна, оскільки компілятор, здійснюючи оптимізацію коду програми, може змінити порядок виконання операцій. Більш того, за умовчанням обчислення булівського виразу проводиться до моменту одержання значення, що не може змінитися при продовженні обчислень. Звичайна річ, в одному виразі практично неможливе одночасне використання всіх операцій, хоча вигадати такий вираз можна.

Операція «унарний мінус» є операцією зміни знака (наприклад, `-z`). Операція «унарний плюс» (наприклад, `+z`) введена з розумінь симетрії й звичайно не вживається. Зазначимо також, що деякі знаки операцій

задаються не одним символом, а складеним символом або службовим словом.

3.11. Оператори. Оператор присвоювання

Алгоритмічні дії, які необхідно виконати для розв'язання задачі, у програмі записуються за допомогою ***операторів*** (***тверджень***), які можуть записуватися в один або кілька рядків.

У деяких випадках виникає необхідність у використанні так званого ***порожнього оператора***, тобто оператора, що не виконує ніяких дій. Він позначається відсутністю оператора перед крапкою з комою або між службовими словами **then** та **else**.

Найпростішим способом надання змінній значення є виконання ***оператора присвоювання*** (призначення), що має вигляд

змінна := вираз

Порядок дій такий: спочатку обчислюється значення виразу, після чого результат записується в змінну:

```
k := 1;                  //У змінну k записане значення 1
number := k + 1;     //Змінна number дістала значення 2
```

При виконанні оператора присвоювання повинна забезпечуватися вимога сумісності щодо присвоювання типів лівої та правої його частин (див. п. 3.15.2).

Акцентуємо увагу на такому: змінна, **у яку** записується значення за допомогою оператора присвоювання, повинна знаходитися **ліворуч** від складеного символу :=.

3.12. Розділ опису типів

Програміст може визначити в програмі деякий нестандартний тип (такий тип часто називають ***користувальницьким***). При цьому виконуються два кроки: на першому в розділі опису типів, що починається зі службового слова **type**, задається зразок (шаблон) нового типу даних, на другому уведений новий тип застосовується в оголошенні змінних:

```
const
  limit = 47;
  len = 5;
type                                    //Розділ опису типів
  range = 1..limit;
  StringLen = string[len];
```

```
var
  number: range;    //Використання раніше оголошених...
  st    : StringLen;     //...типів range та StringLen
```

У даному випадку часто говорять про оголошення ***псевдонімів*** типів.

Розділ опису типів є важливим розділом у програмі, написаній мовою Delphi. По-перше, якщо тип заданий не ім'ям, а конструюється по ходу оголошення змінних (`var a: array[1..20] of Real`), то в програмі можлива несумісність типів у таких змінних, що на перший погляд мають один і той самий тип. Це може викликати помилки (в Delphi прийнята іменна еквівалентність типів, про що йтиметься далі в п. 3.15.1). По-друге, зручніше спочатку оголосити ім'я для якого-небудь типу, а потім це ім'я використовувати.

Можливе також оголошення ***типізованих псевдонімів***, для чого перед базовим типом в оголошенні його псевдоніма вказується службове слово **`type`**:

```
type
  TmyInt = type Integer;
```

Тут для компілятора типи `TMyInt` та `Integer` є різними типами, хоча й сумісними в різних виразах. У той же час, якщо оголошення типу має вигляд

```
type
  Int = Integer;
```

компілятор буде сприймати типи `Int` та `Integer` як один і той же тип.

3.13. Скалярні типи

3.13.1. Цілі типи

У Delphi є 9 цілих типів, перелік і характеристика яких наведені в таблиці 3.2.

Основними цілими типами є типи `Integer` і `Cardinal`. По можливості рекомендується використовувати саме їх, оскільки з ними краще працюють центральний процесор та операційна система.

Над даними цілих типів допустимі такі арифметичні операції, що дають цілий результат:

`+` – додавання;

`-` – віднімання;

`*` – множення;

`div` – цілочислове ділення (з відкиданням остачі);

mod – ділення за модулем (одержання остачі від ділення).

При цьому результат має тип `Int64`, якщо хоча б один операнд має цей тип. У всіх інших випадках результат має тип `Integer`.

Таблиця 3.2 – Цілі типи Delphi

Тип	Діапазон	Пам'ять у байтах
`Integer`	–2147483648..2147483647	4
`Cardinal`	0..4294967295	4
`ShortInt`	–128..127	1
`SmallInt`	–32768..32767	2
`LongInt`	–2147483648..2147483647	4
`Byte`	0..255	1
`Word`	0..65535	2
`LongWord`	0..4294967295	4
`Int64`	$-2^{63}..2^{63}-1$	8

Характерним для подання цілочислових значень у пам'яті комп'ютера є те, що наступним (попереднім) стосовно максимально (мінімально) можливого значення даного типу є найменше (найбільше) можливе значення у відповідному типі (іноді говорять, що «плюс і мінус нескінченності замикаються»). Так, у типі `Byte` наступним щодо `255` значенням є число `0`. Це буває джерелом помилок, які виявити досить важко.

Операції відношення, застосовані до цілих операндів, дають результат, що належить до типу `Boolean`, а саме, `True` або `False`. Операціями відношення є:

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

До аргументів цілого типу, крім згаданих раніше функцій `Pred` і `Succ`, можна застосувати ряд стандартних функцій:

◊ що дають цілий результат:

- `Abs(x)` – модуль `x`;
- `Hi(x)` – повертає старший байт аргументу (результат має тип `Byte`); якщо тип аргументу має розмір, більше за 16 біт (наприклад, тип `Integer`), опрацьовуються тільки два молодших байти;

- `High(x)` – повертає найбільше значення, якого може набути `x` (параметром може бути й ім'я типу, у тому числі й `Int64`);
- `Lo(x)` – повертає молодший байт аргументу (результат має тип `Byte`);
- `Low(x)` – повертає найменше значення, якого може набути `x` (параметром може бути й ім'я типу, у тому числі й `Int64`);
- `Random(x)` – випадкове ціле число з інтервалу `0..x-1` (аргумент має тип `Integer`; при `x=0` `Random(x)=0`);
- `Sqr(x)` – квадрат `x`;
- `Swap(x)` – опрацьовує аргумент як значення типу `SmallInt` або `Word` і міняє місцями старший і молодший байти, повертаючи значення типу `Integer` (якщо тип аргументу має розмір, більший за 16 біт, опрацьовуються тільки два молодших байти);

◊ що дають дійсний результат:

- `ArcTan(x)` – арктангенс `x`;
- `Cos(x)` – косинус `x`;
- `Exp(x)` – експонента `x`;
- `Ln(x)` – натуральний логарифм `x`;
- `Sin(x)` – синус `x`;
- `Sqrt(x)` – квадратний корінь з `x`;

◊ що дає символьний результат:

- `Chr(x)` – повертає символ з кодом, що задається значенням аргументу `x`, що має тип `Byte`;

◊ що дає булівський результат:

- `Odd(x)` – `True`, якщо `x` – непарне ціле число, `False` – у противному разі.

Функція `Random` при цілому аргументі повертає рівномірно розподілене випадкове ціле число. Її особливістю є те, що при повторному запуску програми вона повертає ті ж самі значення. Це пояснюється тим, що насправді значення, що генеруються за допомогою функції `Random`, є псевдовипадковими числами, які обчислюються на підставі попереднього значення за деяким алгоритмом. При цьому перше обчислене значення залежить від значення, записаного у визначеній у модулі `System` системній змінній `RandSeed` типу `LongInt`, яка за умовчанням має значення `0`. Щоб уникнути повторюваності генерованих чисел треба перед першим викликом функції `Random` викликати процедуру без параметрів `Randomize`, що записує в змінну `RandSeed` початкове значення, яка формується на підставі показань системного таймера.

До виразів цілого типу застосовні також процедури Inc і Dec, які можуть мати по одному або по два параметри цілого типу. Якщо параметрів два (вони в даному випадку розділяються комою), то значення першого параметра збільшується (для Inc) або зменшується (для Dec) на величину, що дорівнює значенню другого параметра (наприклад, Inc(x,2)). Якщо параметр один, то його значення збільшується (у Inc) або зменшується (у Dec) на 1 (наприклад, Dec(x)).

Взагалі кажучи, арифметичні операції над цілими даними повертають значення типу Integer, що еквівалентний типу LongInt. Операції повертають значення типу Int64, тільки коли вони виконуються над операндом типу Int64 (для цього можна здійснити явне перетворення типу операнда до типу Int64 – див. п. 3.15.3).

Більшість стандартних підпрограм, що обробляють цілочислові дані, усікають значення типу Int64 до 32 біт. Однак підпрограми High, Low, Succ, Pred, Inc, Dec, IntToStr і IntToHex повністю підтримують опрацювання аргументів типу Int64, а підпрограми Round, Trunc, StrToInt64 і StrToInt64Def повертають значення типу Int64.

При записі значень у змінні цілого типу вихід за межі допустимого діапазону не контролюється (наприклад, якщо n – змінна типу Integer, то в другому з наступних двох операторів n:=2000000000;n:=2*n не буде виявлена помилка не тільки під час компіляції, але і при виконанні програми). Тому під час налаштування програми рекомендується включати директиву $R, встановлюючи її в активний стан {$R+} (за умовчанням вона пасивна – {$R-}). Можна також в ICP Delphi виконати команду Project ► Options (Проект ► Опції) і встановити прапорець Range Checking (Перевірка області) на вкладці Compiler (Компілятор) діалогового вікна, яке при цьому відкриється. Далі необхідно повторити компіляцію за допомогою команди Project ► Build All Projects (Проект ► Побудувати Всі Проекти).

Неприємною особливістю дій над цілими даними є й те, які в цьому випадку результат виконання арифметичних операцій не контролюється на переповнення. У зв'язку з цим можлива поява помилок, що важко виявляються. Наприклад, якщо n – змінна типу Integer, то результатом виконання операторів

```
n := 10000000;
n := n * 10000000 div 10000000;
```

є значення 27, і ця помилка не буде виявлена навіть за умови активізації директиви $R. У Delphi для контролю переповнення при виконанні операцій над цілими введена директива $Q, що за умовчанням знаходиться в пасивному стані {$Q-}. При необхідності включення контролю

переповнення ця директива повинна бути в потрібному місці активізована: `{$Q+}`. Можна також в ICP Delphi виконати команду Project ► Options (Проект ► Опції) і встановити прапорець Overflow Checking (Перевірка переповнення) на вкладці Compiler (Компілятор) діалогового вікна, що буде відкрите. При цьому слід повторити компіляцію за допомогою команди Project ► Build All Projects (Проект ► Побудувати Всі Проекти).

Директивою `{$Q+}`, як і директивою `{$R+}`, рекомендується користуватися тільки при налаштуванні програми, оскільки їхнє використання сповільнює виконання програми і спричиняє створення компілятором великого програмного коду.

Для забезпечення зв'язку з ядром Windows у модулі `Windows` мови Delphi визначені додаткові цілі типи, а саме:

- на базі типу `Cardinal` тип `ULONG`;
- на базі типу `Byte` тип `UCHAR`;
- на базі типу `SmallInt` тип `SHORT`;
- на базі типу `Word` тип `LANGID`;
- на базі типу `LongWord` типи `DWORD`, `LCID` і `UINT`.

3.13.2. Дійсні типи

Змінні для збереження даних, які реалізуються дійсними числами, найчастіше визначають із типом `Real`. Однак за необхідності в програмах можна використовувати й інші дійсні типи. Усього в Delphi визначено 7 дійсних типів (див. таблицю 3.3).

Таблиця 3.3 – Дійсні типи Delphi

Тип	Діапазон значень	Кількість значущих цифр	Пам'ять у байтах
`Real`	5.0×10^{-324} .. 1.7×10^{-308}	15–16	8
`Real48`	2.9×10^{-39} .. 1.7×10^{38}	11–12	6
`Single`	1.5×10^{-45} .. 3.4×10^{38}	7–8	4
`Double`	5.0×10^{-324} .. 1.7×10^{-308}	15–16	8
`Extended`	3.6×10^{-4951} .. 1.1×10^{4932}	19–20	10
`Comp`	-2^{63} .. $2^{63}-1$	19–20	8
`Currency`	–922337203685477.5808.. 922337203685477.5807	19–20	8

Тип `Real48` введений замість 6-байтового типу `Real` мов Pascal і Delphi версій 1..3. Вживання його в програмах неефективне.

Тип `Real`, будучи 8-байтовим, еквівалентний типу `Double`. Це найбільш вживаний з дійсних типів.

Тип Comp містить тільки цілі значення, які подаються в обчисленнях як дійсні. Замість нього краще застосовувати тип Int64.

Тип Currency використовується в грошово-кредитних обчисленнях з метою мінімізації помилок округлення. Фактично, як і тип Comp, він містить тільки цілі значення. В обчисленнях при змішуванні з іншими дійсними типами значення, що подані в типі Currency, діляться на 10000 і подаються в обчисленнях як дійсні. Довідкова система Delphi замість типу Currency рекомендує використовувати тип Int64.

Над дійсними операндами виконуються такі операції, що дають дійсний результат:

+ – додавання;	- – віднімання;
* – множення;	/ – ділення,

а також операції відношення (порівняння).

Якщо застосувати операцію ділення до двох цілих операндів, то результат буде дійсним і матиме тип Extended.

До дійсних аргументів застосовні раніше розглянуті функції Abs, Sqr, Sqrt, Sin, Cos, ArcTan, Ln, Exp, а також функції Int (повертає у форматі дійсного значення цілу частину аргументу) і Frac (повертає дробову частину аргументу). Дві функції дають цілий результат типу Int64:

Trunc(x) – відсікання дробової частини;

Round(x) – округлення до найближчого цілого.

Функція без аргументу Random повертає рівномірно розподілене випадкове число r з інтервалу $0 < r \le 1$. Функція без аргументу Pi повертає число Піфагора 3.1415...

В доповнення до розглянутих стандартних математичних функцій, які визначені в модулі System, у Delphi є ще безліч математичних функцій, визначених у модулі Match.

3.13.3. Булівські типи

Тип Boolean визначає ті дані, які можуть набувати логічних значень False (неправда) або True (істина). До булівських даних застосовні наступні логічні операції:

not – заперечення;

and – кон'юнкція («і»);

or – диз'юнкція («або»);

xor – виключаюче «або».

Якщо a і b – змінні типу Boolean, то результат виконання логічних операцій над ними визначається таблицею 3.4.

Таблиця 3.4 – Логічні операції

a	b	**not** a	a **and** b	a **or** b	a **xor** b
False	False	True	False	False	False
False	True	True	False	True	True
True	False	False	False	True	True
True	True	False	True	True	False

Крім типу Boolean, що займає 1 байт, у Delphi для сумісності з Windows введено ще чотири булівських типи:

- ByteBool (займає 1 байт);
- Bool (займає 2 байти);
- WordBool (займає 2 байти);
- LongBool (займає 4 байти).

Для всіх булівських типів у Delphi виконується умова False<True, причому значенню False відповідає число 0, записане у виділену під булівську змінну ділянку пам'яті, а значенню True – будь-яке ненульове значення. Відмінність додаткових булівських типів від типу Boolean полягає в такому: при присвоюванні їм значення, що задається константою True, у виділену ділянку пам'яті записується число –1, а у тип Boolean – число 1. При цьому, хоча числове значення, що відповідає значенню True, у різних булівських типів відрізняється, співвідношення False<True виконується завжди.

До даних булівського типу можна застосовувати всі операції відношення. Крім того, до них застосовні функції Ord, Succ, Pred, High, Low і процедури Inc і Dec. Однак варто пам'ятати особливості подання значення True у додаткових булівських типах і користуватися цими типами з обережністю.

Складні логічні вирази можуть не опрацьовуватися до кінця, якщо продовження обчислень не змінить результат. За умовчанням опрацювання складних булівських виразів проводиться саме так. Якщо булівський вираз в обов'язковому порядку потрібно опрацювати до кінця, то це забезпечується включенням директиви компілятору {$B+}. При задаванні {$B–} булівський вираз може до кінця не опрацьовуватися. Цієї ж установки можна забезпечити звертанням до ICP Delphi, для чого необхідно виконати команду Project ► Options (Проект ► Опції) і встановити або скинути прапорець Complete Boolean eval (Повне обчислення) на вкладці (сторінці) Compiler (Компілятор) діалогового вікна, яке при цьому відкриється.

3.13.4. Символьні типи

Фундаментальними символьними типами в Delphi є типи `AnsiChar` і `WideChar`. Тип `AnsiChar` займає 1 байт і служить для подання 256 різних символів, що відрізняються кодом ANSI, який, власне, і зберігається у відведеному байті. Свою назву код отримав за назвою Національного інституту стандартизації США (American National Standards Institute), що запропонував його. Відповідно до назви коду було дано і назву типу.

Тип `WideChar` використовує більш як 1 байт для подання кожного з символів (зараз 2 байти). Він введений для подання символів у кодуванні Unicode, що дозволяє закодувати символи всіх мов світу і має 65536 можливих значень. Перші 256 символів цього коду відповідають ANSI-символам.

У Delphi введено також родовий символьний тип `Char`, який еквівалентний типу `AnsiChar` і визначає впорядковану сукупність основних символів мови відповідно до кодування ANSI.

Перша половина кодової таблиці (символи з кодами від `0` до `127`) є стандартною, а друга половина залежить від типу шрифту; у ній, зокрема, розміщають символи національних алфавітів.

У таблиці 3.5 наведена основна частина кодової таблиці відповідно до кодування ANSI.

У стандартних для Windows шрифтах Arial, Courier New, Times New Roman для подання символів кирилиці (за винятком російських літер «Ё» і «ё») використовуються коди від 192 до 255 (великі літери мають коди від 192 до 223, а малі – від 224 до 255). Російські літери «Е» і «е» мають відповідно коди 168 і 184. Українські літери «Ґ», «ґ», «Є». «є», «Й», «й», «Ї» і «ї» мають коди 165, 180, 170, 186, 178, 179, 175 і 191. Прийнята в нас друга половина кодової таблиці відповідно до кодування ANSI наведена в таблиці 3.6.

Символи впорядковані відповідно до їх коду (десяткові значення кодів від 0 до 255). Тому до даних символьного типу можна застосовувати всі операції відношення. Значення символьної змінної – один символ. Як це вже відзначалося раніше, у програмі значення символьного типу можна подати записом, що складається з символу `#` і цілого коду в десятковій або шістнадцятковій системі числення (`#214`, `#17`, `#$FF`, `#$1D`).

Якщо символ має екранне подання, то його можна вказати в явному вигляді, «взявши» в апострофи (наприклад, `'a'`, `'A'`, `'6'`, `'-'`). Символи з десятковими кодами від 0 до 31 є ***службовими*** (***символами керування***). Ці символи інакше називають ***узагальненими пробільними символами***, оскільки при їх вживанні у програмі вони вважаються пробілами.

Таблиця 3.5 – Основна частина кодової таблиці для кодування ANSI

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

Таблиця 3.6 – Додаткова частина кодової таблиці для кодування ANSI

Код	Символ	Код	Символ	Код	Символ	Код	Символ
128	Ђ	160		192	А	224	а
129	Ѓ	161	Ў	193	Б	225	б
130	‚	162	ў	194	В	226	в
131	ѓ	163	J	915	Г	227	г
132	„	164	¤	196	Д	228	д
133	…	165	Ґ	197	Е	229	е
134	†	166	¦	198	Ж	230	ж
135	‡	167	§	199	З	231	з
136	€	168	Ё	200	И	232	и
137	‰	169	©	201	Й	233	й
138	Љ	170	Є	202	К	234	к
139	‹	171	«	203	Л	235	л
140	Њ	172	¬	204	М	236	м
141	Ќ	173	-	205	Н	237	н
142	Ћ	174	®	206	О	238	о
143	Џ	175	Ї	207	П	239	п
144	ђ	176	°	208	Р	240	р
145	‘	177	±	209	С	241	с
146	’	178	І	210	Т	242	т
147	“	179	і	211	У	243	у
148	”	180	ґ	212	Ф	244	ф
149	•	181	µ	213	Х	245	х
150	–	182	¶	214	Ц	246	ц
151	—	183	·	215	Ч	247	ч
152	□	184	ё	216	Ш	248	ш
153	™	185	№	217	Щ	249	щ
154	љ	186	є	218	Ъ	250	ъ
155	›	187	»	219	Ы	251	ы
156	њ	188	j	220	Ь	252	ь
157	ќ	189	S	221	Э	253	э
158	ћ	190	s	222	Ю	254	ю
159	џ	191	ї	223	Я	255	я

Функція `Ord(x)`, де `x` – значення символьного типу, повертає код даного символу. Зворотною їй є функція `Chr(x)`, що дає символьне подання цілого невід'ємного числа `x`.

До даних типу `Char` застосовні функції `Pred`, `Succ`, `High`, `Low`, а також процедури `Inc` і `Dec`.

Можливе також використання функції

```
UpCase(x)
```

з одним параметром, яка повертає велику літеру, якщо `x` – мала **латинська** літера, і сам символ `x` у противному разі.

3.13.5. Перелічений тип

У Delphi існує тип даних, значеннями якого є імена, перелічені при оголошенні типу. Формат оголошення подібного роду типу такий:

```
type
  ім'я_типу = (ім'я1, ім'я2, ...);
```

Можливими значеннями оголошеної в програмі змінної переліченого типу будуть імена, зазначені в дужках. Наприклад:

```
type
  Months = (January, February, March, April, Mai,
            June, July, August, September, October,
            November, December);
var
  Month: Months;                                    //Змінна

const
  Month1: Months = Mai;          //Типізована константа
```

Значення переліченого типу впорядковані за зростанням відповідно до порядку їх переліку. Так, для типу `Months` справедливі нерівності `January<February<...<December`. До змінних переліченого типу можна застосовувати функції `Ord`, `Pred`, `Succ` і процедури `Inc` і `Dec`. Порядкові значення переліченого типу відраховуються від 0: застосування функції `Ord` до першого зі значень дає в результаті 0 (наприклад, `Ord(January)=0`, `Ord(April)=3`). Усього можна задати до 256 різних значень для одного переліченого типу.

Нехай необхідно розв'язати таку задачу:

```
//Приклад 3.1
//Чи правда, що середньорічний доход був меншим за середній
//доход за перше півріччя?
```

Скористаємося для її розв'язанням кінцевим варіантом форми з прикладу 2.2, внісши в неї деякі зміни, а саме, властивості Caption форми присвоїмо значення Приклад на перелічений тип, властивості Caption кнопки Button1 – значення Ввести, а властивості Caption мітки – значення Уведіть доход за січень.

Включимо в розділ опису типів секції **interface** модуля описаний вище тип Months, а далі включимо наступні рядки, у яких описується типізована константа – рядковий масив (див. п. 5.2.1) :

```
const
      Ukr_Months: array [Months] of string=('січень', 'лютий',
        'березень', 'квітень', 'травень', 'червень', 'липень',
        'серпень','вересень','жовтень','листопад','грудень');
```

Крім того, у розділ опису змінних секції **implementation** включимо такі оголошення:

```
     Month: Months;
     SumEmol1, SumEmol2: Real;   //Сумарний доход по півріччях
     Emolument: Real;                          //Доход за місяць
```

Для опрацьовувача події OnCreate форми напишемо такий код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DecimalSeparator := '.';
  SumEmol1 := 0; SumEmol2 := 0;
  Month := January
end;
```

Цей код забезпечує присвоювання початкових значень змінним (у тому числі й змінній Month, що має перелічений тип).

Код опрацьовувача події OnClick компонента Button1 може бути наступним:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Emolument := StrToFloat(edInput1.Text);
  if (January <= Month) and (Month <= June)       //Порівняння
    then SumEmol1 := SumEmol1 + Emolument
    else SumEmol2 := SumEmol2 + Emolument;
  if Month = December then begin
    if (SumEmol1 + SumEmol2) / 12 < SumEmol1 / 6
      then mmOutput1.Lines.Add('Так')
      else mmOutput1.Lines.Add('Ні');
    Button1.Visible := False;
    edInput1.Visible := False;
    lbOutput1.Visible := False;
  end
```

```
  else begin
    Inc(Month);                                    //Змінювання
    lbOutput1.Caption:='Уведіть доход за '+Ukr_Months[Month];
    edInput1.SetFocus;
  end;
end;
```

Даний приклад ілюструє методику опису, як самого переліченого типу, так і змінних цього типу, а також можливість використання таких змінних.

У наведеному прикладі виконане присвоювання значення `False` властивості `Visible` компонентів `Button1`, `edInput1` і `lbOutput1` по закінченні обчислень. Результатом є те, що перераховані компоненти стають невидимими, а оскільки активними є тільки видимі компоненти, після виведення результату обчислень повторний запуск програми можливий лише після закриття форми.

Перелічений тип є порядковим. За умовчанням першому з імен, перелічених в описі переліченого типу, ставиться у відповідність значення 0. Порядкові значення інших імен утворюються збільшенням на 1 значення порядкового номера попереднього імені в списку переліку імен.

Порядкові значення імен можна задавати явно, порушуючи початкову нумерацію. Для цього в списку перелічуваних імен потрібно після знака = навести значення імені. При цьому одне й те саме значення може бути в декількох імен. У дійсності перелічений тип є відрізком з мінімальним і максимальним значеннями, що відповідають мінімальному і максимальному значенням констант у списку переліку (нехай навіть не заданих явно). Деякі з цих значень досяжні через імена, а деякі – за допомогою підпрограм `Pred`, `Succ`, `Inc`, `Dec`.

Нехай є такий опис типу:

```
type
  SomeEnum = (e1=1, e2, e3 = 1, e4=0, e5, e6=6, e7);
```

У даному випадку діапазон порядкових значень – це цілі числа від `0` до `7`, що відповідають іменам `e4` і `e7`. Іменам `e1`, `e3`, `e5` відповідає одне й те саме порядкове значення `1`, а імені `e2` – порядкове значення `2`.

3.13.6. Відрізки

Нехай ми маємо деякий тип даних, утворений сукупністю дискретних величин (наприклад, `Integer`). На базі цього типу можна побудувати новий тип, що називається ***відрізком*** (***діапазоном***). Наприклад, дані типу `Integer` набувають значення від `−2147483648` до `2147483647`. Якщо

нас цікавить вужчий діапазон (наприклад, від 1 по 300), то ми можемо скористатися відрізком.

При описі відрізка зазначають його початок і кінець, розділені двома крапками, між якими не повинні стояти пробіли:

```
var
  ім'я_відрізка: початок_діапазону..кінець_діапазону;
```

Тип-діапазон можна визначити, взявши за базовий будь-який порядковий тип. Наприклад:

```
var
  n      : 1..300;
  letter : 'a'..'z';
```

Домогтися економії пам'яті за рахунок використання відрізків не вдасться, бо елементи даних типу «відрізок» займають обсяг пам'яті, необхідний для розміщення кожного зі значень даного типу (наприклад, під змінну, що має тип 200..300, буде виділено 2 байти пам'яті, оскільки її базовим типом буде тип SmallInt). Базовим типом кожного з відрізків є «найвужчий» тип, що включає всі можливі значення відрізка.

Межі діапазону можна задавати виразами над константами, але в цьому випадку можливі синтаксичні труднощі. У будь-якому визначенні типу, в якому перший значущий символ після = (або після двокрапки у випадку опису змінної) – ліва кругла дужка, компілятор вважає, що має місце визначення переліченого типу. Отже, код

```
const
  a = 100; b = 200;
var
  d: (a + b) div 2 + 1..b;
```

буде помилковим, оскільки компілятор сприйме ліву дужку як дужку, що обмежує визначення переліченого типу. Ця помилка легко виправляється, наприклад, у такий спосіб:

```
var
  d: 1 + (a + b) div 2..b;
```

Для того щоб компілятор виявляв вихід за діапазон значення змінної відрізкового типу, потрібно включати директиву компілятору $R.

3.13.7. Тип дата-час

У Delphi можливе визначення змінних, призначених для одночасного зберігання дати й часу. Такі змінні описуються як змінні, що мають тип

TDataTime. Дані типу TDataTime займають у пам'яті 8 байт і являють собою дійсне число з фіксованою дробовою частиною (аналогічні даним типу Currency). Ціла частина цього числа служить для подання дати, а дробова – часу. Точкою відліку є 0 годин 0 хвилин 0 секунд 30 грудня 1899 року. Дата визначається як кількість діб, що пройшли від точки відліку, а час – як частина доби, що минула від 0 годин.

Від'ємним значенням, записаним у змінні типу TDataTime, відповідають дата й час, що передують точці відліку. Даті 00.00.0000 від Різдва Христова відповідає значення −693594. Менші значення функціями перетворення дати й часу до рядкового виду перетворяться до дати 00.00.0000.

Для одержання рядкового подання дати й часу в Delphi визначені такі стандартні функції з одним параметром типу TDataTime:

- DateTimeToStr(дата_час) – перетворює дату й час;
- DateToStr(дата_час) – перетворює дату;
- TimeToStr(дата_час) – перетворює час.

Визначена також функція

FormatDateTime(формат, дата_час),

що має два параметри (перший – рядок символів, а другий – типу TDataTime). Ця функція перетворює дату й час у рядок символів відповідно до формату, який заданий першим параметром.

Крім того, визначені три функції без параметрів, що зчитують поточний системний час і повертають значення у форматі TDataTime. Це такі функції:

- Date – повертає поточну дату;
- Now – повертає поточну дату й час;
- Time – повертає поточний час.

Є також функції зворотного перетворення рядка до типу TDateTime.

Крім того, у модулі DateUtils визначені функції IncMilliSecond, IncSecond, IncMinute, IncHour, IncDay, IncWeek, IncMonth, IncYear, що змінюють дату й час, задані у вигляді параметра, на задану кількість мілісекунд, секунд, хвилин, годин, доби, тижнів, місяців, років відповідно.

Тип TDataTime є сумісним з форматом дійсних чисел. Тому дані цього типу можуть брати участь в арифметичних виразах без додаткового перетворення (однак варто пам'ятати про їх фізичний зміст).

```
//Приклад 3.2
//Зчитати і вивести поточну дату й час.
//Збільшити дату на 5 діб і вивести результат змінення.
```

Скористаємося для розв'язання задачі кінцевим варіантом форми з прикладу 2.2, видаливши з неї мітку й однорядковий редактор, а також присвоївши властивості Caption форми значення Приклад на тип дата-час, а властивості Caption компонента Button1 – значення Вивести.

Тоді задачу розв'язує такий опрацьовувач події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  dt: TDateTime;                //Описуємо змінну типу TDateTime
begin
  dt := Now;                       //Зчитуємо поточну дату й час
                                //Виводимо результат як дату-час
  mmOutput1.Lines.Add(DateTimeToStr(dt));
  dt := dt + 5;          //Збільшуємо дату на 5 діб (без IncDay)
                                   //Демонструємо результат зміни
  mmOutput1.Lines.Add(DateTimeToStr(dt));
end;
```

Відзначимо, що в операторі `dt := dt + 5` значення `dt`, як дійсне число, збільшується на `5`. А оскільки при цьому змінюється тільки ціла частина значення `dt`, відбувається зміна дати на 5 діб без зміни часу, що зберігається в дробовій частині `dt`.

3.14. Оголошення констант

3.14.1. Іменування констант

Якщо в програмі використовується числовий матеріал, що задається константами, то у випадку її налаштування на нові числові значення доведеться робити корекцію по всьому тексту. Більш розумно дати цим константам імена, що здійснюється шляхом визначення констант.

Визначення константи дозволяє надати деяке ім'я певним значенням, і це значення може бути використане в будь-якому місці програми і не може бути змінене (насамперед, випадково). Таке визначення виконується в спеціальному розділі, що має вигляд

```
const
  ім'я_константи1 = значення1;
  ім'я_константи2 = значення2;
```

Константи можуть бути не тільки числовими, але й символьними, булівськими, рядковими і т. д. Наприклад:

```
const                                    //Оголошення констант:
  e = 2.72; Zero = 0;     //числових (дійсної і цілої),
  No = False; Yes = True;                          //булевих,
  Divider = '|';                                //символьної,
  Language = 'Delphi 7';                           //рядкової
```

Тип константи при цьому визначається автоматично відповідно до типу значення.

Якщо потрібно модифікувати програму зміною яких-небудь значень, достатньо внести відповідні зміни в розділ визначення констант. Приклад 3.3 демонструє перевагу програм, що використовують константи.

Нехай необхідно розв'язати таку задачу: **знайти суму перших 30 натуральних чисел.**

```
//Приклад 3.3
//Сума перших 30 натуральних чисел.
```

Розмістимо на формі мітку й дві кнопки (Button1 та BitBtn1) і змінимо їхні властивості в такий спосіб:

- Форма:
  Caption — Сума перших натуральних чисел
  Position — poScreenCenter
- Мітка:
  Caption — очистити
  Name — lbOutput1
- Кнопка Button1:
  Caption — Обчислити

Опрацьовувач події OnClick компонента Button1 може мати, наприклад, такий вигляд:

```
procedure Tfm1.Button1Click(Sender: TObject);
var
  i, Sum: Integer;
begin
  Sum := 0;
  for i := 1 to 30 do
    Sum := Sum + i;
  lb1Output.Caption := IntToStr(Sum) +
    ' – це сума перших 30 натуральних чисел';
end;
```

Якщо обчислюється сума іншої кількості чисел, потрібно буде в двох місцях внести зміни. Скористаємося визначенням констант:

```
procedure Tfm1.Button1Click(Sender: TObject);
const                                              //Константи:
```

```
  NumberToSum = 30;                              //...числова і...
  Message1 = ' - це сума перших ';              //...дві рядкові
  Message2 = ' натуральних чисел ';
var
  i, Sum: Integer;
begin
  Sum := 0;
  for i := 1 to NumberToSum do
    Sum := Sum + i;
  lb1Output.Caption := IntToStr(Sum) + Message1 +
                       IntToStr(NumberToSum) + Message2;
end;
```

Тепер, якщо це буде потрібно, достатньо змінити константу `NumberToSum` на початку процедури `Tfm1.Button1Click`. Визначення констант можна виконати в розділі **`interface`** модуля, і тоді вони будуть відомі у всьому тексті програми, а не тільки в опрацьовувачі події OnClick компонента Button1.

3.14.2. Типізовані константи

Додатково до звичайних констант є можливість застосувати типізовані константи, що є чимсь проміжним між константами і змінними:

1. Типізовані константи описуються в розділі опису констант.
2. Як і звичайні іменовані константи, типізовані константи при описі одержують значення.
3. Аналогічно змінним типізовані константи мають тип, що задається описом.
4. Як і змінні, типізовані константи можуть змінювати своє значення.

Опис типізованої константи має такий вигляд:

```
const
  ім'я: тип = значення;
```

Значення типізованої константи може задаватися виразом з використанням нетипізованих констант, константою посилального типу, ім'ям підпрограми, зображенням масиву, зображенням запису, зображенням множини, зображенням об'єкта або класу.

Нижче наводиться приклад оголошення числових типізованих констант:

```
const
   zero: Integer = 0;
   a   : Real = 1.34e-7;
```

При описі типізованих констант повинна бути забезпечена сумісність за присвоюванням.

Для того щоб значення типізованої константи могло змінюватися по ходу виконання програми, необхідно виконати налаштування в ICP Delphi: виконати команду Project ► Options (Проект ► Опції) і встановити на вкладці Compiler (Компілятор) діалогового вікна, що при цьому відкриється, прапорець Assignable Typed Constants (Обумовлені константи). Цього ж можна домогтися за допомогою директиви компілятору `$J`.

Принцип опису типізованих констант для більш складних типів буде розглянутий у розділах, присвячених відповідним типам даних.

3.15. Еквівалентність і сумісність типів

3.15.1. Еквівалентність типів

У Delphi строго визначено, які типи описують ідентичні множини значень (є еквівалентними). У мові прийнято принцип іменної еквівалентності типів, який встановлює, що два типи `T1` і `T2` еквівалентні, якщо виконується одна з умов:

1) `T1` і `T2` – одне й те саме ім'я типу;

2) тип `T2` описаний з використанням типу `T1` рівністю вигляду

```
type
  T2 = T1;
```

або послідовністю таких рівностей:

```
type
  T1 = Integer;
  T3 = T1;
  T2 = T3;
```

Наприклад, не є еквівалентними типи

```
type
  Tp1 = array[1..10] of Real;
  Tp2 = array[1..10] of Real;
```

незважаючи на їхню абсолютну ідентичність. У той же час при описі

```
type
  Tp3 = Tp1;
  Tp4 = Tp3;
```

пари типів `Tp1` і `Tp3`, `Tp4` і `Tp3`, `Tp4` і `Tp1` еквівалентні.

Якщо змінні описані спільно, то вони мають еквівалентні типи. Так, в описах

```
var
  v1, v2: array[1..10] of Real;
  v3, v4: Tp1;
```

у пар змінних v1 і v2, v3 і v4 типи еквівалентні, а в парах v1 і v3, v1 і v4, v2 і v3, v2 і v4 типи не є еквівалентними.

При оголошенні користувальницького типу зі службовим словом **type** після знака = навіть повністю ідентичні типи не будуть еквівалентними. Так, не будуть еквівалентними тип Integer і тип MyInt, описаний за допомогою оголошення

```
type
  MyInt = type Integer;
```

3.15.2. Сумісність типів

Однією з вимог Delphi є така: у виразах (у тому числі при порівнянні) повинні використовуватися операнди з сумісними типами. Типи сумісні, якщо виконується хоча б одна з умов (не розглянуті до даного моменту типи будуть описані далі):

- обидва типи еквівалентні;
- обидва типи цілі;
- обидва типи дійсні або один з них – тип TDateTime;
- один з типів є відрізком, причому базовим для нього є інший тип:

```
type
  Tp1 = LongInt;
  Tp2 = -10..10;
```

- обидва типи є відрізками одного і того ж базового типу:

```
type
  Tp1 = 'A'..'Z';
  Tp2 = 'A'..'F';
```

- один тип рядковий, а другий – або рядковий, або символьний, або упакований символьний масив;
- обидва типи – упаковані символьні масиви з однаковим числом елементів;
- обидва типи множинні з сумісними базовими типами:

```
type
   Tp1 = set of Byte;
   Tp2 = set of 1..100;
```

- один тип є посилальним, а другий – або посилальний, або безтиповий вказівник;
- обидва типи – це процедурні типи з одним і тим самим числом параметрів, причому типи параметрів повинні бути еквівалентними (відповідно до порядку, в якому вони зустрічаються), а для функціональних типів, крім того, повинні бути еквівалентними типи результатів.

Якщо у виразі типи сумісні, але різні, то тип результату визначається більш загальним з типів операндів.

Крім понять еквівалентності і сумісності типів, існує поняття сумісності за присвоюванням.

Оператор присвоювання коректний, якщо тип змінної в його лівій частині (`T1`) є сумісним за присвоюванням з типом виразу в правій частині (`T2`). Для цього повинна виконуватися хоча б одна з умов:

- обидва типи еквівалентні, але жоден з них не є файловим типом або складним типом, що використовує файловий тип;
- обидва типи – сумісні порядкові типи, і поточне значення типу `T2` потрапляє в діапазон можливих значень типу `T1`;
- обидва типи дійсні, і поточне значення типу `T2` потрапляє в діапазон можливих значень типу `T1`;
- тип лівої частини дійсний, а тип правої частини – цілий;
- тип `T1` – рядковий тип, а `T2` – або будь-який рядковий тип, або символьний тип, або упакований символьний масив;
- обидва типи – упаковані символьні масиви;
- обидва типи – сумісні множинні типи, причому множина з правої частини цілком входить у множину типу `T1`;
- обидва типи – сумісні посилальні типи;
- тип лівої частини – процедурний тип, а права частина – ім’я процедури або функції з тим же числом параметрів, що й у типу лівої частини; типи відповідних параметрів (а також типи результату для функції) повинні бути еквівалентними;
- обидва типи – об’єктові типи або класи, причому тип `T2` є нащадком типу `T1`;
- обидва типи – посилальні типи на сумісні об’єктові типи або класи.

Використання несумісних за присвоюванням типів спричиняє появу помилок.

3.15.3. Явне перетворення типів

У деяких випадках у Delphi відбувається автоматичний перехід від одного типу даних до іншого (від цілого до дійсного, від символьного до рядкового). Існує також ряд функцій, що виконують перетворення типів (`Ord`, `Chr`, `Trunc`, `Round`). Поряд з цим, в Delphi можливе явне перетворення типів (ретипізація даних). Для того щоб здійснити явне перетворення типу, необхідно використати ім'я типу аналогічно тому, як використовується ім'я функції. Як параметр у цьому випадку зазнається ім'я перетворюваного елементу даних (можливе також використання виразу). Наприклад, якщо є описи

```
var
  n: Byte;
  c: Char;
  b: Boolean;
```

то в результаті виконання операторів

```
c := #0;
n := Byte(c);
b := Boolean(c);
```

у змінну `n` буде записане значення `0`, а у змінну `b` – значення `False`.

Перетворення типів здійснюється не завжди. Так, воно забороняється компілятором, якщо двом типам відповідають різні об'єми пам'яті.

При явному перетворенні типів слід ураховувати також такі обмеження:

- порядковий тип або вказівник можуть перетворюватися до порядкового типу або вказівника;
- символ, рядок, символьний масив або дані типу `PChar` можуть бути перетворені до рядка;
- порядковий, дійсний, рядковий або варіантний тип можуть бути перетворені до варіанта;
- варіантний тип може бути перетворений до порядкового, дійсного, рядкового або варіантного типу;
- посилальна змінна може бути перетворена до будь-якого типу того ж розміру.

Розглянемо найпростіший приклад використання явного перетворення типу:

```
//Приклад 3.4
//Обчислити значення функції sign(x), яка визначається так:
//1 при x>0; 0 при x=0; -1 при x<0.
```

Створимо проект програми, аналогічний проекту в прикладі 2.3, виконавши такі зміни властивостей компонентів: властивості Caption форми присвоїмо значення Приклад на приведення типів, а властивості Caption мітки – значення Уведіть дійсне число.

Крім того, у розділі **implementation** оголосимо дві змінні:

```
x: Real;
sign: Integer;
```

В опрацьовувач події OnCreate для форми включимо тільки один оператор, про призначення якого говорилося в зауваженні до прикладу 2.2:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DecimalSeparator := '.';
end;
```

Створимо також такий опрацьовувач події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  x := StrToFloat(edInput1.Text);
  sign := Byte(x > 0) - Byte(x < 0);
  mmOutput1.Lines.Add(IntToStr(sign));
  edInput1.SetFocus;
end;
```

У ньому при `x>0` вираз `x>0` має значення `True`, а вираз `x<0` – `False`, а значить, `Byte(x>0)=1`, `Byte(x<0)=0` і `sign=1`; при `x<0` вираз `x>0` має значення `False`, а вираз `x<0` – `True`, а значить, `Byte(x>0)=0`, `Byte(x<0)=1` і `sign=-1`; при `x=0` вирази `x>0` і `x<0` мають значення `False`, а значить, `Byte(x>0)=0`, `Byte(x<0)=0` і `sign=0`.

В операторі присвоювання ім’я типу може зустрітися й у лівій частині. Однак типи лівої та правої частин оператора присвоювання і їх розміри у цьому випадку контролюється більш жорстко.

Наприклад, якщо змінні `n` і `x` мають відповідно типи `Byte` і `Real`, то є допустимим такий оператор:

```
Boolean(n) := (x > 0);
```

Тип даних (у цьому випадку `Boolean`), що використовується тут у лівій частині оператора присвоювання, повинен відповідати типу значення, що стоїть у правій частині. У змінну в лівій частині (у даному випадку `n`)

записується значення не у форматі, що відповідає типу цієї змінної (`Byte`), а у форматі того типу, до якого йде перетворення (`Boolean`).

3.16. Бітова арифметика

У Delphi, крім звичайних дій над цілими й дійсними даними, введено додаткові операції над цілими типами (`Byte`, `ShortInt`, `Word`, `Integer`, `LongInt`, `Int64` і т. д.) – бітова (порозрядна) арифметика.

3.16.1. Логічні операції над бітами

Якщо є два цілих операнди `A1` і `A2`, то над їх відповідними бітами можна виконувати логічні операції:

`not` – порозрядне заперечення;
`and` – логічне множення;
`or` – логічне додавання;
`xor` – виключаюче «або».

Результат виконання кожної з цих операцій залежить від значень відповідних бітів операндів і визначається таблицею 3.7.

Таблиця 3.7 – Виконання логічних операцій над бітами

A1	A2	**not** A1	A1 **and** A2	A1 **or** A2	A1 **xor** A2
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Приклади:

```
                6 and 4 = 4                    6 or 4 = 4
and   00000110 (6)                    or   00000110 (6)
      00000100 (4)                         00000100 (4)
      ------------                         ------------
      00000100 (4)                         00000110 (6)
```

У наведених прикладах передбачається, що опрацьовувані значення мають тип `Byte`.

Операція **`and`** практично завжди використовується тільки для досягнення однієї з двох цілей: перевірити наявність установлених в 1 бітів або провести обнуління деяких з них.

Подібна перевірка потрібна, коли число – це набір ознак з двома можливими значеннями (набір прапорів). Так, багато системних комірок пам’яті містять відомості про конфігурацію комп’ютера або його стан. При

цьому установка біта з конкретним номером в 1 трактується як вмикання якого-небудь режиму, а в 0 – вимикання.

Нехай змінна A має тип Byte і є байтом з вісьма прапорами, і потрібно перевірити стан біта з номером 5 (справа наліво від 0 до 7). Одиниця в біті 5 – це п'ятий степінь числа 2, тобто 32. Тому, якщо в п'ятому біті змінної A стоїть одиниця, то виконується умова (A **and** 32)=32, що і перевіряється в операторі **if**. Якщо необхідно перевіряти стан декількох одночасно встановлених в 1 бітів, то також слід обчислити відповідне число як суму степенів числа 2, де показники степені дорівнюють номерам бітів, встановлених в 1. Наприклад, для бітів 5, 2 й 0 маємо 32 + 4 + 1 = 37. Якщо A має серед інші одиниці в бітах 5, 2 й 0, то виконується умова (A **and** 37)=37.

Приклади:

Перевірка установки біта 5

```
  10110010 (178)
and
  00100000 ( 32)
  --------------
  00100000 ( 32)   – Так
```

Перевірка установки бітів 5, 2, 0

```
  10110110 (182)
and
  00100101 ( 37)
  --------------
  00100100 ( 36)   – Не всі
```

Нехай потрібно записати нуль у який-небудь біт змінної A типу Byte (наприклад, у біт 3). Визначимо спочатку число, що містить одиниці у всіх бітах, крім третього. Максимальне число, яке можна записати в тип Byte – це 255. Щоб записати нуль у третій біт віднімемо з цього числа третій степінь числа 2 (255 – 8). Якщо це число логічно помножити на A, то його одиниці ніяк не позначаться на стані змінної A, а 0 у третьому біті незалежно від значення третього біта змінної A дасть в результати 0. Отже, маємо: A := A **and** (255-8).

Аналогічно можна записати нулі у декілька бітів.

Приклади:

Обнуління 3-го біта

```
  10111010 (188)
and
  11110111 (255 – 8)
  ------------------
  10110010 (172)
```

Обнуління декількох бітів (5, 2, 0)

```
  10110011
and
  11011010
  --------
  10010010
```

Операція **or** застосовується при установці в 1 окремих бітів двійкового подання цілих чисел. Так, щоб установити біт 4 змінної A в 1 без змінювання інших бітів, достатньо записати A := A **or** 16, де 16 – четвертий степінь числа 2. Аналогічно встановлюються в 1 декілька бітів.

Операція **xor** вживається для зміни значення біта (або декількох бітів) на протилежне (1 на 0 або 0 на 1). Наприклад, щоб переключити на протилежний стан біта 3 змінної A, достатньо записати A := A **xor** 8, де

8 – третій степінь числа 2. Відзначимо, що дворазове застосування операції **xor** відновлює старе значення змінної, тобто (A **xor** B) **xor** B) = A.

Приклади:

Установка декількох бітів (0 і 4)

```
  101|1|101|0|  (188)
or
  000|1|000|1|  ( 17)
  ---------------------
  101|1|101|1|  (189)
```

Інверсія бітів 0 і 4

```
  101|1|101|0|  (188)
xor
  000|1|000|1|  ( 17)
  ---------------------
  101|0|001|1|  (173)
```

3.16.2. Операції циклічного зсуву

У Delphi визначено ще дві операції над даними цілого типу, що мають той же самий пріоритетний рівень, що й операції **and**, *, /, **div** та **mod**. Це операції **shl** і **shr**, які зсовують послідовність бітів у двійковому поданні числа на задану кількість позицій уліво і вправо відповідно. При цьому біти, які виходять за розрядну сітку, втрачаються. При виконанні операції **shl** біти, що звільнилися праворуч, заповнюються нулями. При виконанні операції **shr** біти, що звільнилися ліворуч, заповнюються одиницями при зсуві від'ємних значень і нулями у випадку додатних значень.

Приклади:

A **shl** 2 – циклічний зсув вліво на два біти;

A **shr** 3 – циклічний зсув вправо на три біти.

За допомогою операції **shl** можлива заміна операції множення цілих чисел на степінь двійки.

Приклади:

(J **shl** 1) – значення J помножується на 2;

(J **shl** 2) – значення J помножується на 4;

(J **shl** 3) – значення J помножується на 8.

Вирішимо наступну задачу:

```
//Приклад 3.5
//Дане натуральне число n. Обчислити n-й степінь числа 2.
```

Скористаємося для розв'язання формою з прикладу 3.1, присвоївши властивостям Caption форми і мітки відповідно значення Приклад на використання shl й Уведіть цілий показник степеня 0<n<=30).

Крім того, у розділі **implementation** оголосимо дві змінні: n типу Byte і Res типу Integer. Не створюючи опрацьовувач події OnCreate для форми, створимо такий опрацьовувач події OnClick для компонента Button1, який, власне, розв'язує задачу:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Res := 1;
  n := StrToInt(edInput1.Text);
  Res := Res shl n;
  mmOutput1.Lines.Add(IntToStr(Res));
  edInput1.SetFocus;
end;
```

3.17. Найпростіші введення та виведення

У попередніх розділах у прикладах розв'язання задач нами вже розглядалися деякі засоби введення і виведення даних, які використовуються в програмах, написаних мовою Delphi.

До цих засобів, у першу чергу, можна віднести такі компоненти, як однорядкове редаговане текстове поле `TEdit`, багаторядкове редаговане текстове поле `TMemo` і мітку `TLabel`.

Найпростіша методика введення така:

1. Елемент даних, що вводиться (число, символ, текстовий рядок), вводиться з клавіатури з відображенням його в компоненті `TEdit` (при цьому значення у вигляді рядка записується у властивість `Text`).
2. За деякою командою (наприклад, за подією OnClick, яка генерується кліком мишкою над компонентом `TButton` – кнопка або `TBitBtn` – кнопка з зображенням) проводиться опрацювання введеного рядка. При цьому треба пам'ятати, що властивість `Text` компонента `TEdit` є текстовим рядком, у зв'язку з чим при введенні числових даних уведене значення повинне бути перетворене до числового виду (наприклад, за допомогою функції `StrToInt` при введенні цілочислових даних і `StrToFloat` при введенні дійсних чисел або за допомогою інших засобів).
3. Якщо необхідно організувати повторне введення за допомогою `TEdit` без активізації його за допомогою мишки, то слід передати цьому компоненту так званий фокус введення, що забезпечується звертанням до методу `SetFocus` цього компонента.

Відзначимо також наступне. У поле введення можна занести відразу декілька чисел, розділивши їх одним або декількома пробілами. У цьому випадку потрібно подбати про виділення кожного з числових значень.

Нагадаємо також (див. зауваження до прикладу 2.2), що в русифікованій версії Windows як роздільник цілої і дробової частин при записі дійсних чисел вживається кома, а не десяткова точка, як це має місце в

англомовній версії Windows. Щоб уникнути неоднозначності в трактуванні запису дійсних чисел, можна рекомендувати присвоювати системній змінній `DecimalSeparator`, у якій зберігається поділяючий символ, загальноприйнятий у програмуванні поділяючий символ «крапка» (тобто виконати оператор `DecimalSeparator:='.';`).

Для виведення даних можна також використовувати однорядковий редактор `TEdit`, за необхідності перетворюючи даних, які підлягають виведенню, до рядкового подання за допомогою функцій `IntToStr`, `FloatToStr`, `BoolToStr`, `DateTimeToStr` або за допомогою інших підпрограм. У цьому випадку достатньо виконати присвоювання виведеного рядка властивості `Text` однорядкового редактора.

Для виведення даних можна також скористатися присвоюванням виведеного текстового рядка властивості `Caption` мітки `TLabel`.

Компонент `TMemo` придатний і для виведення, і для введення даних, як і компонент `TEdit`. При введенні даних сформований виведений рядок тексту записується у властивість `Lines` компонента `TMemo` за допомогою методу `Add` (див. приклад 2.2 й інші приклади). Оскільки властивість `Lines` є доступною і для читання, дані можуть бути набрані у вікні багаторядкового редактора, після чого переписані в змінні, що використовуються в програмі (з можливими перетвореннями, аналогічними тим, що виконуються при введенні за допомогою однорядкового редактора `TEdit`). Слід ураховувати, що рядки властивості `Lines` індексуються (нумеруються) від нуля, і при читанні даних з цієї властивості необхідно організовувати цикл.

Приклади введення і виведення даних (за винятком використання багаторядкового редактора `TMemo` для введення) розглядалися раніше. Однак особливістю розглянутих раніше прикладів є те, що в них виконувалося введення тільки одного елемента даних (числа). При введенні декількох елементів даних можна або для кожного елемента створювати своє поле введення, або розміщати на формі кілька кнопок, або здійснювати деяке програмне керування процесом введення.

```
//Приклад 3.6
//Увести три цілих числа. Вивести їхню суму.
```

Створимо форму із властивістю Caption, що містить текст Приклад введення декількох чисел. Розмістимо на формі багаторядкове поле виведення mmOutput1 з очищеною властивістю Lines, три однорядкових редактори edInput1, edInput2, edInput3, у властивості Text яких записані нулі, і компонент Button1 з властивістю Caption, що містить текст Результат. Над однорядковими полями введення помістимо мітки lbOutput1, lbOutput2,

lbOutput3 з властивостями Caption, що містять імена трьох змінних a, b, c, котрі оголосимо у секції **implementation** модуля як змінні типу Integer.

Для розв'язання задачі може бути використаний такий опрацьовувач події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  a := StrToInt(edInput1.Text);
  b := StrToInt(edInput2.Text);
  c := StrToInt(edInput3.Text);
  mmOutput1.Lines.Add('Сума дорівнює ' + IntToStr(a + b + c));
end;
```

Слід зазначити, що перед тим, як натиснути кнопку Результат, необхідно в трьох однорядкових полях введення набрати значення, що будуть вводитися, переключаючись від одного до іншого або за допомогою кліків мишкою, або натисканням клавіші Tab. Це спричиняє можливість появи помилок на етапі виконання, оскільки користувач може попросту забути заповнити одне з полів введення і натиснути при цьому кнопку Результат.

Дещо простішим буде процес введення даних за наявності на формі одного однорядкового редактора edInput1, трьох кнопок Button1, Button2, Button3, а також мітки lbOutput1 над однорядковим редактором.

```
//Приклад 3.7
//Увести три цілих числа. Вивести їхню суму.
```

Присвоїмо властивостям Caption компонентів Button1, Button2, Button3 значення у вигляді текстів Введення a, Введення b, Введення c та створимо для кнопок три таких опрацьовувача події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  a := StrToInt(edInput1.Text);
  Button1.Enabled := False;    //Кнопка Button1 є недоступною,
  Button2.Enabled := True;      //а кнопка Button2 є доступною
  lbOutput1.Caption := 'Уведіть значення b';
  edInput1.SetFocus;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  b := StrToInt(edInput1.Text);
  Button2.Enabled := False;    //Кнопка Button2 є недоступною,
  Button3.Enabled := True;      //а кнопка Button3 є доступною
  lbOutput1.Caption := 'Уведіть значення c';
  edInput1.SetFocus;
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  c := StrToInt(edInput1.Text);
  Button3.Enabled := False;    //Кнопка Button3 є недоступною,
  Button1.Enabled := True;      //а кнопка Button1 є доступною
  mmOutput1.Lines.Add('Сума дорівнює ' + IntToStr(a + b + c));
  lbOutput1.Caption := 'Уведіть значення a';
  edInput1.SetFocus;
end;
```

Крім того, створимо такий опрацьовувач події OnCreate для форми:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  edInput1.TabOrder := 0;
  lbOutput1.Caption := 'Уведіть значення a';
      //Змінюємо значення властивості Enabled кнопок
  Button1.Enabled := True;        //Кнопка Button1 є доступною
  Button2.Enabled := False;     //Кнопка Button2 є недоступною
  Button3.Enabled := False;     //Кнопка Button3 є недоступною
end;
```

Розглянемо стисло особливості опрацьовувачів подій.

Характерною рисою візуальних компонентів є те, що вони можуть бути в кожен конкретний момент часу як доступними, так і недоступними. Доступністю компонентів керує властивість `Enabled` типу `Boolean`. Якщо ця властивість має значення `True`, то компонент доступний, у противному разі – недоступний. Якщо компонент доступний, то він може реагувати на події від миші й клавіатури (недоступний компонент звичайно зафарбований сірим кольором).

Наведений вище опрацьовувач події OnCreate форми виводить у мітці підказку про необхідність введення значення `a`, установлює доступність кнопки Button1 і недоступність кнопок Button2 і Button3 для того, щоб забезпечити правильне введення першого числа. Надалі переключення порядку введення забезпечують опрацьовувачі події OnClick кнопок. Для того щоб однорядковий редактор відразу ж при запуску програми одержував фокус введення, в опрацьовувачі події OnCreate форми властивості `TabOrder` однорядкового редактора введення присвоєне значення `0`. Це значення вказує, що однорядковий редактор є найпершим з компонентів при обході їх за допомогою клавіші Tab, а отже, йому буде присвоєний фокус введення (за умовчанням цій властивості компонентів присвоюється значення в міру їх створення, починаючи від 0).

Опрацьовувачі події OnClick кнопок крім того, що переключають активність кнопок, забезпечують виведення необхідних підказок у мітку і переключають фокусу введення.

Зазначимо також, що недоступними є і невидимі в даний момент (приховані) компоненти. Прихований компонент також не реагує на події від миші й клавіатури (у тому числі від клавіші Tab) і не може одержати фокус введення. Приховується або показується компонент за допомогою властивості `Visible` типу `Boolean` або за допомогою методів `Hide` і `Show`.

Розглянемо ще один варіант організації введення декількох значень на прикладі тієї ж елементарної задачі.

```
//Приклад 3.8
//Увести три цілих числа. Вивести їхню суму.
```

Скористаємося тією же формою, що й у прикладі 3.7, але розмістимо кнопки таким чином, щоб вони всі накладалися одна на одну. Створимо для кнопок такі опрацьовувачі події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  a := StrToInt(edInput1.Text);
  Button1.Hide;                          //Сховати кнопку Button1
  Button2.Show;                         //Показати кнопку Button2
  lbOutput1.Caption := 'Уведіть значення b';
  edInput1.SetFocus;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  b := StrToInt(edInput1.Text);
  Button2.Hide;                          //Сховати кнопку Button2
  Button3.Show;                         //Показати кнопку Button3
  lbOutput1.Caption := 'Уведіть значення c';
  edInput1.SetFocus;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  c := StrToInt(edInput1.Text);
  Button3.Hide;                          //Сховати кнопку Button3
  Button1.Show;                         //Показати кнопку Button1
  mmOutput1.Lines.Add('Сума дорівнює ' + IntToStr(a + b + c));
  lbOutput1.Caption := 'Уведіть значення a';
  edInput1.SetFocus;
end;
```

Крім того, створимо для форми такий опрацьовувач події OnCreate:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.Visible := True;     //Зробити видимою кнопку Button1
```

```
  Button2.Visible := False;          //Зробити невидимою Button2
  Button3.Visible := False;          //Зробити невидимою Button3
  EdInput1.TabOrder := 0;
  lbOutput1.Caption := 'Уведіть значення a';
end;
```

Розглянемо, як буде в даному випадку виконуватися програма. При запуску програми створюється форма з виконанням її опрацьовувача події OnCreate, який одну з кнопок (Button1) робить видимою, а дві інші – невидимими, забезпечуючи тим самим можливість введення тільки значення змінної a, оскільки тільки кнопка Button1 буде реагувати на події від мишки і клавіатури. Крім того, буде переданий фокус введення однорядковому редактору.

Після запису першого значення у поле введення однорядкового редактора необхідно клікнути мишкою над кнопкою введення, у відповідь на що виконається процедура TForm1.Button1Click, яка присвоїть уведене значення змінній a, після чого за допомогою методів Hide і Show зробить невидимою кнопку Button1 і видимою кнопку Button2 відповідно та передасть фокус введення однорядковому редактору.

Опрацьовувачі події OnClick компонентів Button2 і Button3 працюють аналогічно опрацьовувачу події OnClick компонента Button1, але опрацьовувач TForm1.Button3Click додатково виводить результат обчислень.

Досить зручним для організації введення є використання описаної в модулі Dialogs функції InputBox, що служить для виклику діалогового вікна введення, яке дозволяє користувачеві редагувати рядок, що вводиться. При звертанні до функції InputBox на екран виводиться діалогове віконце, аналогічне тому зображеному на рис. 3.1.

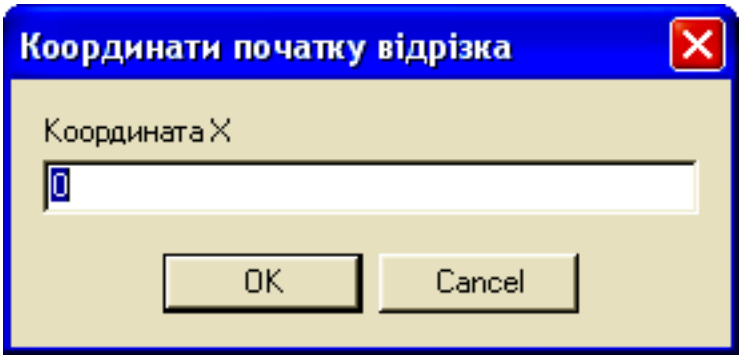


Рис. 3.1. Віконце функції InputBox

Віконце містить однорядкове редаговане текстове поле, яке автоматично одержує фокус введення і містить деяке значення за умовчанням. Користувач може або погодитися з відображеним значенням за умовчанням, або набрати в редакторському вікні нове значення. Безпосередньо

введення здійснюється кліком мишкою над кнопкою OK (відповідає натисканню клавіші Enter) або кліком мишкою над кнопкою Cancel (відповідає натисканню клавіші Esc).

Функція `InputBox` має такий формат:

`InputBox(`заголовок, підказка, рядок_за_умовчуванням`).`

Тут заголовок – рядок, що задає заголовок (на рис. 3.1 – Координати початку відрізка);

підказка – рядок-підказка для користувача (на рис. 3.1 – Координата x);

рядок_за_умовчуванням – рядок, що з'являється в однорядковому редакторі даного віконця при його виведенні на екран і вводиться при натисканні кнопки Cancel (на рис. 3.1 – рядок 0) .

При виборі користувачем кнопки Cancel (або натисканні клавіші Esc) функція повертає значення за умовчанням (третій параметр), при виборі ж кнопки OK (або натисканні клавіші Enter) функція повертає значення, що міститься в однорядковому редакторі.

Функція повертає рядок, який за необхідності може бути перетворений до потрібного формату за допомогою однієї з функцій `StrToXXX`.

Функцію `InputBox` рекомендується використовувати в тому випадку, коли додатку байдуже, яка кнопка вибиралася – OK або Cancel. Якщо ж додаток повинен розпізнавати обрану кнопку, рекомендується застосовувати функцію `InputQuery`.

Функція `InputQuery` має ті ж самі три параметри, що й функція `InputBox`, але третій параметр є таким, за допомогою якого повертається або набране в редакторському вікні значення (якщо вибиралася кнопка OK), або значення за умовчанням (при виборі кнопки Cancel). Записане в третій параметр значення і є введеним значенням. Результат роботи функції `InputQuery` є – значення: `True` при виборі кнопки OK або `False` при виборі кнопки Cancel.

Особливістю введення за допомогою функцій `InputBox` і `InputQuery` є те, що звертання до цих функцій може бути здійснене багаторазово й не сполучене з виконанням яких-небудь опрацьовувачів подій. Ці функції зручно використовувати при циклічному введенні (див. приклад 4.10), зокрема, при введенні масивів (див. приклади 5.3 й 11.2).

У даному ж розділі розглянемо чисто ілюстративний простий приклад використання функції `InputBox`.

```
//Приклад 3.9
//Увести два цілих числа.
//Чи є їхня сума парною?
```

Скористаємося формою з прикладу 2.2, видаливши з неї однорядковий редактор введення і мітку виведення. Тоді для розв'язання задачі можна вдатися до такого опрацьовувача події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Odd(StrToInt(InputBox('Введення цілого числа',
                           'Перше число', '0')) +
         StrToInt(InputBox('Введення цілого числа',
                           'Друге число', '0')))
  then mmOutput1.Lines.Add('No')
  else mmOutput1.Lines.Add('Yes');
end;
```

У цьому випадку здійснюється дворазове звертання до функції `InputBox` для введення двох цілих чисел. Якщо у віконці введення буде натиснута кнопка `OK`, то на подальше опрацювання надійде набране в однорядковому редакторі значення. При натисканні ж кнопки `Cancel` буде введене значення `0`, що задане як третій параметр функції `InputBox`.

Запитання для контролю і самоконтролю

1. Охарактеризуйте групи символів, які використовуються в програмах, написаних мовою Delphi.
2. Що таке службове слово?
3. Що таке стандартна директива?
4. Що таке визначене ім'я?
5. Що таке ідентифікатор?
6. Як впливають на роботу програми вдалі й невдалі імена?
7. Які з перерахованих нижче послідовностей символів можна вважати гарними і поганими іменами? Які з них є неприпустимими для використання як імена?

   а) `One` *б*) `.pas` *в*) `R2ab` *г*) `FullContact` *д*) `_Invalid`
   е) `__` *ж*) `134` *и*) `_age_` *к*) `_13qwerty` *л*) `5Variant`
8. Дайте визначення поняттю «змінна».
9. Охарактеризуйте елемент даних, що називається константою.
10. Яким способом здійснюється явне задавання цілочислової константи?
11. Як явно задається дійсна константа?
12. Що таке «тип» і що він визначає?
13. Які стандартні скалярні типи існують у Delphi?
14. У чому різниця між типами `LongInt` і `LongWord`?
15. У чому різниця між цілочисловою і дійсною змінними?

16. Що необхідно внести в текст програми, щоб у ній можна було використовувати нестандартну змінну?
17. Для чого використовується розділ **var**?
18. Які типи змінних були б правильними для зберігання в них такої інформації: вік, площа земельної ділянки, номер квартири, кількість зірок у галактиці, середній рівень опадів за місяць? Визначте імена для цієї інформації.
19. Яку роль у програмі відіграють мітки?
20. Для чого служать підпрограми?
21. Яке призначення коментарів?
22. Які типи коментарів ви знаєте? У чому їхня різниця?
23. Чи можуть коментарі займати кілька рядків?
24. Чи можуть бути вкладені коментарі?
25. Чи є помилки у фрагменті програми, що наводиться нижче, у припущенні, що всі змінні описано правильно? За наявності помилок внесіть виправлення.

```
S := a * b; {Спочатку обчислюємо площу основи, //Зайве
            після чого обчислюємо об'єм паралелепіпеда}
V := S * c;
```

26. Дайте відповідь на задане вище запитання, розглянувши наступний фрагмент програми:

```
S := a * b;          //Площа основи {Без змінної S можна
    обійтися, обчислюючи відразу об'єм паралелепіпеда}
V := S * c;
```

27. Що таке директива компілятору і як вона визначається в програмі?
28. Визначте поняття «вираз».
29. Що таке «пріоритет операцій»?
30. Навіщо потрібні круглі дужки у виразах, якщо є система пріоритетів операцій?
31. Що відбувається, якщо в правильному виразі операнди мають різні типи?
32. Які помилки були допущені в наведеному нижче виразі при записі в програмі арифметичного виразу $\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ (замість σ і μ використані відповідно імена `sigma` і `myu`)?

```
1/sigma*Sqrt 2Pi*Exp(-Sqr(x-myu)/2*Sqr(sigma))
```

33. Чому дорівнює значення наведених нижче виразів:
    а) `8 + 2 * 7`

б) `72 / 5`

в) `72 div 5`

г) `72 mod 5` ?

34. Дано такі арифметичні вирази:

а) $2R^2 \sin A \sin B \sin C$;

б) $\frac{bc}{2R}$;

в) $\sqrt{p(p-a)(p-b)(p-c)}$;

г) $\frac{1}{b+c}\sqrt{bc(a+b+c)(b+c-a)}$.

Нижче наведена їхня реалізація в програмі. Чи є помилки в реалізації? Якщо помилки є, укажіть і виправте їх:

а)
```
2 * R * R Sin A Sin B Sin C
```

б)
```
b * c / 2 * R
```

в)
```
Sqrt(p * (p - a) * (p - b) * (p - c))
```

г)
```
1 / (b + c) * Sqrt(b * c * (a + b + c)(b + c - a))
```

35. Що таке «порожній оператор»?
36. Яке призначення оператора присвоювання?
37. Дайте опис формату оператора присвоювання.
38. Що відбудеться, якщо беззнаковій змінній присвоїти від'ємне значення?
39. Нехай змінна `w` має тип `Word`. Чи є помилковим фрагмент програми з двох операторів, що наведений нижче? Якщо помилка має місце, то опишіть її. Якщо помилка відсутня, то чому буде дорівнювати кінцеве значення змінної `w`?

```
w := 0;
Dec(w);
```

40. Нехай змінна `Variable` має тип `Integer`. Яке значення буде міститися в цій змінній після виконання таких двох операторів:

```
Variable := MaxInt;
Variable := variable - 1;
```

41. Для чого служить розділ опису типів?
42. Дайте стислу характеристику кожного зі стандартних скалярних типів Delphi.
43. Чим характерний відрізковий тип?
44. Що таке перелічений тип?

45. Що таке іменована константа? Що необхідно зробити для забезпечення можливості використання власних іменованих констант?
46. Що таке типізована константа? Як вона оголошується?
47. Чому замість констант, що задаються явно, краще використовувати іменовані константи?
48. Як організуються найпростіші введення і виведення в Delphi?
49. Які типи у Delphi вважаються еквівалентними? Наведіть приклади.
50. Які типи в Delphi вважаються сумісними? Наведіть приклади.
51. Яким способом в Delphi здійснюється явне перетворення типів?
52. Які бітові операції існують у Delphi?
53. У чому відмінність звичайних логічних операцій від логічних операцій над бітами?
54. Чи всі з перерахованих нижче виразів є синтаксично правильними? Чому будуть дорівнювати значення тих з них, які є правильними?
    а) `2 * $20`
    б) `2 and 2 + 2 * 2`
    в) `10 or 15 - 7 or 13 and 126`
    г) `7 * (132 xor 300 + $A * $1A)`
55. Опишіть найпростішу методику введення.
56. Які компоненти можуть використовуватися для введення даних?
57. Які компоненти можуть використовуватися для виведення даних?
58. Для чого і як використовуються функції `InputBox` і `InputQuery`?

Завдання для практичного відпрацьовування матеріалу

1. Запишіть мовою Delphi наступні вирази:

$$а)\ \frac{\frac{y+z}{2}x+3.1}{(2y-z)x+7};$$

$$б)\ \frac{1}{3+\frac{1}{3+\frac{1}{2w}}}+w;$$

$$в)\ \frac{-b+\sqrt{b^2-4ac}}{2a};$$

$$г)\ \frac{1}{\sqrt{n+3}}\left(e^{1/\sqrt{n}}-1\right).$$

2. Дано три цілі невід'ємні числа, хоча б одне з яких відмінне від нуля. Обчислити суму цих чисел і відсотковий внесок кожного з них у суму.
3. Дано дійсні числа. Обчислити їхню суму. Як варіант розв'язання пропонується такий: розмістити на формі дві кнопки, перша з яких забезпечує введення одного числа, а друга – виведення суми всіх чисел, введених до кліку мишкою над цією кнопкою.
4. Пряма на площині може бути задана рівнянням $ax + by + c = 0$, де a й b одночасно не дорівнюють нулю. Будемо розглядати тільки прямі з цілими коефіцієнтами a, b, c. Нехай дані коефіцієнти рівнянь двох прямих: a_1, b_1, c_1 і a_2, b_2, c_2. Визначити:

   а) чи є ці прямі збіжними;

   б) чи є ці прямі паралельними.

   Вивести значення `1` при позитивній відповіді на запитання і значення `-1` при негативній відповіді. Скористатися явним перетворенням типу.

4. КОНСТРУКЦІЇ КЕРУВАННЯ

4.1. Найпростіший оператор перевірки умови

Досить часто в програмі необхідно виконувати деякі дії в тому випадку, коли є істинною деяка умова. Щоб задати яке-небудь запитання, в Delphi застосовують оператор **if** (оператор галуження, умовний оператор):

```
if умова then оператор;
```

Якщо умова є істинною (`True`), то виконується оператор, розташований після службового слова **then**, а потім виконується наступний оператор; якщо перевірка умови дає значення `False`, то оператор, що стоїть після **then**, пропускається (див. рис. 4.1, *а*). Як умова може задаватися будь-який вираз, результатом обчислення якого є одне з булівських значень `True` або `False`.

Наведена конструкція зручна у випадку, коли деяка дія повинна виконуватися тільки при позитивній відповіді на запитання і не виконуватися, якщо отримано негативну відповідь. Якщо ж при негативній відповіді слід виконати іншу дію, то керуючу конструкцію доведеться повторити з умовою, протилежною умові, що перевіряється в першій конструкції:

```
if умова then оператор1;
if протилежна_умова then оператор2;
```

Розв’яжемо таку задачу.

```
//Приклад 4.1
//Дано число x. Обчислити y=-1, якщо sin(x)<0,
//y=1 у противному разі.
```

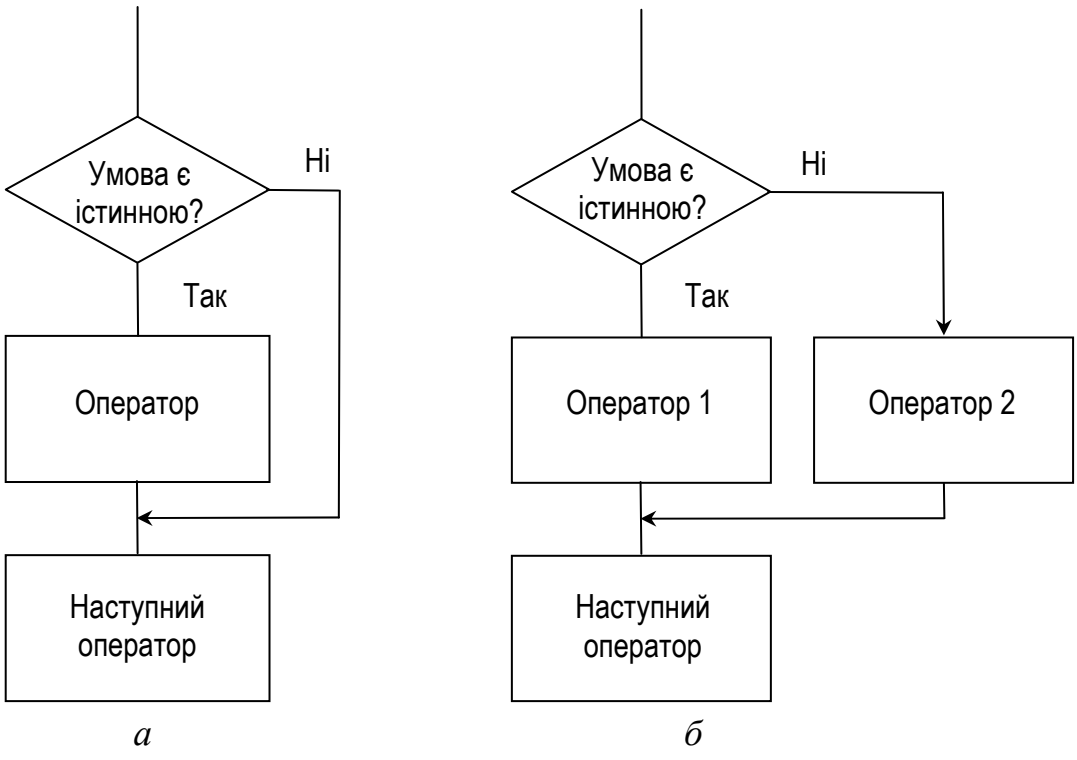
Скористаємося кінцевим варіантом форми з прикладу 3.4 і дещо змінимо код модуля: у розділі **implementation** замість змінної `sign` оголосимо змінну `Y` типу `Integer`, а для опрацьовувача події OnClick компонента Button1 створимо такий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  x := StrToFloat(edInput1.Text);
  if Sin(x) < 0 then Y := -1;                  //Перевірка умови
  if Sin(x) >= 0 then Y := 1;                 //Протилежна умова
  mmOutput1.Lines.Add(IntToStr(Y));
  edInput1.SetFocus;
end;
```

Код опрацьовувача події OnCreate форми залишимо таким же, як й у прикладі 3.4.

Більш загальною конструкцією є конструкція **if** – **then** – **else**, ідея якої полягає в тому, щоб виконувати тільки один з пунктів – **then** (якщо умова є істинною) або **else** (якщо є хибною), але ніколи не виконувати обидва (див. рис. 4.1, *б*):

if умова **then** оператор1 **else** оператор2;



а *б*

Рис. 4.1. Можливі схеми дії оператора **if**

З використанням такої конструкції код опрацьовувача події OnClick компонента Button1 набуде такого вигляду:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  x := StrToFloat(edInput1.Text);
```

```
  if Sin(x) < 0 then
    Y := -1
  else
    Y := 1;
  mmOutput1.Lines.Add(IntToStr(Y));
  edInput1.SetFocus;
end;
```

Зазначимо, що після **then** може стояти тільки один оператор. За необхідності виконання декількох операторів вони повинні бути взяті в операторні дужки **begin** – **end** (формується так званий ***складений оператор***).

```
//Приклад 4.2
//Створити елементарний тренажер для перевірки правильності
//множення цілих чисел: програма виводить два цілих числа, а
//користувач уводить результат їхнього перемножування, після
//чого програма виводить правильний результат і повідомляє
//про помилку, якщо вона є, з видачею звукового сигналу.
```

Створимо форму fm1, на яку помістимо багаторядковий редактор mmOutput1 для виведення (для властивості ScrollBars задамо значення ssBoth – дві смуги скролінга), однорядкове поле введення edInput1 (властивість Caption очистимо) і компонент Button1 із властивістю Caption, що містить текст Увести. Над однорядковим редактором помістимо мітку lb1 з властивістю Caption, що містить текст Уведіть добуток.

У секції **implementation** модуля опишемо дві змінні a і b типу Integer. Крім того, способом, аналогічним раніше розглянутому (клік мишкою у Дереві Об'єктів над значками, що відповідають формі й кнопці), створимо два опрацьовувача подій (OnClick для компонента Button1 і OnCreate для форми), що містять такі коди:

```
procedure Tfm1.Button1Click(Sender: TObject);
begin
  if StrToInt(edInput1.Text) <> a * b then begin
    mmOutput1.Lines.Add('Помилка');
    MessageBeep(MB_OK);
  end
  else
    mmOutput1.Lines.Add('Помилка');
  mmOutput1.Lines.Add('Добуток дорівнює ' + IntToStr(a * b));
  a := Random(100);
  b := Random(100);
  mmOutput1.Lines.Add('  a=' + IntToStr(a));
  mmOutput1.Lines.Add('  b=' + IntToStr(b));
  edInput1.SetFocus;
end;
```

```
procedure Tfm1.FormCreate(Sender: TObject);
begin
  Randomize;
  a := Random(100);
  b := Random(100);
  mmOutput1.Lines.Add('  a=' + IntToStr(a));
  mmOutput1.Lines.Add('  b=' + IntToStr(b));
  edInput1.Text := '0';
end;
```

У коді процедури `Tfm1.Button1Click` усередині оператора **if** після службового слова **then** ужито складений оператор: якщо опустити операторні дужки **begin** – **end**, то видача звукового сигналу, здійснювана функцією `MessageBeep` зі значенням параметра `MB_OK` (див. підрозд. 3.8) буде проводитися й у випадку неправильної відповіді, і при правильній відповіді, у той час як передбачається, що звуковий сигнал повинен сигналізувати про помилку. В процедурі `Tfm1.Button1Click` за допомогою функції `Random` випадково генеруються два цілочислові значення з інтервалу `0..99` із записом їх у змінні `a` і `b`, виводяться ці значення у вікно багаторядкового редактора `mmOutput1`, після чого фокус введення передається однорядковому редактору `edInput1` для того, щоб дати користувачеві можливість увести результат.

Код процедури `Tfm1.FormCreate` містить оператори, які виконуються при створенні форми. Насамперед, це випадкове генерування і виведення перших значень співмножників `a` і `b`. Щоб при повторному запуску програми дані не повторювалися, перший оператор процедури `Tfm1.FormCreate` (а саме, виклик процедури `Randomize`) забезпечує випадкову початкову установку генератора `Random`. Останній оператор опрацьовувача події OnCreate форми програмно записує `0` у властивість `Text` редактора `edInput1`, визначаючи тим самим «передбачуване» початкове значення добутку перших двох чисел, яке буде виведене у віконці однорядкового редактора `edInput1` для сприйняття його користувачем програми.

Щоб відразу ж передати фокус введення редактору `edInput1` для приймання першого результату, створимо для форми опрацьовувач події OnShow, що виконується при «показі» форми. Для цього в Інспекторі Об'єктів виберемо форму і перейдемо для неї на вкладку Events (Події), після чого виконаємо подвійний клік мишкою праворуч від імені властивості OnShow. У відповідь на клік у секції **interface** модуля в описі класу `Tfm1` перед службовим словом **private** буде автоматично вставлений такий рядок:

```
    procedure FormShow(Sender: TObject);
```

Крім того, перед термінатором **end** з крапкою автоматично вставиться заготівка для коду опрацьовувача події OnShow форми і буде здійснений перехід у цю заготівку. Скорегуємо вставлений код у такий спосіб:

```
procedure Tfm1.FormShow(Sender: TObject);
begin
  edInput1.SetFocus;
end;
```

Іноді деяку дію потрібно виконувати тільки у випадку хибності тієї умови, яка перевіряється. Тоді після **then** вказують порожній оператор:

```
if умова then else оператор;
```

У цьому разі доцільніше подати умову, що перевіряється, у вигляді протилежної умови:

```
if протилежна_умова then оператор;
```

Наприклад, еквівалентними за одержуваним результатом є такі дві конструкції:

```
if x < x1 then else z := x1;
```

та

```
if x >= x1 then z := x1;
```

Слід зазначити, що і оператор1, і оператор2, які фігурують в операторі **if**, можуть бути будь-якими операторами, у тому числі операторами циклу (див. підрозд. 4.5) та операторами, що перевіряють умову. Тому, якщо необхідно вибрати одну з декількох взаємовиключаючих альтернатив, на допомогу приходить вкладений оператор **if**. Формат вкладеного оператора **if** такий:

```
if умова1
  then оператор1
  else
    if умова2
      then оператор2
      else оператор3;
```

або

```
if умова1
  then
    if умова2
      then оператор1
      else оператор2
  else оператор3;
```

Відповідність між **then** і **else** установлюється в такий спосіб: кожному **then** відповідає найближче наступне службове слово **else**, не задіяне при встановленні відповідності з іншим **then**.

Використання операторних дужок **begin** – **end** у конструкціях з оператором **if** зовсім не обов'язкове. Це справедливо й у випадку вкладення декількох операторів **if**. Однак у складних вкладених конструкціях без операторних дужок буває важко розібратися, який оператор входить у ту або іншу гілку **then** або **else**.

Варто також пам'ятати, що пробіли та відступи роблять програму значно зрозумілішою програмісту, не впливаючи на роботу компілятора. Навіть якщо за допомогою відступу буде показано, що дана гілка **else** стосується деякої гілки **then**, компілятор на це не зреагує й установить відповідність згідно із закладеним у нього правилом, яке описане вище.

Розглянемо, наприклад, такий фрагмент програми:

```
if x >= 100 then
  if x > 1000 then
    Memo1.Lines.Add('Більше за 1000')
else
  Memo1.Lines.Add('Менше 100');
```

Якщо проаналізувати відступи, то можна дійти висновку, що цей фрагмент забезпечує аналіз значення змінної `x` що до того, що менше воно від `100` або більше від `1000`, результатом чого повинне бути виведення відповідного повідомлення.

Якщо умова, що перевіряється в першому рядку, є істинною, то здійснюється перевірка умови, записаної в другому рядку. У випадку її істинності виконується оператор, розташований у третьому рядку, і виводиться правильне повідомлення. Однак якщо результатом перевірки другої умови буде значення `False`, виконається оператор, що стоїть після службового слова **else**, і буде виведене неправильне повідомлення. Мало того, якщо `x` менше за `100`, взагалі ніяке повідомлення виводитися не буде, хоча, судячи з відступу, воно виводитися повинне.

Допишемо в кінець першого рядка службове слово **begin** і вставимо перед **else** службове слово **end**:

```
if x >= 100 then begin
  if x > 1000 then
    Memo1.Lines.Add('Більше за 1000')
end
else
  Memo1.Lines.Add('Менше 100');
```

Тепер усе, що знаходиться між **begin** та **end** сприймається як один оператор, і службове слово **else** виявляється явно пов'язаним з першим службовим словом **then**, результатом чого буде правильна робота даного фрагмента програми.

Зазначимо, що якщо аналізоване значення перебуває в інтервалі від 100 до 1000, останній фрагмент програми не виводить ніяке повідомлення. Було б краще виводити яке-небудь повідомлення й у цьому випадку, додавши у внутрішній оператор **if** гілку **else**.

Найчастіше для перевірки складних умов замість вкладення операторів використовують булівські операції. Так, фрагмент програми, що розглядався вище, може бути перетворений у такий спосіб:

```
if (x >= 100) and (x <= 1000) then
then
  Memo1.Lines.Add('В інтервалі від 100 до 1000')
else
  if x > 1000 then
    Memo1.Lines.Add('Більше за 1000')
  else
    Memo1.Lines.Add('Менше 100');
```

У цьому фрагменті програми вже здійснюється виведення одного з трьох повідомлень у результаті перевірки значення змінної x.

4.2. Оператор вибору

Оператор вибору, або оператор **case**, можна трактувати як деяке запитання, що має велику кількість відповідей (а не тільки дві, як це має місце в операторі **if** – **then** – **else**). Його формат такий:

```
case селектор of
      альтернатива1: оператор1;
      альтернатива2: оператор2;
                ...
      альтернативаN: операторN;
end;
```

Селектором може бути будь-який вираз порядкового типу (наприклад, Integer, Char, перелічений тип, але не Real). Поточне значення селектора зумовлює те, який з операторів потрібно виконати. Елементи альтернатива1, альтернатива2, ..., альтернативаN є константними значеннями, які може приймати селектор. Їх тип і тип селектора повинні бути сумісні за присвоюванням. Якщо селектор прибирає значення

альтернатива1, то виконується оператор1, а всі інші пункти пропускаються; якщо селектор набуває значення альтернатива2, то виконується оператор2 і т. д. Кожен з цих операторів може бути або простим, або складеним, або порожнім (тільки крапка з комою).

```
//Приклад 4.3
//Встановити, чи кратне дане ціле число трьом.
```

Скористаємося для розв'язання задачі формою з прикладу 2.2, присвоївши властивостям Caption форми і мітки відповідно до значення Приклад на використання case та Уведіть ціле число. У секції **implementation** модуля опишемо змінні:

```
    var
      number, remainder: Integer; //Число і остача від ділення
```

Удамося до такого опрацьовувача події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  number := StrToInt(edInput1.Text);
  remainder:= number mod 3;
  case remainder of
    0: mmOutput1.Lines.Add('Число ' + IntToStr(number) +
                           ' є кратним 3');
    1: mmOutput1.Lines.Add('Число ' + IntToStr(number) +
                           ' є кратним 3 із остачею 1');
    2: mmOutput1.Lines.Add('Число ' + IntToStr(number) +
                           ' є кратним 3 із остачею 2');
  end; //case
  edInput1.SetFocus;
end;
```

В альтернативі вибору можна зазначити більше одного значення селектора (їх перелічують через кому), частиною альтернативи може бути відрізок (ряд послідовних значень) – його задають зазначенням початку і кінця, розділеними двома крапками (наприклад: `7..27`).

```
//Приклад 4.4
//Класифікувати малі латинські літери за правилом:
//літери b та d – клас 1; c, j, q – клас 2; a – клас 3;
//h та всі літери від r по z – клас 4;
//інші літери – клас 5.
```

Для розв'язання задачі скористаємося формою з прикладу 4.2, змінивши значення властивості Caption мітки на Уведіть літеру (a-z) для класифікації.

У секції **implementation** модуля опишемо змінні:

```
    var
      letter: 'a'..'z';
      let_type: Integer;
```

і скористаємося таким опрацьовувачем події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  letter := edInput1.Text[1];
  case letter of
    'b', 'd'                : let_type := 1;
    'c', 'j', 'q'           : let_type := 2;
    'a'                     : let_type := 3;
    'r'..'z', 'h'           : let_type := 4;
    'e'..'g', 'i', 'k'..'p' : let_type := 5;
  end; //case
  mmOutput1.Lines.Add('Літера ' + letter +
                      ' у класі ' + IntToStr(let_type));
  edInput1.SetFocus;
end;
```

Зазначимо, що в цьому випадку програма не захищена від введення великих латинських літер, а також символів, що не є латинськими літерами, у зв'язку з чим її рекомендується доробити самостійно (при введенні неприпустимого символу номер класу визначається номером раніше визначеного класу; при цьому початковий номер дорівнює нулю, оскільки глобальна змінна `let_type` за умовчанням має значення `0`).

Обмеження:

1) селектор повинен мати який-небудь порядковий тип;
2) кожна альтернатива має бути константою, відрізком чи їх списком, але не змінною або виразом.

Якщо серед альтернатив перелічені не всі можливі значення селектора, то при одержанні селектором такого значення оператор **`case`** ніби пропускається.

Мова програмування Delphi дозволяє згрупувати всі значення, що не ввійшли в жодну з альтернатив, у пункті **`else`**:

```
//Приклад 4.5
//Уводиться ціле додатне число – вік у роках.
//Вивести його разом з одним зі слів "рік", "роки", "років".
```

Для розв'язання задачі скористаємося остаточною формою з прикладу 4.3, описавши в секції **`implementation`** модуля одну змінну:

```
    var
      n: Integer;
```

Скористаємося також таким опрацьовувачем події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  n := StrToInt(edInput1.Text);
  if (n mod 100 > 10) and (n mod 100 < 20)
  then
    mmOutput1.Lines.Add(edInput1.Text + ' років')
  else
  case n mod 10 of
    1:       mmOutput1.Lines.Add(edInput1.Text + ' рік');
    2, 3, 4: mmOutput1.Lines.Add(edInput1.Text + ' роки');
    else     mmOutput1.Lines.Add(edInput1.Text + ' років');
  end; //case
  edInput1.SetFocus;
end;
```

Зауваження. Програма орієнтована на аналіз остачі від ділення цілого n числа на 10. Ця остача розглядається як остання цифра цілого числа. Оскільки остача від ділення також має знак, програма не захищена від введення від'ємних значень. Якщо необхідна виділення цифр цілих від'ємних чисел (звичайно, це не вік!), то перший оператор написаної вище процедури повинен мати такий вигляд:

```
    n := Abs(StrToInt(edInput1.Text));
```

Перед **else** в операторі **case** крапка з комою може ставитися, а може не ставитися.

Розглянемо ще один приклад використання оператора вибору для організації введення трьох чисел з використанням одного однорядкового редактора і однієї кнопки введення.

```
//Приклад 4.6
//Увести три коефіцієнти квадратного рівняння
//a*x*x+b*x+c=0 (a<>0). Вирішити це рівняння.
//Якщо корені відсутні, вивести повідомлення.
```

Створимо форму із властивістю Caption, що містить текст Введення трьох чисел. Розмістимо на формі багаторядкове поле виведення mmOutput1 з очищеною властивістю Lines, однорядковий редактор edInput1 із властивістю Text, що містить 0, і кнопку Button1 із властивістю Caption, що містить текст Введення. Над однорядковим полем введення помістимо мітку lb1, із властивістю Caption, що містить текст Уведіть a.

Крім того, в секції **implementation** модуля опишемо змінні a, b, c типу Real для запису коефіцієнтів квадратного рівняння і змінну key типу Byte. Останню змінну будемо використовувати для керування проце-

сом введення даних, а саме, в залежності від її значення (1, 2 або 3) у програмі вводитимуться коефіцієнти a, b або c відповідно. Початковим значенням змінної key повинне бути число 1.

Створимо для форми опрацьовувач події OnCreate:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DecimalSeparator := '.';
  key := 1;
  EdInput1.TabOrder := 0;
end;
```

Крім того, створимо також наступний опрацьовувач події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  d: Real;
begin
  case key of
  1: begin
      a := StrToFloat(edInput1.Text);
      lb1.Caption := 'Уведіть b';
     end;
  2: begin
      b := StrToFloat(edInput1.Text);
      lb1.Caption := 'Уведіть c';
     end;
  3: begin
      c := StrToFloat(edInput1.Text);
      lb1.Caption := 'Уведіть a';
     end;
  end;
  Inc(key);
  if key = 4 then begin          //Уведені всі три коефіцієнти
    key := 1;
    d := b * b - 4 * a * c;
    if d < 0 then
       mmOutput1.Lines.Add('Рішень немає')
    else begin
      mmOutput1.Lines.Add('x1=' +
                  FloatToStr((-b + Sqrt(d)) / (2 * a)));
      mmOutput1.Lines.Add('x2=' +
                  FloatToStr((-b - Sqrt(d)) / (2 * a)));
    end;
  end;
  edInput1.SetFocus;
end;
```

4.3. Оператор безумовного переходу

Призначення оператора переходу – порушувати природний порядок виконання операторів програми у разі виникнення яких-небудь особливих ситуацій. Він здійснює перехід до оператора, позначеного спеціальною міткою, що відокремлюється від самого оператора двокрапкою. Як ***мітка*** може бути використане будь-яке ціле число без знака, що містить не більш чотирьох цифр (тобто від 0 до 9999), або будь-який ідентифікатор.

Щоб можна було застосувати мітку, вона повинна бути в обов'язковому порядку оголошена в розділі міток в описовій частині програми. Цей розділ починається службовим словом **label**, після якого через кому перераховуються мітки:

```
label 2, 1234, label_1, 7777;
```

Щоб помітити оператор, необхідно перед ним указати мітку, відокремивши її від оператора двокрапкою (в оператора може бути кілька міток, які в цьому випадку також розділяються двокрапкою).

Для переходу до поміченого оператора призначений оператор переходу, що має вигляд

```
goto мітка;
```

Наприклад, передача керування може бути виконана оператором

```
goto label_1;
```

Мітки локалізуються усередині блоків (підпрограм), у яких вони описані, і не можуть бути описані в інтерфейсній частини модуля. Передати керування за допомогою мітки ззовні підпрограми усередину її не можна.

Оператор **goto** дозволяє здійснювати перехід у будь-яку точку програмного блоку – уперед або назад. Безладне використання цього оператора може привести до появи заплутаних й абсолютно непридатних для сприйняття програм. У зв'язку з цим останні два десятиліття висувалася така вимога: програма не повинна містити операторів **goto**. Ця вимога стала одним з принципів структурного програмування.

На зміну оператору **goto** прийшли оператори з дещо складнішою структурою, але з більшими можливостями, якими є оператори циклу. Однак у деяких випадках застосування оператора **goto** навпаки полегшує сприйняття тексту програми. Наприклад, при великій кількості рівнів вкладеності циклів (див. наступний підрозділ) обґрунтованим є використання оператора переходу з метою забезпечення примусового виходу з самого внутрішнього з них, оскільки у противному разі доводиться ускладнювати

як тіло циклів, так і умови їх продовження або припинення. Як і будь-який інший інструмент програмування, при правильному застосуванні оператор **goto** може виявитися досить корисним.

Отже, можна констатувати: наявність у програмі великої кількості операторів переходу свідчить про поганий стиль програмування.

4.4. Примусове припинення програми

Зазвичай програма завершує свою роботу з виконанням її останнього оператора (тобто при виході на **end** з крапкою). Якщо виникає необхідність у припиненні виконання програми де-небудь усередині неї, то можна скористатися процедурою без параметрів `Halt`.

Процедура `Halt` може бути викликана і з вказівкою одного параметра:

`Halt(`ціле_значення`).`

Параметром у цьому випадку є ціле значення від `0` до `255`. Це значення повертається в операційну систему як код помилки (ERRORLEVEL) і може бути проаналізоване нею у випадку запуску даної програми з командного файлу. Відсутність параметра в процедурі `Halt` відповідає його значенню `0`.

Формально виконання програми можна припинити за допомогою процедури без параметрів `Exit` при її розміщенні в проекті (а не в модулі). Але оскільки текст проекту програмістом звичайно не корегується, такий варіант залишається на рівні формальної можливості. Взагалі кажучи, ця процедура застосовується для виходу з даної підпрограми без припинення виконання визивальної підпрограми.

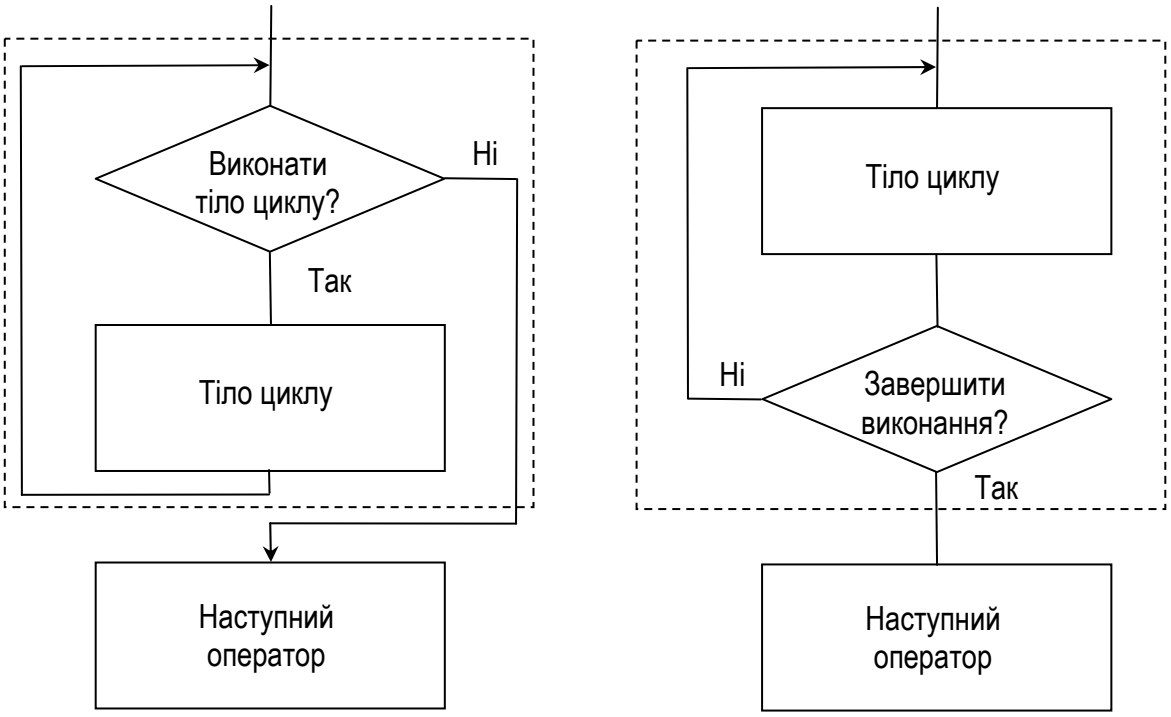
4.5. Цикли

4.5.1. Види циклів

Досить часто в програмі доводиться організовувати багаторазове повторення одних і тих самих операторів до виконання якої-небудь умови. Такі процеси називаються *циклічними*. Їх можна організувати з використанням операторів, що перевіряють умову, та оператора переходу. Однак у Delphi існують спеціальні оператори для організації циклів.

Загалом кажучи, існує два види циклів – цикл із передумовою і цикл із постумовою (див. рис. 4.2).

У першому випадку (рис. 4.2, *а*) спочатку проводиться перевірка деякої умови й, залежно від результату перевірки, або виконується, або пропускається сукупність операторів, що утворюють тіло циклу. Якщо тіло циклу виконане, то процес повторюється, починаючи з перевірки умови. У другому ж випадку (рис. 4.2, *б*) спочатку виконується тіло циклу, після чого здійснюється перевірка умови завершення циклу. Якщо умова не виконується, то процес повторюється.



а *б*

Рис. 4.2. Види циклів: *а* – цикл з передумовою; *б* – цикл з постумовою

Частковим випадком циклу із передумовою є цикл із параметром, у якому тіло циклу виконується для послідовного ряду значень деякого параметра, що змінюється автоматично.

4.5.2. Цикл із передумовою

Даний цикл у Delphi задається конструкцією вигляду

```
while умова do тіло_циклу;
```

Спочатку обчислюється значення умови (це будь-який вираз, що дає або `True`, або `False`). Якщо вона істинна, виконується тіло циклу, якщо хибна, – виконання циклу припиняється. Тіло циклу задається або одним простим оператором, або декількома операторами, укладеними в опера-

торні дужки **begin** – **end**. У тілі циклу повинне змінюватися значення хоча б однієї змінної, що входить в умову, інакше цикл буде нескінченним.

```
//Приклад 4.7
//Знайти суму елементів ряду 1, -0.5, 0.25, -0.125,...,
//що за модулем перевищують деяке додатне число.
```

Для розв'язання задачі скористаємося кінцевим варіантом форми з прикладу 3.4. У розділі **implementation** опишемо три дійсні змінні sum, epsilon, a. Для опрацьовувача події OnClick компонента Button1 створимо такий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  epsilon := StrToFloat(edInput1.Text);
  a := 1;
  sum := 0;
  while Abs(a) > epsilon do begin
    sum := sum + a;
    a := -a / 2
  end;
  mmOutput1.Lines.Add('Сума дорівнює ' + FloatToStr(sum));
  edInput1.Visible := False;
  lbOutput1.Caption := 'Дивися результат';
  Button1.Visible := False;
end;
```

Оскільки в процедурі використано цикл із передумовою, то при введенні значення epsilon, яке є більшим або дорівнює одиниці, цикл виконуватися не буде, що дасть значення суми, яка дорівнює нулю.

У наведеній вище процедурі TForm1.Button1Click відповідно до умови задачі не проводиться контроль додатності величини epsilon.

Розв'яжемо ще одну задачу.

```
//Приклад 4.8
//Відомо, що на відрізку [a, b] знаходиться єдиний
//корінь рівняння x+Sin(x)+f=0, де f – деяке дійсне
//значення. Знайти цей корінь методом ділення відрізка
//навпіл. Під наближеним значенням кореня розуміють будь-яку
//точку відрізка числової осі, що містить точне значення
//кореня й має довжину, що не перевищує задане додатне число.
```

Скористаємося кінцевим варіантом форми з прикладу 4.6, записавши у властивість Caption мітки lb1 текст Уведіть точність. У секції **implementation** модуля опишемо три змінні a, b, f, epsilon типу Real, а також змінну key типу Integer. Опрацьовувач події OnClick компонента Button1 замінимо на такий:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  buf, c: Real;
begin
  buf := StrToFloat(edInput1.Text);
  case key of
  1: begin
      epsilon := buf;
      lb1.Caption := 'Уведіть ліву межу відрізка';
     end;
  2: begin
      a := buf;
      lb1.Caption := 'Уведіть праву межу відрізка';
     end;
  3: begin
      b := buf;
      lb1.Caption := 'Уведіть вільний член f';
     end;
  4: begin
      f := buf;
      lb1.Caption := 'Дивись результат';
      while Abs(b - a) > epsilon do begin
        c := (b + a) / 2;                  //Середина відрізка
        if (a + Sin(a) + f) * (c + Sin(c) + f) <= 0
        then b := c    //Переміщення правої межі в середину...
     //... відрізка, якщо корінь ліворуч від середини відрізка
        else a := c;                 //Інакше зсуваємо ліву межу
      end;
      mmOutput1.Lines.Add('Корінь дорівнює ' + FloatToStr(c));
      Button1.Visible := False;
      edInput1.Visible := False;
     end
  end;
  Inc(key);
  if key <> 5 then
    edInput1.SetFocus;
end;
```

Опрацьовувач події OnCreate форми залишимо тим же, що й у прикладі 4.6.

Вираз, що визначає умову в операторі **while** може бути як завгодно складним. Це дозволяє записувати в конструкції його заголовка будь-який логічний вираз, що використовує логічні операції. Наприклад, якщо цикл повинен виконуватися доти, доки значення деякої змінної w не вийде за межі інтервалу [a, b], то його заголовок може мати такий вигляд:

```
while (w >= a) and (w <= b) do
```

або

```
while not ((w < a) or (w > b)) do
```

При некоректному записі циклу можлива ситуація, коли його виконання буде нескінченним. Це може статися або через неправильний запис умови, або через помилки, що призводять до неправильної зміни значень тих змінних, які входять в умову.

У деяких випадках спеціально організують нескінченний цикл (**while** `True` **do**). Вийти з такого циклу можна або за допомогою оператора **goto**, або за допомогою спеціального оператора виходу з циклу (оператор `Break`). В обох цих випадках у тілі циклу повинна бути передбачена перевірка деякої умови, згідно з якою здійснюється вихід. Найчастіше подібна ситуація свідчить про поганий стиль програмування (але іноді використання нескінченного циклу цілком виправдане).

Все сказане вище стосується і циклу з постумовою.

Якщо умова, записана в заголовку циклу **while**, відразу ж має значення `False`, то цикл не виконається жодного разу.

4.5.3. Цикл із постумовою

Цей вид циклу в Delphi задається конструкцією вигляду

```
repeat
  тіло_циклу
until умова;
```

Тут спочатку виконується тіло циклу (воно може складатися з декількох операторів, оскільки службові слова **repeat** та **until** обмежують тіло циклу з двох сторін), після чого обчислюється значення умови. Якщо воно дорівнює `False`, тіло циклу виконується повторно. Процес повторюється доти, доки значенням умови не стане `True`. Як і у циклі з передумовою, у тілі циклу **repeat** повинне змінюватися значення хоча б однієї змінної, що входить в умову.

```
//Приклад 4.9
//З яких цифр складається ціле число n.
```

Скористаємося формою з прикладу 2.2, змінивши значення властивості Caption мітки lbOutput1 на текст Уведіть ціле число. У секції **implementation** модуля опишемо змінну `n` типу `Integer`. Опрацьовувач події OnClick компонента Button1 замінимо на такий:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
```

```
  n := StrToInt(edInput1.Text);
  if n < 0 then n := -n;                  //Знищення знака числа n
  repeat          //Нижче остача від ділення на 10 – це  остання
    mmOutput1.Lines.Add(IntToStr(n mod 10));   //цифра числа n
    n := n div 10;           //Відкидаємо останню цифру числа n
  until n = 0;                             //Всі цифри розглянуті?
  Button1.Enabled := False;
end;          //Програма працює при -2147483647<=n<=2147483647
```

У коді опрацьовувача подій `TForm1.Button1Click` використано цикл із постумовою, оскільки хоча б один раз цикл повинен виконатися (будь-яке число складається хоча б з однієї цифри).

4.5.4. Цикл із параметром

У тих випадках, коли тіло циклу необхідно виконувати для послідовних значень деякої змінної якого-небудь дискретного типу, найчастіше застосовують оператор **for**, або, інакше, оператор циклу з параметром, що має один з наступних двох форматів:

```
for параметр := поч_знач to кінц_знач do тіло_циклу;
```

або

```
for параметр := поч_знач downto кінц_знач do тіло_циклу;
```

Тут параметр, або, інакше, індекс циклу, – це змінна будь-якого з порядкових типів (цілого, булівського, символьного, переліченого, відрізка), описана в блоці (підпрограмі), що містить цикл **for**, у якому використовується ця змінна; початкові та кінцеві значення – це вирази, сумісні за присвоюванням з параметром циклу; тіло циклу задається так само, як і в операторі **while**.

Цикл із параметром виконується в такий спосіб. Спочатку обчислюються початкове (поч_знач) і кінцеве (кінц_знач) значення параметра циклу. Початкове значення присвоюється параметру циклу, а кінцеве – запам'ятовується; внаслідок цього зміна усередині тіла циклу значення будь-якої змінної, яка входить до виразу для кінцевого значення, не позначається на кількості кроків циклу. Якщо запам'ятоване кінцеве значення більше або дорівнює поточному значенню параметра при **to** і менше або дорівнює при **downto**, то виконується тіло циклу, після чого параметр одержує наступне по порядку значення у випадку **to** і попереднє при **downto**. Процес повторюється, і востаннє цикл виконається при значенні параметра циклу, зазначеному після **to** (**downto**). Так, якщо параметр циклу має який-небудь цілий тип, то при виконанні циклу він автоматично

збільшується (при **to**) або зменшується (при **downto**) на 1. Оскільки перевірка умови закінчення циклу здійснюється до першого виконання тіла циклу (цикл із передумовою), цикл із параметром може не виконатися жодного разу.

По закінченні циклу значення параметра є невизначеним. Параметр циклу може змінювати своє значення тільки автоматично – примусове його змінювання у тілі циклу забороняє компілятор.

```
//Приклад 4.10
//Уводяться цілі числа до першого числа, меншого
//за два. Скільки простих чисел було уведено?
```

Скористаємося формою з прикладу 2.2, видаливши з неї мітку виведення та однорядкове редаговане текстове поле, а також змінивши значення властивості Caption компонента Button1 на текст Виконати. Опрацьовувач події OnClick компонента Button1 замінимо на такий:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  v: string;
  n, quantity: Integer;
  f: Boolean;                  //True - число просте, False - ні
begin
  Button1.Enabled := False;
  quantity := 0;
  v := '0';       //'0' - значення за умовчанням для InputQuery
  while InputQuery('Введення до числа меншого за 2',
          'Уведіть  ціле число', v) do begin  //Поки не Cancel
    n := StrToInt(v);
    if n < 2 then Break;                          //Вихід з циклу
    f := True;
    for  i := 2  to n div 2 do
      if n mod i = 0 then f := False;     //Якщо n ділиться на
           //одне з чисел 2, 3,..., n div 2, воно не є простим
    if f then quantity := quantity + 1;         //Якщо True, то
  end;                                           //число просте
  mmOutput1.Lines.Add('Кількість простих чисел: '
                      + IntToStr(quantity));
end;
```

У програмі цикл **for** використано для організації перебору можливих дільників уведеного числа `n` із записом значення `False` у булівську змінну `f` у випадку виявлення числа, на яке значення `n` ділиться без остачі (ознака того, що `n` не є простим числом).

Для організації введення чисел застосовано функцію `InputQuery` (див. підрозд. 3.17. Оскільки у випадку натискання в її віконці кнопки

Cancel функція `InputQuery` повертає значення `False`, у програмі забезпечена наявність додаткової умови припинення введення – натискання при введенні кнопки Cancel або клавіші Esc. Виконання ж основної умови забезпечує припинення циклу за допомогою оператора `Break` (див. наступний підрозділ).

У програмі вжито змінну `v`, що має тип **`string`**, який вказує на те, що значенням змінної є текстовий рядок (див. п. 5.2.1). Ця змінна використовується як третій параметр функції `InputQuery` і служить для прийому уведеного рядка (нагадаємо, що функція `InputQuery` уводить рядок символів, а не числове значення). Перед початком введення в змінну `v` записується рядок `'0'`, що є значенням функції `InputQuery` за умовчанням при першому її виконанні. Надалі значенням за умовчанням буде останнє значення, що вводилося.

Задавання значення за умовчанням перед початком введення забезпечить нормальну роботу функції `InputQuery` навіть у випадку, коли відразу ж при введенні першого значення буде натиснута кнопка Cancel (за відсутності списку введення).

Зауважимо, що серед операторів, що утворюють тіло циклу, у свою чергу, можуть зустрічатися оператори циклу, тобто можлива конструкція «цикл у циклі» («вкладений цикл»). Це стосується всіх видів операторів циклу мови Delphi. Так, у наведеній вище процедурі усередині циклу **`while`** використаний цикл **`for`**.

4.6. Оператори Break та Continue

Досить часто ціль виконання циклу досягається раніше, ніж він буде припинений за умовою виходу. Наприклад, в останній програмі внутрішній цикл буде виконуватися (`n` **`div`** `2 - 1`) разів, хоча те, що число не є простим, може бути виявлене на перших кроках циклу. Щоб зменшити кількість кроків циклу, потрібно або скористатися оператором **`goto`**, або сформувати складну умову виконання (припинення) циклу; наприклад, в останній програмі цикл **`for`** повинен бути замінений циклом

```
while (i <= n div 2) and flag do
```

У Delphi введено спеціальну процедура без параметрів `Break`, що перериває той цикл, до якого вона входить, незалежно від виконання умови припинення (продовження) циклу. Так, в останній програмі оператор `Break` використаний для припинення циклу при виконанні умови припинення введення. Крім того, у цій же програмі оператор **`if`**, що стоїть усередині оператора **`for`**, краще записати так:

```
    if n mod i = 0 then begin
      flag := True;
      Break
    end;
```

Це забезпечить припинення циклу **for** при першому ж діленні n на i без остачі.

Крім оператора Break, у Delphi введено процедуру без параметрів Continue, що, пропускаючи всі оператори, які стоять за нею, передає керування на кінець циклу, в якому вона міститься.

Оператори Break та Continue виконуються в кожному з видів циклів (**repeat**, **while**, **for**) і дійсні тільки для самого внутрішнього з тих циклів, до яких вони входять. Наприклад, якщо потрібно забезпечити примусовий вихід з подвійного циклу, оператор Break повинен бути розташований як у внутрішньому, так і в зовнішньому циклах. До певної міри оператори Break та Continue – це приховані оператори **goto**.

```
//Приклад 3.11
//Додати одне до одного два числа без використання
//операції додавання.
```

Створимо форму з двома однорядковими редакторами edInput1, edInput2 і кнопкою Button1 і напишемо для кнопки наступний опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, b,
  carry,                                    //розряд переносу
  Res: Integer;                                        //сума
begin
  a := StrToInt(edInput1.Text);
  b := StrToInt(edInput2.Text);
  while True do begin                       //Нескінченний цикл
   //При додаванні двох бітів, є біт результату і біт переносу
    res := a xor b;   //Біт результату 1, якщо додаються 1 й 0
     //або 0 й 1, інакше він дорівнює 0 (те ж що і бітова xor)
    carry := a and b;   //Біт переносу 1, якщо обидва біти =1;
       //інакше він дорівнює 0 (те ж що і бітова операція and)
    if carry=0 then Break;//Вихід з циклу, якщо немає переносу
    a := carry shl 1;   //Тепер перший доданок – біти переносу
    b:=Res;//після зсуву, а другий – результат додавання бітів
  end;
  mmOutput1.Lines.Add('Сума чисел ' + edInput1.Text + ' і '
    + edInput2.Text + ' дорівнює ' + IntToStr(Res));
  edInput1.SetFocus;
end;
```

Запитання для контролю і самоконтролю

1. Які оператори застосовуються для організації галуження?
2. У яких форматах може бути використаний оператор **if**?
3. Як встановлюється відповідність між службовими словами **else** і **then** в операторі **if**?
4. Яка різниця між **if** `n = 7` **then** та **if** `(n = 7)` **then**?
5. Коли доцільно застосовувати оператор **case**?
6. Як описуються і як використовуються мітки в програмі?
7. Що таке оператор переходу і коли його використання є виправданим?
8. Яке призначення оператора `Halt`?
9. У чому особливість циклів з перед- і постумовою?
10. Які оператори застосовуються для організації циклів?
11. Чим відрізняються оператори **while** та **repeat**?
12. Як здійснювати вибір між операторами **while** та **repeat**?
13. У чому особливість оператора циклу з параметром? Коли його використання є доцільним?
14. Чи можна організувати цикл **while** усередині циклу **for**?
15. Що відбувається при запуску нескінченного циклу?
16. Яке призначення процедур `Break` та `Continue`?

Завдання для практичного відпрацьовування матеріалу

1. У припущенні, що змінні `a` і `b` мають дійсний тип, напишіть фрагмент програми, що забезпечує присвоєння більшого значення змінній, яка має менше значення. Розглянути три варіанти: *а*) дозволено використовувати один оператор **if** з гілкою **else**; *б*) дозволено використовувати довільну кількість операторів **if** без гілки **else**.
2. Дано дійсні числа *a*, *b*, *c*. Чи можна побудувати трикутник, довжини сторін якого дорівнюють цим значенням?
3. Дано додатні дійсні числа *a* та *r*. Чи можна описати коло радіусом *r* навколо рівнобічного трикутника зі стороною *a*?
4. Два прямокутники зі сторонами, паралельними осям координат, задано координатами їх центрів і довжинами сторін. Обчислити площу їх перерізу.
5. Знайдіть помилку в наведеному нижче фрагменті програми:

```
var
  counter, sum: Integer;
```

```
        //Продовження тексту
sum := 0;
counter := 0;
while counter < 100 do begin
  Inc(sum, counter);
end;
```

6. Знайдіть помилку в наведеному нижче фрагменті програми.

```
var
  i: Integer;
        //Продовження тексту
for i := 0 to 100 do;
  Memo1.Lines.Add(IntToStr(i));
```

7. Знайдіть помилку в наведеному нижче фрагменті програми.

```
var
  counter, sum: Integer;
        //Продовження тексту
sum := 0;
counter := 50;
while counter < 10 do begin
  Inc(sum, counter);
  Dec(counter);
end;
```

8. Дано натуральне число n. Обчислити:

$а)$ $n!$; $б)$ $\sqrt{3+\sqrt{6+\cdots+\sqrt{3(n-1)+\sqrt{3n}}}}$;

$в)$ $\underbrace{\sqrt{2+\sqrt{2+\ldots+\sqrt{2}}}}_{n \text{ корней}}$; $г)$ $\left(1+\frac{1}{1^2}\right)\left(1+\frac{1}{2^2}\right)\ldots\left(1+\frac{1}{n^2}\right)$;

$д)$ $\frac{\cos 1}{\sin 1}+\frac{\cos 1+\cos 2}{\sin 1+\sin 2}+\ldots+\frac{\cos 1+\ldots+\cos n}{\sin 1+\ldots+\sin n}$.

9. Дано дійсне число x і натуральне число n. Обчислити:

$а)$ x^n; $б)$ $x(x+1)\ldots(x+n-1)$; $в)$ $x(x-n)(xa-2n)\ldots(x-n^2)$;

$г)$ $\frac{1}{x^2}+\frac{1}{x^2}+\frac{1}{x^4}+\ldots+\frac{1}{x^{2^n}}$; $д)$ $\frac{1}{x}+\frac{1}{x(x+1)}+\ldots+\frac{1}{x(x+1)\ldots(x+n)}$.

10. Дано натуральне число n та дійсне число x. Обчислити:

$а)$ $\sin x+\sin^2 x+\ldots+\sin^n x$; $б)$ $\sin x+\sin x^2+\ldots+\sin x^n$;

в) $\sin x + \sin\sin x + ... + \underbrace{\sin\sin...\sin}_{n} x$.

11. Дано дійсне число *a*. Знайти:

а) серед чисел 1, $1+\frac{1}{2}$, $1+\frac{1}{2}+\frac{1}{3}$, ... перше значення, що більш за *a*;

б) таке найменше *n*, що $1+\frac{1}{2}+...+\frac{1}{n}>a$.

12. Дано дійсні числа *a*, *h* і натуральне число *n*. Обчислити

$f(a)+2f(a+h)+2f(a+2h)+...+2f(a+(n-1)h)+f(a+nh)$, де $f(x)=(x^2+1)\cos^2 x$.

13. Дано ціле число *n*. *а*) Скільки цифр у числі *n*? *б*) Чому дорівнює сума його цифр? *в*) Знайти першу цифру числа *n*. *г*) Одержати нове число, приписавши по одиниці в початок та у кінець запису числа *n*.
14. Дано ціле число *n*. Знайти знакопереміжну суму цифр числа *n* (якщо запис *n* у десятковій системі є $\alpha_k\alpha_{k-1}\ldots\alpha_0$, то знайти $\alpha_k - \alpha_{k-1} + \ldots + (-1)^k\alpha_0$).
15. Дано натуральні числа *n* і *m*. Знайти: *а*) їх найбільший спільний дільник; *б*) їх найменше спільне кратне.
16. Нехай $\nu_1 = \nu_2 = 0;\ \nu_3 = 1.5;\ \nu_i = \frac{i+1}{i^2+1}\nu_{i-1} - \nu_{i-2}\nu_{i-3},\ i = 4,5,...$ Дане натуральне число *n* ($n \ge 4$). Одержати ν_n.
17. Нехай $x_0 = c;\ x_1 = d;\ x_k = qx_{k-1} + rx_{k-2} + b,\quad k = 2, 3, \ldots$ Дано дійсні числа *q*, *r*, *b*, *c*, *d* та натуральне число *n* ($n \ge 2$). Одержати x_n.
18. Нехай $u_1 = u_2 = 1;\ v_1 = v_2 = 1$, а для $i = 3, 4, \ldots$ значення u_i і v_i обчислюються по формулах $u_i = \frac{u_{i-1} - u_{i-2}v_{i-1} - v_{i-2}}{1 + u_{i-1}^2 + v_{i-1}^2};\quad v_i = \frac{u_{i-1} - v_{i-1}}{|u_{i-2} + v_{i-1}| + 2}$.

Дано натуральне число *n* ($n \ge 3$). Одержати v_n.

19. Нехай $a_1 = b_1 = 1;\ a_k = \frac{1}{2}\left(\sqrt{b_{k-1}} + \frac{1}{2}\sqrt{a_{k-1}}\right);\ b_k = 2a^{2k-1} + b^{k-1},\ k=2, 3,\ldots$

Дано натуральне число *n*. Знайти $\sum_{k=1}^{n} a_k b_k$.

20. Нехай $a_1 = b_1 = 1;\ a_k = 3b_{k-1} + 2a_{k-1};\ b_k = 2a_{k-1} + b_{k-1},\ k = 2, 3, \ldots$ Дано натуральне число *n*. Знайти

$$\sum_{k=1}^{n} \frac{2^k}{(1 + a_k^2 + b_k^2)k!}.$$

21. Обчислити нескінченну суму із заданою точністю ε (ε > 0). Вважати, що необхідна точність досягнута, якщо обчислено суму декількох перших доданків і черговий доданок виявився за модулем меншим, ніж ε (цей та всі наступні доданки можна вже не враховувати). Розглянути такі суми:

$$а)\ \sum_{i=1}^{\infty}\frac{(-1)^i}{i^2};\quad б)\ \sum_{i=1}^{\infty}\frac{(-2)^i}{i!};\quad в)\ \sum_{i=1}^{\infty}\frac{(-1)^{i+1}}{i(i+1)(i+2)};\quad г)\ \sum_{i=0}^{\infty}\frac{1}{4^i+5^{i+2}}.$$

22. Нехай $y_0 = 0;\ y_k = \dfrac{y_{k-1}+1}{y_{k-1}+2},\ k = 1, 2,...$ Дано дійсне значення ε > 0. Знайти перший член y_n, для якого виконане співвідношення $y_n - y_{n-1} < \varepsilon$.
23. Дано натуральне число n. Обчислити добуток перших n співмножників:

$$а)\ \frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot...;\quad б)\ \frac{1}{1}\cdot\frac{3}{2}\cdot\frac{5}{3}\cdot....$$

24. Дано натуральне число n. Одержати суму тих чисел виду $i^3 - \sin 2 + n$ ($i = 1, 2, ..., n$), які є потроєними непарними.
25. Дано натуральне число n, дійсне число x. Серед чисел $e^{\cos(x^{2k})}\sin(x^{3k})$ ($k = 1, ..., n$) знайти найближче до якого-небудь цілого.
26. Для $x_1 = y_1 = 1;\ x_i = x_{i-1} + \dfrac{y_{i-1}}{i^2},\ y_i = y_{i-1} + \dfrac{x_{i-1}}{i},\ i = 2,3,\ldots$ одержати x_{18}, y_{18}.
27. Дано натуральне число n. Одержати всі натуральні числа, що менші за n і не мають із ним спільних дільників, крім 1.
28. Дано натуральне число n. Одержати всі його прості дільники.
29. Дано натуральні числа m і n. Одержати $\dfrac{m!+n!}{(m+n)!}$.
30. Дано натуральне число m. Знайти всі прості дільники цього числа.
31. Дано натуральні числа m і n ($m \le n$). Знайти натуральне число між m і n з найбільшою сумою дільників.
32. Знайти 100 перших простих чисел.
33. Дано натуральні числа m і n ($m \le n$). Знайти всі прості числа між m і n.
34. Знайти найменше натуральне число n, що може бути подане двома різними способами як сума кубів двох натуральних чисел: $n = x^3 + y^3$ ($x \le y$).
35. Дано натуральне число n. Вирахувати $f_0 f_1 \ldots f_n$, де

$$f_i = \frac{1}{i^2+1} + \frac{1}{i^2+2} + \ldots + \frac{1}{i^2+i+1}.$$

36. Дано натуральне число n. Обчислити

$$а)\ \sum_{k=1}^{n} k(k+1)\ldots k^2; \quad б)\ \sum_{k=1}^{n}(-1)^{k+1}(3k^2-1)!; \quad в)\ \sum_{k=1}^{n} k^k; \quad г)\ \sum_{k=1}^{n}\frac{1}{(k^2)!}.$$

37. Дано дійсне число x і натуральне число n. Обчислити

$$а)\ \sum_{i=1}^{n}\frac{(2i)!+|x|}{(i^2)!}; \quad б)\ \sum_{p=1}^{n}\sum_{q=p}^{n}\frac{x+p}{q}(p+q)!; \quad в)\ \frac{1}{n!}\sum_{k=1}^{n}(-1)^k\frac{x^k}{(k!+1)!}.$$

38. Припустімо, Ви хочете дістати позику на k місяців у розмірі n, грн. Умови кредиту такі: процентна ставка становить p, %, на рік від загальної суми кредиту, причому вся сума процентних виплат відраховується із загальної суми кредиту відразу ж у момент його отримання. Погашення позики здійснюється щомісяця рівними частками, виходячи з номінальної вартості кредиту. Яка номінальна сума позички повинна бути нарахована і яка сума повинна погашатися щомісяця? Так, при $n = 775$, $k = 18$, $p = 15$ номінальна сума кредиту становитиме $m = \frac{p}{100}\cdot n\cdot\frac{k}{12} = 1000$ грн, виплата при одержанні кредиту дорівнює $m\cdot p\cdot\frac{k}{12} = 225$ грн, а щомісячні виплати будуть $\frac{1000}{18} = 55{,}56$ грн. Для обчислення номінальної вартості кредиту використайте цикл замість наведеної вище формули. Варто врахувати те, що можлива ситуація, коли початкова виплата дорівнюватиме або перевищуватиме номінальну вартість кредиту.
39. Розв’язати задачу 38 при уточненні: якщо на запропонованих умовах надання позики неможливе, повинен бути знайдений найбільший можливий термін кредитування.
40. Розв’язати задачу 38, уточнивши її у такий спосіб: якщо на запропонованих умовах надання позики неможливе, повинна бути знайдена найбільша можлива річна процентна ставка.
41. Нехай деяка покупка вартістю n, грн., зроблена в кредит із процентною ставкою p, %, на рік і відсутністю початкового внеску. Виплати проводяться щомісяця фіксованою сумою в m, грн, до якої входить виплата відсотка за кредит у даному місяці, причому щомісячні відсотки за кредит нараховуються, виходячи з річної процентної ставки та залишку боргу на момент виплати (сума виплати в останній місяць може виявитися меншою за m). Скільки місяців буде потрібно для погашення боргу. Яку суму становитиме плата за кредит? Урахувати те, що щомісячна виплата повинна перевищувати оплату відсотків за кредит.

5. СКЛАДЕНІ ТИПИ

Раніше у підрозд. 3.5 говорилося про систему типів Delphi. Основні скалярні типи було розглянуто у підрозд. 3.13. Перейдемо до розгляду складених типів, до яких належать масиви, рядки, множини та записи.

5.1. Масиви

5.1.1. Одновимірні масиви

У Delphi існує механізм, що дуже просто розв'язує проблему позначення великої кількості однорідних елементів. При цьому можна не обмежуватися однією змінною, у яку по черзі заноситься множина значень, або оголошувати велику кількість різнойменних змінних. Альтернативою є використання ***масиву*** – змінної типу **array**, яка містить визначене число однойменних елементів одного й того самого типу і дозволяє посилатися на перший, другий і всі наступні елементи за значенням їх індексів.

Для опису так званого одновимірного масиву потрібно, крім його імені, зазначити межі зміни значення індексу та тип елементів, що є спільним для всіх елементів масиву:

```
var
  ім'я:array [ниж_межа_інд.. верх_межа_інд] of тип_елементів;
```

При цьому діапазон зміни індексу не може перевищувати 2 Гбайт (у випадку багатовимірних масивів це стосується кожного з вимірів). Таке ж саме обмеження має місце і для загального обсягу пам'яті, займаної масивом.

Наприклад:

```
var
  a, b: array[-10..100] of Integer;
  c   : array['a'..'z'] of Boolean;
```

Для звертання до елемента масиву вказують ім'я масиву й у квадратних дужках індекс, значення якого визначає розташування елемента

усередині масиву. Найчастіше використовуються цілочислові індекси, але в загальному випадку індексами масиву можуть бути дані будь-якого дискретного типу, тобто такого, у якому, по-перше, визначено дискретну послідовність значень і, по-друге, кожне з цих значень можна перерахувати по порядку. Індексувати можна як константами і змінними, так і виразами, результат обчислення яких дає значення дискретного типу.

Коли індекс масиву може набувати всіх можливих значень деякого дискретного типу, тоді при описі масиву доцільно задавати ім'я типу замість меж зміни індексу (при цьому межами індексу будуть перше та останнє значення в описі типу індексу).

Межі зміни індексів можуть задаватися з використанням раніше оголошених констант.

Рекомендується попередньо оголошувати масивний тип у розділі опису типів, а потім вживати його для опису масивів:

```
const
  Num = 150;
type
  DaysOfWeek = (Monday, Tuesday, Wednesday, Thursday,
      Friday, Saturday, Sunday);     //Перелічений тип
  Years = 1991..2010;                        //Відрізок
  TTax = array[Years] of Real;             //Тип-масив
var                          //Можливі методи опису масиву
  Gain: array[DaysOfWeek] of Real;
  TaxOfYear: TTax;
  Production: array[1..Num] of string[10];
```

У наведеному прикладі тип `TTax` визначає масив, індекси якого можуть набувати значень від `1991` до `2010`. За допомогою цього типу оголошено масив `TaxOfYear`. Для індексування елементів масиву `Gain` має використовуватися значення переліченого типу `DaysOfWeek`, причому сам масив має 7 елементів відповідно до опису типу `DaysOfWeek`.

Функція `SizeOf`, застосована до імені масиву або імені масивного типу, повертає кількість байт, що відводиться під масив. Так для наведених вище описів, якщо змінна `Size` має тип `Integer`, то після виконання, як оператора

```
Size := SizeOf(TaxOfYear),
```

так і оператора

```
Size := SizeOf(TTax),
```

вона матиме значення `160`.

Функції `Low`, `High` і `Length`, застосовані до імені масиву, повертають відповідно мінімальне значення індексу, максимальне значення індексу й кількість елементів масиву. Для описаного вище масиву `TaxOfYear` згадані функції дадуть такі результати:

`Low(TaxOfYear) = 1991,`
`High(TaxOfYear) = 2010,`
`Length(TaxOfYear) = 20.`

При описі масиву і звертанні до його елементів квадратні дужки можуть бути замінені парою складених символів `(.` та `.)`.

```
//Приклад 5.1
//Які різні цифри входять у ціле число N?
```

Скористаємося формою з прикладу 2.2, записавши у властивість Caption мітки текст Уведіть ціле число, і створимо для компонента Button1 такий опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  N: Int64;
  Numerals: array [1..19] of Byte;              //Різні цифри
  i,                                          //Параметр циклу for
  QuantityOfNumerals,                       //Кількість різних цифр
  Numeral: Byte;                              //Виділювана цифра
  Flag: Boolean;
begin
  N := StrToInt64(edInput1.Text);
  QuantityOfNumerals := 0;        //Поки що немає жодної цифри
  repeat
    Numeral := Abs(N mod 10);           //Одержуємо нову цифру
    flag := False;   //Вважаємо, що цифра раніше не виділялася
                //Перевіряємо, чи виділялася раніше така цифра
  for i := 1 to QuantityOfNumerals do
    if Numeral = Numerals[i] then begin
      Flag := True; Break;        //Раніше така цифра вже була
    end;
    if Flag = False then begin        //Краще так: if not Flag
      QuantityOfNumerals := QuantityOfNumerals + 1;
      Numerals[QuantityOfNumerals] := Numeral;
    end;
    N := N div 10    //Відтинаємо останню цифру в запису числа
  until N = 0;
  mmOutput1.Lines.Add('Число ' + edInput1.Text +
                      ' містить такі різні цифри:');
  for i := 1 to QuantityOfNumerals do
    mmOutput1.Lines.Add(IntToStr(Numerals[i]));
  edInput1.Visible := False;
```

```
  lbOutput1.Caption := 'Дивись результат';
  Button1.Visible := False;
end;
```

Метод розв'язання задачі полягає в багаторазовому діленні уведеного числа N на 10 з одержанням як нового значення змінної N цілої частини від ділення (при N=0 обчислення припиняються). Як цифра, що входить у запис числа, береться модуль остачі від ділення його на 10 (змінна Numeral). Модуль остачі від ділення заноситься в масив Numerals з попередньою перевіркою в циклі **for** наявності цього значення в масиві (з метою виключення повторення значень у масиві). Перед занесенням нового елемента в масив збільшується кількість різних цифр (змінна QuantityOfNumerals). Оскільки в запис цілого числа завжди входить хоча б одна цифра, для виділення цифр обрано цикл **repeat**.

Зауважимо, що в програмі цифри виділяються, починаючи від самої правої з них у запису числа. Аналогічно вони і виводяться. Щоб змінити порядок виведення на протилежний, останній цикл **for** потрібно записати в такий спосіб:

```
for i := QuantityOfNumerals downto 1 do
  mmOutput1.Lines.Add(IntToStr(Numerals[i]));
```

Щоб описати типізовану константу – одновимірний масив, потрібно в круглих дужках подати через кому значення всіх її елементів. При цьому тип масиву краще оголошувати в розділі опису типів:

```
type
  T_Flags = array[1..5] of Boolean;
const
  Flags: T_Flags = (True, False, True, True, False);
```

```
//Приклад 5.2
//У які дні прибуток був вищим за середній прибуток за
//тиждень?
```

Скористаємося формою з прикладу 2.2, записавши у властивість Caption мітки текст Уведіть прибуток за Понеділок. Включимо також в опис класу TForm1 у розділ **public** опис двох полів:

```
Gain: array[DaysOfWeek] of Real;  //Прибуток по днях тижня
d: DaysOfWeek;                                  //День тижня
```

При цьому **перед** описом класу TForm1 у розділі опису типів секції **interface** опишемо такий тип:

```
DaysOfWeek = (Monday, Tuesday, Wednesday,
              Thursday, Friday,
              Saturday, Sunday);
```

Щоб забезпечити можливість виведення українських назв днів тижня, визначимо одновимірний масив `ukr_days` як типізовану константу:

```
  const
    ukr_days: array[DaysOfWeek] of string =
       ('Понеділок', 'Вівторок', 'Середа', 'Четвер',
        'П''ятниця', 'Субота', 'Неділя');
```

Для розв'язання задачі скористаємося таким опрацьовувачем події OnClick компонента Button1, що є модифікацією аналогічного опрацьовувача події OnClick з прикладу 2.2:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Sum: Real;
  i: DaysOfWeek;
begin
  Gain[d] := StrToFloat(edInput1.Text);
  if d = Sunday then begin
    Sum := 0;
    for i := Monday to Sunday do
      Sum := Sum + Gain[i];
    mmOutput1.Lines.Add(
            'Вище за середній прибуток був у такі дні:');
    for i := Monday to Sunday do
      if Gain[i] > Sum / 7 then
        mmOutput1.Lines.Add(ukr_days[i]);
    Button1.Visible := False;
    edInput1.Visible := False;
    lbOutput1.Visible := False;
  end
  else begin
    Inc(d);                    //Перехід до наступного дня тижня
    lbOutput1.Caption:='Уведіть прибуток за ' + ukr_days[d];
    edInput1.SetFocus;
  end;
end;
```

Опрацьовувач події OnCreate форми з прикладу 2.2 також дещо зміниmo:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DecimalSeparator := '.';
  d := Monday;
end;
```

У прикладі 5.2 діапазон змінення індексів у масивах `ukr_days` і `Gain` задається типом перерахування `DaysOfWeek`. Природно,

індексування елементів цих масивів також здійснюється значеннями типу DaysOfWeek. Введення числових значень супроводжується збільшенням значення змінної d. При виконанні рівності d=Sunday введення припиняється з переходом до безпосереднього розв'язку задачі. Помітимо, що в процедурі TForm1.Button1Click організовані два цикли по змінним переліченого типу.

За умовчанням у Delphi при звертанні до елементів масиву не контролюється вихід значення індексу за діапазон, зазначений в описі масиву. Це є джерелом помилок, зумовлених можливістю запису даних за межі масиву.

Наприклад, нехай у програмі масив myArray описаний так:

```
var
  myArray: array[1..10] of Integer;
```

Якщо змінна Ind має тип Integer, то наступні два оператори

```
Ind := -1;
myArray[Ind] := 25;
```

запишуть значення 25 у деяку комірку пам'яті, що знаходиться поза масивом. Результат роботи такої програми не передбачуваний. Добре, якщо помилка виявиться відразу ж, і програма припинить виконуватися. У гіршому ж випадку виконання програми продовжиться, й остаточний результат її роботи буде зовсім несподіваним.

Поява такого роду помилок спричинена тим, що компілятор будує програмний код, який реалізує звертання до елемента масиву, сліпо обчислюючи його зсув щодо адреси першого елемента. Можливість забезпечення автоматичного контролю значень індексу розглянута в п. 5.1.4.

5.1.2. Динамічні масиви

У Delphi (починаючи з версії 4) є можливість описувати не тільки так звані статичні масиви (з діапазоном зміни індексів, що задається константами), але й динамічні масиви, розміри яких задаються під час виконання програми. Ця можливість існувала й у мові Turbo Pascal (Object Pascal), але для цього потрібно було в програмі організовувати роботу з вказівниками. У Delphi це здійснюється значно простіше. Формат опису одновимірного динамічного масиву такий:

```
var
  ім'я: array of тип_елементів;
```

Отже, при описі динамічного масиву кількість елементів ніяк не задається, – задається тільки їх тип, наприклад:

```
var
  a: array of Real;
```

Щоб такий масив можна було використовувати в програмі, необхідно організувати відведення пам'яті під нього (інакше – здійснити ***ініціалізацію*** масиву). Для цього повинна бути виконана функція відведення пам'яті `SetLength`, що має два параметри, першим з яких є ім'я масиву, а другий задає кількість його елементів (не обов'язково константа – частіше вираз, що містить змінні):

```
SetLength(a, 100);
```

При цьому задається не діапазон зміни індексів, а кількість елементів, оскільки нижнє значення індексу динамічного масиву завжди дорівнює 0 (отже, максимальне значення індексу дорівнює кількості елементів мінус 1). Таким чином, наведений вище оператор створює масив зі 100 елементів з номерами від 0 до 99.

При браку пам'яті для ініціалізації динамічного масиву виникає виняток `EOutOfMemory` (див. розд. 13).

Якщо в ході виконання програми необхідність у використанні динамічного масиву, пам'ять під який була відведена раніше, відпадає, він може бути знищений, для чого вживають процедуру `Finalize` (звичайно з одним параметром – ім'ям масиву):

```
Finalize(a);
```

Оскільки фактично ім'я масиву є вказівником (відповідні питання будуть розглядатися у розд. 7), більш простим способом є присвоєння константи **`nil`** імені масиву:

```
a := nil;
```

Функція `SizeOf`, застосована до імені динамічного масиву, повертає не обсяг пам'яті, займаної цим масивом, а обсяг пам'яті, займаної змінною, що є ім'ям масиву (а це вказівник).

```
//Приклад 5.3
//Упорядкувати числовий масив за неубуванням
//методом "бульбашки".
```

Скористаємося формою з прикладу 2.2, записавши у властивість Caption мітки текст Уведіть кількість елементів масиву, а у властивість Text однорядкового редагованого текстового поля число 1. Додамо також дві кнопки Button2 і Button3, записавши в їхні властивості Visible значення False. У властивості Caption компонентів Button1 і Button2 запишемо текст Увести, а у властивість Caption компонента Button3 – текст Сортувати. Включимо також в опис класу `TForm1` у розділ **`public`** опис двох полів:

```
    N: Integer;                  //Кількість елементів масиву
    a: array of Real;                                  //Масив
```

Для компонентів Button1, Button2 і Button3 створимо такі опрацьовувачі події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DecimalSeparator := '.';
  N := StrToInt(edInput1.Text);          //Кількість елементів
  mmOutput1.Lines.Add('N=' + IntToStr(N));
  SetLength(a, N);             //Створюємо масив з N елементів
  lbOutput1.Caption:='Для введення масиву натисніть "Ввести"';
  edInput1.Visible := False;         //Однорядковий редактор і
  Button1.Visible := False;   //кнопка Button1 тепер невидимі,
  Button2.Visible := True;     //а компонент Button2 – видимий
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
begin
  lbOutput1.Visible := False;                //Мітка і кнопка
  Button2.Visible := False;           //Button2 тепер невидимі
  for i := 0 to N - 1 do
    a[i] := StrToInt(InputBox('Введення масиву',
              'Уведіть елемент a[' + IntToStr(i) + ']', '0'));
  mmOutput1.Lines.Add('Початковий масив');      //Повідомлення
  for i := 0 to N - 1 do             //Виводимо початкові дані
    mmOutput1.Lines.Add('a[' + IntToStr(i) + ']= ' +
                        FloatToStr(a[i]));
  Button3.Visible := True;    //Кнопка Button2 тепер є видимою
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  i, j: Integer;
  b: Real;
begin                                        //Метод "бульбашки"
  for i := 0 to N - 2 do
    for j := N - 1 downto i + 1 do
      if a[j] < a[j - 1] then begin
        b := a[j];
        a[j] := a[j - 1];
        a[j - 1] := b;
      end;
  mmOutput1.Lines.Add('Результуючий масив');
  for i := 0 to N - 1 do
    mmOutput1.Lines.Add('a[' + IntToStr(i) + ']=' +
```

```
                        FloatToStr(a[i]));
  Button3.Visible := False;
end;
```

В опрацьовувачі події OnClick компонента Button1 здійснюється введення розміру динамічного масиву `a` і його створення за допомогою функції `SetLength`. Елементи масиву вводяться за допомогою функції `InputBox`. Оскільки по закінченні роботи функції `InputBox` вихід з підпрограми `TForm1.Button2Click` не виконуються, здійснюється багаторазове звертання до цієї функції в циклі **`for`**, що дозволяє істотно спростити введення вмісту масиву, порівняно з методикою введення масиву, що реалізована в прикладі 5.2.

Розглянемо метод «бульбашки», реалізований в опрацьовувачі події OnClick компонента Button3. Він полягає в наступному. Розглядається останній елемент («бульбашка»), і, якщо він менший («легший») за попередній, вони переставляються, після чого, незалежно від того, була або не була перестановка, проводиться перехід до попереднього елемента (номер розглянутого елемента зменшується на 1). Процес повторюється до порівняння другого і першого елементів. У результаті найменший («найлегший») елемент займе першу позицію в масиві. На другому кроці описані дії повторюються до моменту порівняння третього і другого елементів (перший елемент розглядати не потрібно, бо він уже найменший). Таким чином, з кожним кроком кількість розглянутих елементів зменшується на 1, і всього потрібно виконати `N-1` кроків (поки не будуть розглянуті тільки останні два з `N` елементів).

5.1.3. Багатовимірні масиви

Ми розглядали одновимірний масив. Але дуже часто структуру даних зручно подати у вигляді прямокутної таблиці. Щоб звернутися до елемента такої структури, потрібно вказати дві координати – по горизонталі і по вертикалі. Для подання таких структур у Delphi використовують двовимірні масиви.

Відмінність в описі двовимірного масиву від одновимірного полягає тільки в тому, що необхідно зазначити верхню і нижню межі другого індексу. При цьому перший індекс служить для звертання до рядків, а другий – до стовпців масиву. Крім звичайного способу опису двовимірного масиву, можливий також розгляд його при описі як масиву масивів. Наприклад, такі два описи є ідентичними:

```
var
  eval: array[1..100, 1..4] of Byte;
```

```
var
  eval: array[1..100] of array[1..4] of Byte;
```

При використанні другого варіанта опису краще спочатку визначити деякий тип одновимірного масиву (рядок двовимірного масиву), що потім буде використаний при описі двовимірного масиву:

```
type
  exam = array[1..4] of Byte;
var
  eval: array[1..100] of exam;
```

Для звертання до елемента двовимірного масиву необхідно зазначити ім'я масиву й у квадратних дужках через кому – значення двох індексів: перший указує номер рядка, а другий – номер стовпця, на перетинанні яких розташовано елемент (`eval[i, 4] := 5`). Можна застосовувати змішане індексування (наприклад, перший індекс має цілий тип, а другий – символьний). Індекси можуть укладатися кожний у свої квадратні дужки без відділення комою (`eval[1][4] := 5`).

Відзначимо, що якщо типи двох масивів (поза залежністю від кількості вимірів) еквівалентні, в Delphi дозволене присвоювання одного масиву іншому.

Наприклад, нехай мають місце такі описи:

```
type T1 = array[1..4] of Real;
var
  v, w: array[1..100] of T1;
  z: T1;
```

Тоді оператор `v := w` переписує всі елементи масиву `w` у відповідні елементи масиву `v`.

Якщо при звертанні до двовимірного масиву зазначено тільки один індекс, то отримана конструкція є вказівником на рядок, номер якого визначається значенням індексу (тобто це «ідентифікатор» одновимірного масиву, що є рядком двовимірного масиву). Тому для наведених вище описів допустимими є й такі оператори:

```
v[5] := w[4];              //Переписування рядка у рядок
z := w[13];          //Переписування рядка двовимірного
v[10] := z;     //масиву в одновимірний масив і навпаки
```

```
//Приклад 5.4
//Дано квадратний масив чисел розміром не більше 100x100.
//Чи правда, що мінімальний з елементів, розташованих
//над головною діагоналлю і на ній, перебуває правіше
//і нижче від максимального елемента масиву? Головна
//діагональ прямує з лівого верхнього кута в правий нижній.
```

Скористаємося формою з прикладу 2.2, записавши у властивість Caption мітки текст Уведіть розмірність масиву (до 100). Включимо також в опис класу TForm1 у розділ **public** опис двох полів:

```
    a: array[1..100, 1..100] of Real;                    //Масив
    n: Integer;                                  //Розмірність масиву
```

Для розв'язання задачі скористаємося таким опрацьовувачем події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i, j: Integer;
  StrMax, ColMax,     //Номери рядків і стовпців максимального
  StrMin, ColMin: Integer;          //і мінімального елементів
  s: string;
begin
  n := StrToInt(edInput1.Text);   //Задаємо розмірність масиву
  Randomize;
  for i := 1 to n do              //Елементи масиву – випадкові
    for j := 1 to n do         //дійсні числа від -50.0 до 50.0
      a[i, j] := (Random - 0.5) * 100;
  mmOutput1.Lines.Add('Початкові дані:');
  mmOutput1.Lines.Add('n = ' + IntToStr(n));
  for i := 1 to n do begin             //Виводимо рядками масив
    s := '';
    for j := 1 to n do//Елементи рядка розділяємо символом та-
      s:=s+FloatToStrF(a[i, j], ffFixed, 10, 3) + #9;//буляції
    mmOutput1.Lines.Add(s);
  end;
  StrMax := 1; ColMax := 1;   //Вважаємо, що шукані елементи -
  StrMin := 1; ColMin := 1;   //у лівому верхньому куті масиву
  for i := 1 to n do    //У подвійному циклі визначаємо дійсне
    for j := 1 to n do begin      //місце розташування шуканих
      if a[i, j] > a[StrMax, ColMax] then begin    //елементів
        StrMax := i; ColMax := j;
      end;
      if (i <= j) and (a[i, j]< a[StrMin, ColMin]) then begin
        StrMin := i;
        ColMin := j;
      end;
    end;
  if (ColMin > ColMax) and (StrMin > StrMax)
    then mmOutput1.Lines.Add('Правда')
    else mmOutput1.Lines.Add('Неправда');
  Button1.Visible := False;
  edInput1.Visible := False;
  lbOutput1.Visible := False;
end;
```

Оскільки програма супроводжується коментарями, зупинимося тільки на фрагменті, призначеному для виведення вмісту двовимірного масиву. Вміст масиву виводиться в подвійному циклі, причому внутрішній цикл служить для формування текстового рядка для виведення. Цей рядок містить елементи рядка масиву. Рядок, що буде виводитись, формується у внутрішньому циклі шляхом зчеплення рядків, одержуваних перетворенням елементів рядка масиву до рядкового виду. Щоб елементи масиву розташовувалися в окремих колонках, між ними вставляється символ табуляції, який має десятковий код 9. Зазначимо також, що для перетворення дійсних чисел до рядкового подання застосовано процедуру `FloatToStrF`, яка дозволяє одержувати різне рядкове подання дійсних чисел. У даному випадку указано, що використовується формат запису з фіксованою точкою (другий параметр дорівнює `ffFixed`) з не більш ніж 10 цифрами в запису числа (третій параметр), з них три – у дробовій частині (четвертий параметр).

Масиви можуть мати розмірність більшу двох. При цьому в пам'яті вони розташовуються так, що останній індекс змінюється швидше за всі попередні, а перший – повільніше за всі.

Якщо описується двовимірний масив як типізована константа, то при задаванні значень його елементів він розглядається як масив масивів. При цьому в спільних круглих дужках перераховуються укладені в круглі дужки значення елементів рядків (кожен рядок у своїх дужках):

```
type T2Arr = array[-1..2, 'A'..'C'] of Real;
const
  g: T2Arr = ((-1.1, -1.2, -1.3), (0.1, 0.2, 0.3),
              (1.1, 1.2, 1.3), (2.1, 2.2, 2.3));
```

Аналогічно описуються типізовані константи – багатовимірні масиви (зовнішні дужки відповідають першому індексу, найбільшого рівня вкладеності – останньому):

```
const
  v: array[1..3, 1..4, 1..2] of Real =
   (((1.11,1.12),(1.21,1.22),(1.31,1.32),(1.41,1.42)),
   ((2.11,2.12),(2.21,2.22),(2.31,2.32),(2.41,2.42)),
   ((3.11,3.12),(3.21,3.22),(3.31,3.32),(3.41,3.42)));
```

Як і одновимірні масиви, багатовимірні масиви можуть бути динамічними, для чого при описі необхідно оголошувати масив масивів. Наприклад, наступний оператор оголошує тривимірний динамічний масив булівських значень:

```
var
  b: array of array of array of Boolean;
```

При ініціалізації такого масиву в процедурі вказують більше двох параметрів, причому параметри, починаючи з другого, задають розміри масиву за кожним з вимірів. Як ці параметри можуть виступати вирази, що містять змінні.

Природно, при ініціалізації багатовимірних масивів можна вдаватися до функції `SetLength` з двома параметрами, установлюючи довжини кожного з вимірів окремо. Наприклад, якщо динамічний масив `r` описаний за допомогою оператора

```
var r: array of array Real;
```

то двовимірний масив дійсних чисел розміру 2×7 можна створити за допомогою таких операторів:

```
SetLength(r, 2); //2 – кількість рядків (перший вимір)
SetLength(r[0], 7);//7 - кількість елементів у рядку 0
SetLength(r[1], 7);//7 - кількість елементів у рядку 1
```

При цьому `r[0]` і `r[1]` – це рядки двовимірного масиву.

Довжини рядків можна задати не обов'язково однаковими, що дозволяє імітувати масиви з рядками різної довжини. Звернемо також увагу на те, що нумерація елементів за всіма вимірами здійснюється від нуля.

```
//Приклад 5.5
//Створити трикутний масив дійсних чисел.
```

Скористаємося формою з прикладу 5.4, записавши у властивість Caption мітки текст Уведіть кількість рядків масиву, і включимо в опис класу `TForm1` у розділ **`public`** опису поля `a` типу **`array of array of`** `Real` і поля `n` типу `Integer`. Створимо також такий опрацьовувач події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i, j: Integer;
  s: string;
begin
  n := StrToInt(edInput1.Text);     //Задаємо кількість рядків
  Randomize;
  SetLength(a, n);       //Установлюємо довжину першого виміру
  for i := 0 to n - 1 do begin
    SetLength(a[i], i + 1);              //Створюємо і-й рядок
    for j := 0 to i do
      a[i, j] := (Random - 0.5) * 100;
  end;
  mmOutput1.Lines.Add('Трикутний масив');
  for i := 0 to n - 1 do begin        //Виводимо рядками масив
    s := '';
```

```
    for j := 0 to i do
      s := s + FloatToStrF(a[i, j], ffFixed, 10, 3) + #9;
    mmOutput1.Lines.Add(s);
  end;
  Button1.Visible := False;
  edInput1.Visible := False; lbOutput1.Visible := False;
end;
```

Функції `Low`, `High` та `Length`, застосовані до імені багатовимірного масиву, повертають відповідно мінімальне значення першого індексу, максимальне значення першого індексу і кількість елементів масиву за першим виміром.

5.1.4. Автоматичний контроль значень індексів

Однією з послуг, яку надає компілятор Delphi, є автоматичний контроль значень індексів під час виконання програми. Для цього достатньо активізувати директиву компілятору `$R`, що дає розпорядження компілятору здійснювати убудований контроль граничних значень змінних. Найчастіше ця директива виключена, що зменшує обсяг програми і час обчислень. При активізації цієї директиви (`{$R+}`) усяка спроба використати індексне значення поза оголошеними межами буде зареєстрована як помилка часу виконання програми, оскільки в цьому випадку забезпечується включення контролю виходу значень порядкових змінних за діапазон відповідного типу. Наприклад, щоб у якому-небудь фрагменті програми здійснювався контроль діапазону, потрібно обмежити цей фрагмент директивою `$R`:

```
{$R+}
Фрагмент програми
{$R-}
```

Як це вже зазначалося при розгляді цілих типів, для активізації директиви `$R` можна також в ICP Delphi виконати команду Project ► Options (Проект ► Опції) й установити прапорець Range Checking (Перевірка області) на вкладці Compiler (Компілятор) діалогового вікна, що при цьому відкриється.

5.2. Робота з рядками

5.2.1. Поняття рядкових змінних. Присвоювання рядків

Як про це говорилося раніше, в Delphi визначено тип `Char`, значеннями якого є символи (літери, цифри, різні знаки). Часто ж потрібно

працювати не з окремими символами, а з їх рядками (наприклад, прізвище). У Delphi визначено кілька типів для опису змінних, у яких можуть зберігатися рядки символів:

`AnsiString` або **`string`** – довгі рядки;

`ShortString` або **`string`**`[N]` – короткі рядки;

`WideString` – широкі рядки.

Крім того, для зберігання рядкових даних використовують ще два типи – тип `PChar` і масив символів (**`array of`** `Char`) з нижньою межею індексу, що дорівнює 0 (символьний масив з нульовою базою).

Будь-який рядок може трактуватися як масив символів. При цьому короткі рядки можна розглядати як масив з кількістю елементів, зумовленою довжиною рядка, а інші види рядків – як масиви символів, що обмежуються першою появою символу `#0` (у рядків типів `AnsiString` та **`string`** наявні обидва обмеження на кількість елементів).

Оголошення рядкових змінних здійснюється звичайним способом. Наприклад:

```
var
  StrAnsi : AnsiString;
  Str     : string;
  StrShort: ShortString;
  Str6    : string[6];
  StrWide : WideString;
  StrPCh  : PChar;
  StrArr  : array [0..1000] of Char;
```

Кращим для більшості цілей є тип `AnsiString`.

У виразах рядки різних типів можуть зустрічатися спільно. Перетворення типів при цьому відбувається автоматично. Однак рядки, що підставляються замість **`var`**- та **`out`**-параметрів при звертанні до процедур і функцій (див. розд. 8), повинні мати тип, який відповідає типу параметра.

Тип **`string`** мови Pascal – це фактично короткі рядки Delphi. У Delphi короткі рядки введено для сумісності з мовою Pascal. Зарезервоване слово **`string`** у Delphi може забезпечити створення як довгого, так і короткого рядка. Якщо при описі рядкової змінної після ідентифікатора **`string`** у квадратних дужках зазначена цілочислова константа, то в цьому випадку створюється короткий рядок. У противному разі тип створюваного рядка визначається налаштуванням компілятора. За умовчанням тип **`string`** еквівалентний типу `AnsiString`, що відповідає установці директиви компілятору `{$H+}` (саме так ми будемо трактувати тип **`string`** надалі). Якщо необхідне трактування типу **`string`** як `ShortString`, застосовується директива `{$H-}`. Це ж можна здійснити в

ICP Delphi, виконавши команду Project ► Options (Проект ► Опції) і встановивши або скинувши на вкладці Compiler (Компілятор) відкриваного діалогового вікна прапорець Huge Strings (Великі рядки)

При використанні типу `ShortString` для зберігання рядкового значення виділяється обсяг пам'яті в 255 байт, у кожному з байтів якого може зберігатися тільки один символ (тобто кожен елемент рядка фактично має тип `Char`). На самому початку області виділяється додатковий байт для зберігання символу, код якого дорівнює фактичній поточній довжині рядкового значення. Таким чином, під змінну типу `ShortString` насправді виділяється 256 байт. Якщо рядок інтерпретувати як масив, то у випадку короткого рядка елемент із номером 0 – це символ, що визначає довжину рядка.

Аналогічно відводиться пам'ять та інтерпретується збережене значення при використанні типу **`string`**`[N]` – при цьому виділяється пам'ять обсягом `N+1` байт. Замість квадратних дужок можна вживати пари символів `(.` та `.)`, так само як і при описі масивів.

Максимальна довжина короткого рядка не може перевищити 255 або значення, зазначеного при його опису. Обсяг пам'яті, що відводиться під рядки типу `AnsiString`, `PChar` і `WideString`, не може перевищити 2 Гбайт, що приблизно відповідає 2^{31} символів для типів `PChar` і `AnsiString` та 2^{30} символів для типу `WideString`. Максимальна довжина рядка, що записується в символьний масив з нульовою базою, обмежується обсягом відведеної пам'яті, але також не може перевищити 2 Гбайт.

Особливістю роботи з `AnsiString`-рядками є те, що змінні даного типу є вказівниками (див. розд. 7) і пам'ять під них виділяється динамічно в міру необхідності, причому вони завершуються символом `#0`. Якщо рядок займав у пам'яті певний обсяг, то при його змінюванні пам'ять виділяється заново, і в неї переписується нове рядкове значення. При цьому пам'ять, у якій розташовувалося старе рядкове значення, звільняється не обов'язково – все залежить від того, чи не посилається на цю ділянку пам'яті інша рядкова змінна.

При присвоюванні рядковій змінній значення, що зберігається в іншому рядку, звичайне копіювання не здійснюється, і копія рядка не створюється. У цьому випадку нова рядкова змінна посилатиметься на ділянку пам'яті, на яку посилається і рядок-оригінал. Це забезпечується методикою відведення пам'яті під довгі рядки: безпосередньо перед рядковим значенням виділяються два блоки пам'яті по 4 байт, у перший з яких заноситься лічильник, що вказує, скільки вказівників типу `AnsiString` посилається на цю ділянку пам'яті, а в другий заноситься так званий індикатор кінця рядка, що фактично є її довжиною. Значення лічильника поси-

лань та індикатора кінця рядка змінюються автоматично з появою нових посилань і зміною рядкових змінних, що раніше посилалися на цю ділянку пам'яті. Якщо значення лічильника дорівнюватиме нулю, пам'ять звільняється. Лічильник посилань й індикатор довжини рядка зберігаються як значення типу `Integer`.

Нехай, наприклад, описані змінні `str1` і `str2` типу **`string`** і нехай у програмі зустрілася така послідовність операторів:

```
str1 := 'Delphi';                    //Дивись рис. 5.1,а
str2 := str1;                        //Дивись рис. 5.1,б
str1 := str1 + ' 7';                 //Дивись рис. 5.1,в
```

Тоді сказане вище умовно ілюструють схеми на рис. 5.1.

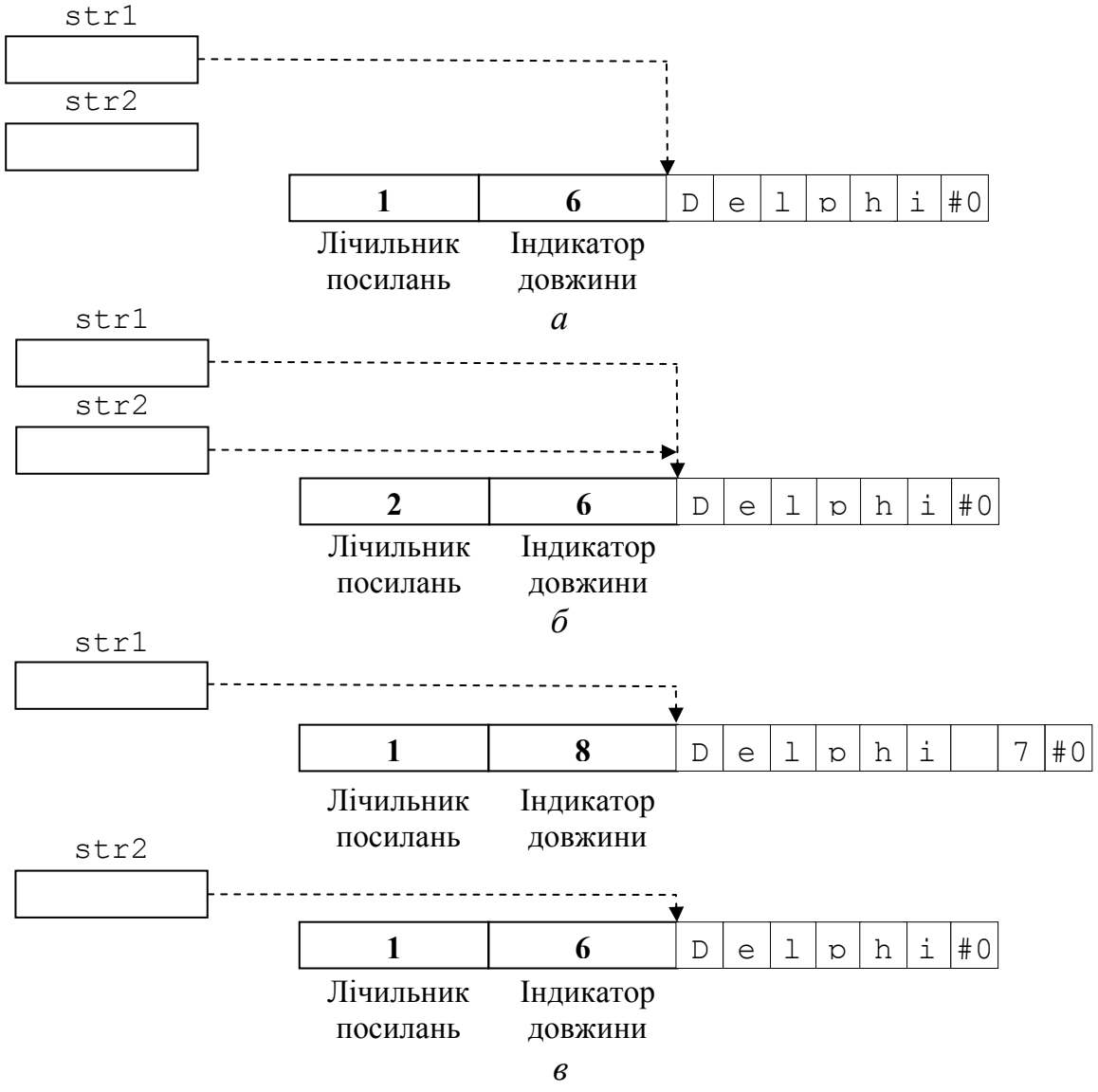


Рис. 5.1. Відведення пам'яті при роботі з довгими рядками

Аналогічно працюють з рядками типу `WideString`, які використовують кодування Unicode, що дозволяє подати 65536 можливих символів типу `WideChar`.

У випадку коротких рядків механізм рядкових присвоювань має деякі особливості:

1. Якщо довжини одержувача і джерела однакові, проблем немає:

```
Str6 := 'Delphi';
```

2. Якщо одержувач довший за джерело (`StrShort:='Delphi 7'`), то фактична довжина одержувача встановлюється такою, як довжина рядка-джерела (таким чином, фактична довжина рядка `StrShort` дорівнюватиме 8, а її значенням буде `'Delphi 7'`).

3. Якщо одержувач коротше джерела (`Str6 := 'Delphi 7'`), то поточна довжина рядка-одержувача дорівнюватиме максимальній в оголошенні, а значенням буде усічене праворуч на потрібне число символів значення рядка-джерела (у цьому випадку `'Delphi'`).

Оголошення типізованої константи для будь-якого рядкового типу виконується у звичайний спосіб. Наприклад:

```
const
  s: string = 'Delphi 7';
```

Виводити рядкові значення можна у звичайний спосіб: присвоюванням їхній властивості Caption видимих компонентів (наприклад, мітки), присвоюванням властивості Text однорядкового редактора, вставкою у вікно багаторядкового редактора тощо. У звичайний спосіб здійснюється і введення рядків.

5.2.2. Порівняння рядків та їхня конкатенація

За правилами Delphi з допомогою операцій =, <, <=, >, >=, <> дозволяється порівнювати рядкові змінні, константи і вирази з будь-якими максимальними та поточними довжинами. При цьому для встановлення факту рівності необхідно, щоб порівнювані об’єкти мали однакові фактичні довжини та точно збіжні значення символів. Коли в програмі порівнюються два рядки, насправді відбувається серія попарних порівнянь їхніх символів зліва направо до першої розбіжності або вичерпання одного з рядків. Меншим буде той рядок, у якого менший код першого незбіжного символу (поза залежністю від максимальних і поточних довжин порівнюваних рядків); якщо один рядок збігається з початком іншого, то більшим буде довший рядок (наприклад, є істинними співвідношення `'Іван' < 'Іваненко'` й `'довше' > 'довжина'`).

Два рядки можуть бути зчеплені один з одним за допомогою операції конкатенації (зчеплення), позначуваної символом «+».

Так, виконання операторів

```
StrAnsi := 'Delphi';
Str := StrAnsi + ' 7';
```

дасть результат `'Delphi 7'`, що буде записаний у `Str`.

Операція конкатенації має більш вищий порівняно з операціями відношення =, <, <=, >, >=, <>.

5.2.3. Робота з окремими елементами рядка

При роботі з рядками можна звертатися до окремих їхніх елементів (символів) аналогічно тому, як це робиться при обробці елементів масиву. Окремі елементи рядкової змінної мають усі властивості змінної типу `Char`.

Можна коректувати будь-який елемент рядкової змінної, для чого у відповідному операторі досить зазначити ім'я рядкової змінної, слідом за яким у квадратних дужках задається номер її елемента (наприклад, `st[10] := 'f'`). Пара складених символів `(.` та `.)` і тут заміняє квадратні дужки. Елементи рядків, крім рядків типу `PChar` і символьних масивів з нульовою базою, нумеруються від 1 (елементи згаданих двох типів нумеруються від 0).

Якщо індексація використовується для змінювання окремих елементів довгих рядків, то при значенні лічильника посилань, більшого від одиниці, створюється копія змінюваного рядка, у якій власне і виконується змінювання елементів; якщо ж лічильник посилань дорівнює одиниці, копія рядка не створюється.

У кожній короткій рядковій змінній є елемент із номером 0, у якому у вигляді символу зберігається поточна довжина рядка. Щоб дізнатися про фактичну довжину короткого рядка, потрібно застосувати функцію `Ord` до нульового елемента рядка (наприклад, `k := Ord(StrShort[0])`, де `k` – змінна цілого типу) або звернутися до функції `Length`. Нульовий елемент коротких рядків можна корегувати; при цьому буде змінюватися поточна довжина рядка (наприклад, оператор `StrShort[0] := #65` задає поточну довжину рівною 65).

```
//Приклад 5.6
//Дано рядок типу ShortString. Якщо в ньому останні символи
//однакові, залишити тільки один з них, усунувши їх
//дублювання.
```

Скористаємося формою з прикладу 2.2, змінивши значення властивості Caption мітки на Уведіть рядок і натисніть кнопку "Виконати", а кнопки Button1 – на Виконати.

У секцію **private** класу TForm1 внесемо такий опис:

```
s: ShortString;
```

Опрацьовувач події OnClick компонента Button1 у цьому випадку може мати такий вигляд:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  s := edInput1.Text;
  mmOutput1.Lines.Add('Початковий рядок:');
  mmOutput1.Lines.Add(s);
  while (s[0] > #1) and
        (s[Ord(s[0])] = s[Ord(Pred(s[0]))]) do
    Dec(s[0]);        //Те ж, що й s[0] := Chr(Ord(s[0]) - 1);
                                    //або s[0] := Pred(s[0]);
  mmOutput1.Lines.Add('Результуючий рядок:');
  mmOutput1.Lines.Add(s);
  Button1.Visible := False;
  edInput1.Visible := False;
  lbOutput1.Visible := False;
end;
```

У цій процедурі нерівність s[0]>#1 (порівняння символів) еквівалентна нерівності Length(s)>1. Номер останнього символу рядка Ord(s[0]), а для доступу до передостаннього символу вжито функцію Pred – номер передостаннього символу дорівнює Ord(Pred(s[0])). Робота програми ґрунтується на послідовному зменшенні довжини рядка за рахунок зміни символу s[0].

Розглянутий у прикладі 5.6 варіант корекції довжини рядка є прийнятним при роботі з короткими рядками. При роботі з даними типу PChar та **array of** Char варто пам'ятати про індексування їхніх елементів від нуля, а також про те, що в такого роду рядках є завершальний символ #0, який зберігається наприкінці рядка, а елемент із номером 0 – це звичайний елемент рядка. Аналогічний метод роботи із завершальним нульовим символом у довгих рядках може спричинити порушення доступу, оскільки пам'ять під рядок у цьому випадку заново не виділяється, а отже, і не коректується значення індикатора кінця рядка.

Розв'яжемо попередню задачу для випадку довгого рядка.

```
//Приклад 5.7
//Даний довгий рядок. Якщо в ньому останні символи однакові,
//залишити тільки один з них, усунувши їх дублювання.
```

Скористаємося тією же формою, що й у прикладі 5.7, але рядок `s` у секції **`private`** класу `TForm1` опишемо з типом **`string`**. Крім того, опишемо також тип `TPInt` як вказівник на тип `Integer`.

Опрацьовувач події OnClick компонента Button1 у цьому випадку може мати такий вигляд:

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TPInt = ^Integer;
var
  Len: Integer;
begin
  s := edInput1.Text;
  mmOutput1.Lines.Add('Початковий рядок:');
  mmOutput1.Lines.Add(s);
  Len := TPInt(PChar(@s[1]) - 4)^;  //Визначаємо довжину рядка
  while (Len > 1) and (s[Len] = s[Len - 1]) do begin
    s[Len] := #0;  //Записуємо завершальний символ та змінюємо
    Dec((PChar(@s[1]) - 4)^);        //індикатор довжини рядка
    Len := TPInt(PChar(@s[1]) - 4)^;      //Нова довжина рядка
  end;
  mmOutput1.Lines.Add('Результуючий рядок:');
  mmOutput1.Lines.Add(s);
  Button1.Visible := False;
  edInput1.Visible := False;
  lbOutput1.Visible := False;
end;
```

Як це видно з тексту наведеної вище процедури, при зміні положення завершального символу `#0` у довгому рядку, щоб не порушити доступ до нього, необхідно додатково корегувати значення, записане в індикаторі його довжини, що, по-перше, призведе до неефективного коду й, по-друге, є потенційним джерелом можливих помилок. Зокрема, якщо подібно тому, як у попередній процедурі, збільшувати значення індикатора довжини рядка (наприклад, за допомогою оператора `Inc((PChar(@s[1])-4)^)`) з одночасною зміною положення завершального символу з кодом `0`, то матиме місце порушення доступу. Порушення доступу матиме місце і при спробі звертання до індикатора довжини рядка, якщо рядок порожній (наприклад, у наведеній процедурі у разі запису в змінну `s` порожнього рядка виконання оператора `Len := TPInt(PChar(@s[1]) - 4)^` приведе до порушення доступу).

Зі сказаного вище можна зробити висновок про те, що змінювати довжину `AnsiString`-рядка за допомогою прямого звертання до її елементів **не треба** (можна вважати, що **заборонено**). Не рекомендується

також передавати індекси довгих рядків як параметри при виклику підпрограм, оскільки це призводить до неефективного коду.

5.2.4. Процедури і функції для обробки рядків

Відразу зазначимо, що при обробці символьних масивів з нульовою базою і даних типу `PChar` перелічені нижче функції та процедури розглядають такого роду рядки як рядки з нумерацією елементів від 1 без порушення самої нумерації.

Замість операції конкатенації в рядкових виразах можливе використання функції `Concat`, причому як її параметр може виступати будь-яка кількість рядків:

```
ім'я_рядка := Concat(рядок1, рядок2, рядок3,...);
```

Значення, що вона повертає, – це рядок, який утвориться приєднанням значень параметрів: другого до кінця першого, третього до результату попереднього приєднання і т. д. Наприклад:

```
st := Concat(st1, st2, st3);
```

Для визначення поточної довжини рядка найчастіше застосовується функція `Length`, що повертає кількість символів, які містяться в даний момент у рядку, зазначеному як параметр:

```
змінна_цілого_типу := Length(рядок);
```

Для пошуку деякої послідовності символів у рядку уведено функцію `Pos`, що має такий формат:

```
p := Pos(шуканий_рядок, оброблюваний_рядок);
```

Обидва параметри можуть бути будь-якими рядковими виразам; значення, що повертає функція `Pos`, має тип `Integer`.

Якщо, наприклад, необхідно знайти рядок `st1` у рядку `st`, то при звертанні `p := Pos(st1, st)` у змінну цілого типу `p` буде записане число, що визначає ту позицію рядка `st`, у якій була виявлена перша поява рядка `st1`. Якщо `st1` відсутній у `st`, то результатом буде `0`.

Функція `Copy` використовується в Delphi для копіювання частини рядка. Можливий формат звертання до неї такий:

```
рядок1 := Copy(рядок, початкова_позиція, кількість_символів);
```

Функція повертає рядок, утворений зазначеним числом символів (параметр кількість_символів), розташованих у заданому рядковому виразі (параметр рядок) підряд, починаючи з позиції, номер якої визначається параметром початкова_позиція. Наприклад, оператор

```
st := Copy('останній', 2, 3);                    //st='стан'
```

записує у рядкову змінну st значення 'стан'.

Якщо значення другого параметра задає позицію з номером більшим поточної довжини вихідного рядка, то повертається порожній рядок. При значенні другого параметра, меншому за 1, позиція початку копіювання визначається третім параметром. Якщо третій параметр дає вихід за поточну довжину рядка, то копіюються тільки реально існуючі символи (при від'ємному третьому параметрі результат – порожній рядок):

```
st := Copy('TForm', 2, Length('TForm'));    //st='Form'
```

```
//Приклад 5.8
//Увести рядковий масив з 10 елементів типу
//"Прізвище Ім'я". Перетворити всі рядки до вигляду
//"Ім'я Прізвище", після чого вивести їх на екран.
```

Скористаємося формою з прикладу 2.2, змінивши значення властивості Caption мітки на Уведіть рядок, а кнопки Button1 – на Введення.

У секцію **private** класу TForm1 внесемо такі описи:

```
Count: Integer;
s: array[1..10] of string;
```

Створимо також для форми опрацьовувач події OnCreate, що містить тільки один оператор:

```
Count := 1;
```

Опрацьовувач події OnClick компонента Button1 у цьому випадку може мати такий вигляд:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i, k: Integer;
begin
  s[Count] := edInput1.Text;
  if Count < 10 then begin;
    Inc(Count);
    edInput1.SetFocus;
  end
  else begin
    for i := 1 to Count do begin
      k := Pos(' ', s[i]);              //Пошук позиції пробілу
      s[i] := Concat(Copy(s[i], k + 1, Length(s[i])),
            ' ', Copy(s[i], 1, k - 1));        //Можна  s[i]:=
           // Copy(s[i],k+1,Length(s[i]))+' '+Copy(s[i],1,k-1
    end;
    mmOutput1.Lines.Add(' Результат:');
    for i := 1 to Count do
```

```
      mmOutput1.Lines.Add(s[i]);
    Button1.Visible := False;
    edInput1.Visible := False;
    lbOutput1.Visible := False;
  end
end;
```

У цій підпрограмі після введення 10 рядків у масив `s`, кожен елемент `s[i]` якого є рядком, здійснюється перетворення уведених рядків у циклі за змінною `i`, що змінює своє значення від `1` до `Count` (у змінній `Count` по закінченні введення буде записана кількість уведених рядків). Функція `Pos` забезпечує визначення позиції першого пробілу в перетвореному рядку. Далі за допомогою функції `Concat` зчіплюються частини рядка, що знаходиться за пробілом (перше звертання до функції `Copy`), символу «пробіл» і частини рядка до пробілу (друге звертання до функції `Copy`). При першому звертанні до функції `Copy` як третій параметр виступає довжина рядка, а оскільки при цьому фактично виконується запит на копіювання більшої кількості символів, ніж їх є від позиції початку копіювання, то копіювання проводиться до кінця рядка.

Для видалення частини рядка використовується процедура `Delete`, що має три параметри:

```
Delete(рядок,поч_позиція,кількість_символів_що_видаляється);
```

Перший параметр – це рядкова змінна, два інших – будь-які цілочислові вирази.

Наприклад, наступні два оператори записують у `st` значення `'метр'`:

```
st := 'метеор';
Delete(st, 4, 2);
```

При застосуванні процедури `Delete` видаляється задане число символів, починаючи з зазначеної позиції, зі зсувом на позиції, що звільнилися, символів, розташованих правіше від вилучених, і зміною поточної довжини рядка. Якщо другий параметр перевищує поточну довжину або менший за 1, а також при від'ємному значенні третього параметра видалення не відбувається.

Процедура `Insert` виконує вставку рядка в деякий інший рядок, починаючи з зазначеної позиції, попередньо відсунувши вправо все, що заважає вставці. Формат процедури:

```
Insert(рядок_що_ вставляється, рядок_що_приймає, поз_вставки);
```

У випадку коротких рядків, якщо максимальна довжина рядка, у який виконується вставка, менша результуючої довжини, то відбувається усікання результату праворуч. Якщо позиція вставки більша від поточної довжини рядка, то перший параметр приєднується до кінця другого. Значення позиції вставки, що менше за 1, приводить до вставки перед першим символом.

Наприклад, оператори

```
st := 'програвати';
Insert('мув', st, 7);
```

забезпечують запис в `st` значення `'програмувати'`.

```
//Приклад 5.9
//Замінити в рядку всі послідовності символів 'Turbo Pascal'
//на сполучення символів 'Delphi 7'.
```

Для розв'язання задачі скористаємося формою з прикладу 5.6, не змінюючи її. При цьому в секції **`private`** класу `TForm1` опишемо поле `s` типу **`string`**. Тоді задачу розв'язує такий опрацьовувач події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  k: Integer;
begin
  s := edInput1.Text;
  mmOutput1.Lines.Add('Початковий рядок:');
  mmOutput1.Lines.Add(s);
  k := Pos('Turbo Pascal', s);          //k – позиція підрядка
  while k <> 0 do begin
   Delete(s, k, Length('Turbo Pascal'));   //Видаляємо підряок
   Insert('Delphi 7', s, k);           //Вставка в k-у позицію
   k := Pos('Turbo Pascal', s);             //Повторюємо пошук
  end;
  mmOutput1.Lines.Add('Результуючий рядок:');
  mmOutput1.Lines.Add(s);
  Button1.Visible := False;
  edInput1.Visible := False;
  lbOutput1.Visible := False;
end;
```

У деяких задачах виникає необхідність у перетворенні числових даних, записаних у рядковому виді, до якого-небудь числового формату. Для цих цілей у Delphi уведено процедуру `Val`, формат звертання до якої такий:

`Val(`рядок, числова_змінна, цілочислова_змінна`);`

Ця процедура присвоює другому аргументу числове значення, що зберігається у виразі, записаному як перший аргумент. Якщо під час перетворення не виявлена помилка, то в третій параметр записується 0; якщо помилка виявлена, то в третій параметр записується номер позиції першого помилкового символу рядка, а значення другого параметра буде дорівнювати нулю (у нуль-термінальних рядках при помилці повертається збільшений на 1 номер позиції першого помилкового символу). Другий параметр може мати будь-який числовий тип, і саме до цього типу буде виконуватися перетворення. У перетвореному рядку можуть бути лідируючі пробіли; пробіли ж наприкінці рядка зумовлюють діагностику помилки за допомогою третього параметра. Перетворення до цілого числа здійснюється завжди до першого помилкового символу з записом результату перетворення в другий параметр. Якщо ж перетворюється дійсне число, і виявляється помилка, то проводиться перетворення до цілого з ігноруванням десяткової точки та знака.

Наприклад, нехай у програмі є такі описи:

```
var
  r: Real;
  m, p: Integer;
  n: LongInt;
```

Нижче наведені приклади використання процедури Val:

```
Val('-12', m, p);                 // m = -12; p = 0
Val('100023', n, p);              // m = 100023; p = 0
Val('-12e+004', r, p);            // r = -1.2e-05; p = 0
Val('-12e+004', n, p);            // n = 0; p = 4
Val('-1.2e+004 ', r, p);          // r = 12; p = 10
Val('   5623', n, p);             // n = 5623;  p 0
Val('5623   ', n, p);             // n = 5623; p = 5
```

```
//Приклад 5.10
//Рядок складається зі слів (ланцюжків символів, відділених
//один від одного одним або декількома пробілами). Вибрати
//ті зі слів, які є записом цілих чисел, а слова,
//що залишилися, розсортувати на слова, що являють
//собою числа типу Real, і слова, що не є
//записом чисел в Delphi.
```

Для розв'язання задачі скористаємося формою з прикладу 5.6, не змінюючи її. При цьому в секції **private** класу TForm1 за допомогою ідентифікаторів s, s_I64, s_Ext, s_Str типу **string** опишемо відповідно початковий рядок, рядок з цілих чисел, рядок з дійсних чисел і рядок з інших слів. Тоді задачу дозволяє розв'язати такий опрацьовувач події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: string;
  k, p: Integer;
  Ex: Extended;
  I: Int64;
begin
  s := edInput1.Text;
  mmOutput1.Lines.Add('Початковий рядок:');
  mmOutput1.Lines.Add(s);
  while s[1] = ' ' do
    Delete(s, 1, 1);      //Видалення пробілів: початкових,...
  while s[Length(s)] = ' ' do
    Delete(s, Length(s), 1);               //...кінцевих й...
  while Pos('  ', s) <> 0 do
    Delete(s, Pos('  ', s), 1);          //...тих, що двояться
  s_I64 := ''; s_Ext := ''; s_Str := '';
  while s <> '' do begin
    k := Pos(' ', s);
    if k = 0 then k := Length(s)+1; //Обробка останнього слова
    s1 := Copy(s, 1, k - 1);                //Виділення слова
    Val(s1, I, p);                //Перетворення до типу Int64
    if p = 0 then s_I64 := s_I64 + s1 + ' '    //Перетворилося
    else begin                                          //Ні
      Val(s1, Ex, p);          //Перетворення до типу Extended
      if p = 0 then s_Ext := s_Ext + s1 + ' '  //Перетворилося
             else s_Str:= s_Str + s1 + ' ';               //Ні
    end;
    Delete(s, 1, k);       //Видалення слова разом із пробілом
  end;     //В рядках s_I64,  s_Ext та s_Str останній символ -
  Delete(s_I64, Length(s_I64), 1);    //пробіл. Видаляємо його
  Delete(s_Ext, Length(s_Ext), 1);
  Delete(s_Str, Length(s_Str), 1);
  mmOutput1.Lines.Add('Цілі числа: ' + s_I64);
  mmOutput1.Lines.Add('Дійсні числа: ' + s_Ext);
  mmOutput1.Lines.Add('Слова: ' + s_Str);
  Button1.Visible := False;
  edInput1.Visible := False;
  lbOutput1.Visible := False;
end;
```

Зворотною стосовно процедури `Val` є процедура `Str`, формат якої

`Str(`арифметичний_вираз, рядок`);`

Процедура перетворює значення свого першого параметра в рядкову змінну, задану як другий параметр. Знак «+» у рядок не заноситься. Після арифметичного виразу може бути зазначений формат у вигляді одного або

двох цілих додатних чисел, відокремлюваних одне від одного і від першого параметра двокрапкою. Перший параметр формату задає кількість знакомісць, що відводиться у рядку під запис числа (при цьому ліворуч рядок доповнюється пробілами), а другий – кількість символів після десяткової точки (можна задавати тільки при перетворенні дійсних значень). Елементи формату ігноруються, якщо їх значення не дозволяють здійснити перетворення.

Наприклад:

```
var
  st: string;
  ...
  st := Str(15 - 2 * 3, st); //st='9', поточна довжина 1
```

```
//Приклад 5.11
//Дано натуральне число n. Одержати рядкове подання
//цього числа у вигляді послідовності цифр і пробілів,
//що відокремлюють групи по три цифри, починаючи праворуч.
```

Для розв'язання задачі скористаємося формою з прикладу 5.6, замінивши текст у властивості Caption мітки текстом Уведіть число й натисніть кнопку "Виконати". У секцію **private** класу TForm1 включимо такі описи:

```
    n: Int64;                                  //Задане число
    n_str: string;                          //Результуючий рядок
```

Тоді задачу розв'язує такий опрацьовувач події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  n := StrToInt64(edInput1.Text);
  mmOutput1.Lines.Add('Задане число');
  mmOutput1.Lines.Add(edInput1.Text);
                              //Одержуємо рядкове подання числа
  Str(n, n_str);
                  // i - це позиція, у яку вставляється пробіл
  i := Length(n_str) - 2;
  while i > 1 do begin
    Insert(' ', n_str, i);
    i := i - 3;
  end;
  mmOutput1.Lines.Add('Результат');
  mmOutput1.Lines.Add(n_str);
  Button1.Visible := False;
  edInput1.Visible := False;
```

```
  lbOutput1.Visible := False;
end;
```

Розв'яжемо ще одну найпростішу задачу, орієнтовану на використання процедури `Str`.

```
//Приклад 5.12
//Вивести дві таблиці символів (основну і додаткову)
//із зазначенням десяткових кодів символів.
```

Скористаємося формою з прикладу 2.2, видаливши з неї мітку і однорядковий редактор і записавши у властивість Caption компонента Button1 текст Вивести. Крім того, напишемо наступний опрацьовувач події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i, j, k: Integer;
  s, s1: string;
const
  Symb: array [0..32] of string = ('NUL', 'SOH', 'STX', 'ETX',
       'EOT', 'ENQ', 'ACK', 'BEL', 'BS ', 'HT ', 'LF ', 'VT ',
       'FF ', 'CR ', 'SO ', 'SI ', 'DEL', 'DC1', 'DC2', 'DC3',
       'DC4', 'NAK', 'SYN', 'ETB', 'CAN', 'EM ', 'SUB', 'ESC',
       'FS ', 'GS ', 'RS ', 'US ', 'BL ');
begin
  for k := 0 to 1 do begin
    if k = 0 then
      s := 'Основна частина кодової таблиці'
    else
      s := 'Додаткова частина кодової таблиці';
    mmOutput1.Lines.Add(s);
    s := '';
    for j := 1 to 4 do
      s := s + 'Код Символ   ';
    mmOutput1.Lines.Add(s);
    for i := 0 to 31 do begin
      s := '';
      for j := 0 to 3 do begin
        str(k * 128 + i + j * 32:3, s1);
        s := s + s1 + ' ';
        if k * 128 + i + j * 32 < 33
         then s := s + Symb[k * 127 + i + j * 32] + '      '
         else s := s + Chr(k * 128 + i + j * 32) + '       ';
      end;
      mmOutput1.Lines.Add(s);
    end;
  end;
end;
```

Тут процедура Str застосовується для формування рядкового подання кодів символів. При цьому використано формат для зазначення кількості елементів рядка s1, що відводяться для запису коду символу. У тексті наведеної процедури ілюструється також опис рядкового масиву як типізованої константи (масив Symb), використання операції зчеплення рядків та використання функції Chr.

Зазначимо також, що дана програма вживалася для формування таблиць 3.5 і 3.6.

При роботі з символами в рядках, а також з окремими символами досить часто виникає необхідність у переході від малих латинських літер до великих. Для цього передбачено функцію UpCase, що має формат

символьна_змінна := UpCase(символ).

Функція не здійснює зворотного перетворення, так само як не перетворює жодних інші символи, у тому числі українські (російські) літери.

```
//Приклад 5.13
//Дано рядок, що складається з латинських літер (великих і
//малих)і пробілів. Чи правда, що він без урахування пробілів
//однаково читається справа наліво і зліва направо?
```

Для розв'язання задачі скористаємося без змін формою з прикладу 5.6, описавши в секції **private** класу TForm1 поле s типу **string**. Задача розв'язується наступним опрацьовувачем події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  f: Boolean;
begin
  s := edInput1.Text;
  mmOutput1.Lines.Add('Початковий рядок:');
  mmOutput1.Lines.Add(s);
  i := Length(s);
  while i >= 1 do begin
    if s[i] = ' '
    then Delete(s, i, 1)                    //Видалення пробілів
    else s[i] := UpCase(s[i]);       //Перехід до великих літер
    i := i - 1;
  end;
  f := True;
  i := 1;
  while f and (i <= Length(s) div 2) do begin
    if s[i] <> s[Length(s) - i + 1] then f := False;
```

```
    i := i + 1;
  end;
  if f then mmOutput1.Lines.Add('Правда')
  else mmOutput1.Lines.Add('Неправда');
  Button1.Visible := False;
  edInput1.Visible := False;
  lbOutput1.Visible := False;
end;
```

У даній програмі виконується досить активна робота з окремими елементами рядка, а саме, їхнє порівняння та корекція.

Описані в даному пункті процедури і функції перейшли в Delphi з мови Turbo Pascal. Однак у Delphi існує достатня кількість своїх процедур і функцій, орієнтованих на можливості і вимоги Windows та Linux. Насамперед, до цих підпрограм належать функції перетворення числових даних до рядкових і зворотно:

- `StrToCurr(`рядок`)` – повертає значення типу `Currency`;
- `StrToFloat(`рядок`)` – повертає значення типу `Extended` (враховується роздільник цілої та дробової частин, установлений за умовчанням або за допомогою системної змінної `DecimalSeparator`);
- `StrToInt(`рядок`)` – повертає значення типу `Integer`;
- `StrToInt64(`рядок`)` – повертає значення типу `Int64`;
- `FloatToStr(`числовий_вираз`)` – перетворить дійсне число до рядкового подання, повертаючи значення типу **`string`** з роздільником цілої та дробової частин, що задаються за умовчанням або за допомогою системної змінної `DecimalSeparator`;
- `IntToStr(`цілочисловий_вираз`)` – перетворює задане першим параметром значення в рядок, повертаючи значення типу **`string`**;
- `Int64ToStr(`цілочисловий_вираз`)` – працює аналогічно функції `IntToStr`.

Зазначимо, що при звертанні до функцій `StrToCurr`, `StrToFloat`, `StrToInt` і `StrToInt64` у рядку не повинно бути початкових і кінцевих узагальнених пробільних символів, для видалення яких можна використовувати функцію

`Trim(`рядок`)`,

що повертає значення типу **`string`**. Початкові та кінцеві пробіли в параметрі функцій `StrToCurr` і `StrToFloat` та початкові пробіли в параметрі функцій `StrToInt` і `StrToInt64` можуть бути присутніми.

До деяких із зазначених вище функцій ми вже зверталися в наведених раніше прикладах розв'язання задач.

Функції

```
StrToIntDef(рядок, цілочисловий_вираз)
```

та

```
StrToInt64Def(рядок, цілочисловий_вираз)
```

здійснюють перетворення рядка з видачею у випадку помилкового перетворення значення, що задається другим параметром.

Раніше в прикладі 5.4 застосовувалася функція `StrToFloat`, що має такий формат звернення:

```
FloatToStrF(числовий_вираз, формат, точність, цифри).
```

Ця функція перетворює дійсне число в рядок відповідно до формату, що задається другим параметром. Другий параметр може набувати одного зі значень наступного переліченого типу, що визначений у модулі `SysUtils`:

```
type
  TFloatFormat = (ffGeneral, ffExponent, ffFixed,
                  ffNumber, ffCurrency).
```

Зміст третього та четвертого параметрів залежить від значення другого параметра. Так, якщо як другий параметр задати значення `ffFixed`, то третій параметр визначає кількість цифр у запису числа, а четвертий – кількість цифр у дробовій частині; якщо другий параметр дорівнює `ffExponent`, то третій параметр задає кількість цифр у запису мантиси, а четвертий – кількість цифр у запису десяткового порядку. Формат `ffGeneral` переключає перетворення на один з форматів `ffFixed` і `ffExponent` залежно від значення числа. Формат `ffCurrency` є грошовим форматом (ураховує знак національної грошової одиниці), а формат `ffNumber` вживається при виведенні великих чисел і забезпечує поділ тисяч.

Визначено й інші функції взаємного перетворення числових і рядкових даних, а також перетворення рядкових даних до даних типу `TDataTime` і зворотно.

У мові визначений також ряд функцій перетворення регістра літер, що входять у рядок (повертають значення типу **`string`**):

- `AnsiLowerCase(рядок)` – повертає перетворений рядок, у якому всі літери (у тому числі й національного алфавіту) замінені на малі;

- `AnsiUpperCase(`рядок`)` – повертає перетворений рядок, у якому всі літери (у тому числі й національного алфавіту) замінені на великі;
- `LowerCase(`рядок`)` – повертає перетворений рядок, у якому всі латинські літери замінені на малі (інші символи не перетворюються);
- `UpperCase(`рядок`)` – повертає перетворений рядок, у якому всі латинські літери замінені на великі (інші символи не перетворюються).

Досить зручною є функція

`StringOfChar(`символ, цілочисловий_вираз`)`,

яка формує рядок із символів, що задаються першим параметром, у кількості, що визначається другим параметром.

Для порівняння рядків у Delphi визначено функції, що повертають значення типу `Integer` і мають параметри типу **`string`**:

- `AnsiCompareStr(`рядок1, рядок2`)` – порівняння рядків у визначеній регіональним стандартом системі сортування символів з урахуванням регістру (у тому числі для символів національного алфавіту);
- `AnsiCompareText(`рядок1, рядок2`)` – порівняння рядків у визначеній регіональним стандартом системі сортування символів без урахування регістру (у тому числі для символів національного алфавіту);
- `CompareStr(`рядок1, рядок2`)` – порівняння рядків за кодами символів з урахуванням регістру;
- `CompareText(`рядок1, рядок2`)` – порівняння рядків за кодами символів без урахування регістру латинських літер (регістр літер національного алфавіту враховується).

Значення, що повертає кожна з цих функцій, визначається таким правилом:

- результат порівняння – будь-яке значення менше `0`, якщо рядок1 < рядок2;
- результат порівняння дорівнює `0`, якщо рядок1 = рядок2;
- результат порівняння – будь-яке значення більше `0`, якщо рядок1 > рядок2.

Функції `AnsiCompareStr` та `AnsiCompareText` орієнтовані на впорядкування символів, яке визначається не таблицею ANSI-кодів, а регіональними стандартами, відповідно до яких великі та малі літери

національних алфавітів розташовуються поруч, причому велика літера передує малій (це характерно для більшості національних алфавітів).

Наступні функції перевіряють два рядки на рівність:

```
AnsiSameStr(рядок1, рядок2),
AnsiSameText(рядок1, рядок2),
SameText(рядок1, рядок2).
```

Ці функції повертають булівське значення `True`, якщо параметри виклику рівні, і значення `False` у противному разі. При цьому друга функція при порівнянні не враховує регістр літер як латинського, так і національного алфавіту. Функції `AnsiSameStr` та `AnsiSameText` орієнтовані на впорядкування символів, визначене регіональним стандартом, у той час як функція `SameText` орієнтована на впорядкування символів за їх кодами.

5.2.5. Рядки і масиви символів

У Delphi змінна типу **`array of`** `Char` з нумерацією від нуля (символьний масив з нульовою базою) може розглядатися як рядок постійної довжини. Змінні такого типу придатні для вільного використання в будь-яких рядкових виразах. При цьому компілятор автоматично перетворить такий масив у рядок, довжина якого дорівнює кількості елементів масиву з урахуванням того, що якщо як елемент масиву зустрічається символ `#0`, то він сприймається як ознака кінця рядка (***нуль-термінальний рядок***, тобто рядок, що закінчується нулем). Масиви типу **`array of`** `Char` можна порівнювати один з одним і часто поводитися з ними так само, як зі змінними типу **`string`** (але з деякими обмеженнями). В операторі присвоювання можна в лівій частині вказувати ім'я символьного масиву з нульовою базою, а в правій – рядкову константу з довжиною, що дорівнює кількості елементів у масиві. У той же час у Delphi символьним масивам з нульовою базою дозволено присвоювати рядкові константи, довжина яких менша, ніж кількість елементів масиву; в елементи, що залишилися незаповненими, заноситься символ `#0`, але для цього повинен бути включений розширений синтаксис (директива `{$X+}`; за умовчанням розширений синтаксис включений). Однак не можна змінній типу **`array of`** `Char` присвоїти значення рядкової змінної або рядкового виразу (крім виразів над рядковими константами з результуючою довжиною, що дорівнює розмірності масиву). Символьні масиви з нульовою базою можуть вживатися в процедурі `Val` і функціях `Concat` (але не як перший параметр) і `Copy`, а також у всіх функціях перетворення рядка до інших типів даних.

Функція `Length`, застосована до змінної типу **`array of`** `Char`, повертає кількість елементів масиву. Для визначення довжини рядка в цьому випадку потрібно використовувати функцію

`StrLen(`нуль-термінальний_рядок`)`.

Відзначимо, що використання символьних масивів з нульовою базою як рядків більш характерно для мови С.

5.2.6. Тип PChar

У Delphi існує тип `PChar`, що, як і символьний масив з нульовою базою, описує так звані нуль-термінальні рядки. Довжина таких рядків не зазначається явно: рядок починається з першого символу й обмежується замикаючим символом `#0`. Оскільки тип `PChar` визначається як вказівник на символ, подібного роду рядки звичайно створюються в динамічній пам'яті (про неї буде йти мова в розд. 7). З типом `PChar` є сумісним будь-який символьний масив, ліва межа якого дорівнює `0`. Більш того, рядки типу `PChar` індексуються з відліком значень індексу від `0`. Для забезпечення можливості оперування даними типу `PChar` необхідно включити розширений синтаксис, що виконується директивою `{$X+}`.

Рядок типу `PChar` може бути оголошено типізованою константою, але її довжина в цьому випадку не може перевищувати 255. Наприклад:

```
const
  PChars: PChar = 'Delphi 7';
```

До даних типу `PChar` за допомогою операції додавання (+) можна додавати цілі значення, що відповідає зсуву початку такого рядка на зазначене число байт. Аналогічно виконується операція віднімання (−). Уведено також операцію віднімання двох рядків типу `PChar`, результатом якої є цілочислове значення, що відповідає зсуву в пам'яті початку одного такого рядка щодо початку іншого (у байтах).

```
//Приклад 5.14
//Приклади роботи із символьним масивом з нульовою базою
//і рядками типу PChar.
```

Скористаємося формою з прикладу 5.6, видаливши з неї мітку й однорядковий редактор. У секцію **`implementation`** модуля включимо такі описи:

```
const
  CharsArray: array[0..20] of Char = 'Turbo Pascal 7.0';
var
```

```
    PC: PChar;
    s: string;
```

Проаналізуємо текст такого опрацьовувача події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  mmOutput1.Lines.Add('Початковий рядок:');
  mmOutput1.Lines.Add(CharsArray);
  CharsArray[12] := #0;            //Тепер у рядку 11 символів
  mmOutput1.Lines.Add('Рядок після зміни:');
  mmOutput1.Lines.Add(CharsArray);
  PC := CharsArray;         //Установлюємо PC на початок рядка
  mmOutput1.Lines.Add('Рядок PC:');
  mmOutput1.Lines.Add(PC);
  PC[5] := #0;                //Усікаємо рядок за допомогою PC
  mmOutput1.Lines.Add('Новий рядок:');
  mmOutput1.Lines.Add(CharsArray);
  PC := PC + 6;                  //Зсув вказівника на 6 позицій
  mmOutput1.Lines.Add('Новий рядок PC:');
  mmOutput1.Lines.Add(PC);
  s := '';              //Далі зчіплюємо довгий рядок, масив з
                //нульовою базою, символьну константу та PChar
  s := Concat(s, CharsArray, ' ', PC);
  mmOutput1.Lines.Add('Результат конкатенації:');
  mmOutput1.Lines.Add(s);
  Button1.Visible := False;
end;
```

У програмі символьний масив оголошено у вигляді типізованої константи. Оскільки під нього відведено 21 байт, його вміст потрібно було б задавати рядком з 21 символу; однак в Delphi рядкова константа, довжина якої менша від кількості елементів символьного масиву, також записується в символьний масив з автоматичним заповненням кінця масиву символом `#0`. Занесенням символу `#0` в елемент масиву `CharsArray` із номером `12` здійснюється зменшення довжини рядка, який імітується масивом `CharsArray`.

Змінна `PC` після виконання оператора `PC := CharsArray` стає деяким чином синонімом змінної `CharsArray`. При цьому рядок `PC` не має власної пам'яті: змінна `PC` оперує пам'яттю, яку займає масив `CharsArray`. Слід зазначити, що в наведеній програмі коректується масив `CharsArray` (оператор `PC[5] := #0`), оскільки змінна `PC` після присвоювання `PC := CharsArray` просто посилається на цей масив. Тому оператор `PC[5] := #0` змінює довжину рядка `CharsArray` до п'яти символів.

Оператор `PC := PC + 6` виконує зсув у пам'яті вказівника `PC` на 6 байт, тим самим переміщаючи його за символ `#0`, який раніше був записаний усередину масиву `CharsArray`. Оскільки правіше від цього символу вміст масиву `CharsArray` не змінювався, то він трактується як рядок, що адресується за допомогою `PC` (масив `CharsArray` фактично містить два рядки). Оператор `s := Concat(s, CharsArray,' ',PC)` демонструє можливість виконання конкатенації рядків різних типів.

Увага. Для забезпечення можливості змінювання типізованої константи `CharsArray` слід виконати зазначене в п. 3.14.2 налаштування середовища.

Зауваження з приводу застосовності різного виду рядкових процедур і функцій, висловлені стосовно рядків типу **`array of`** `Char`, справедливі й стосовно даних типу `PChar`. При цьому функція `Length` повертає довжину рядка.

Основні засоби роботи з рядками із завершальним нулем зосереджено в модулі `SysUtils`.

5.3. Множини

5.3.1. Оголошення множин

У Delphi існує поняття ***множини***, що має значення якогось «зібрання» елементів одного і того самого базового типу. Базовий тип визначає перелік усіх елементів, які взагалі можуть міститися в даній множині. Як базовий тип може виступати тільки порядковий тип. Наприклад, дійсні числа (`Real` не порядковий тип) і рядки (не простий і не порядковий тип) не можуть бути елементами множини. Розмір множини в Delphi завжди обмежений деякою гранично допустимою кількістю елементів. У множинах можуть міститися тільки такі елементи, порядкові значення яких не виходять за межі `0..255`. Для цілочислових множин це означає, що в них можуть бути включені тільки числа від 0 до 255. Від'ємні елементи множин у Delphi неприйнятні. Тому базовими типами не можуть бути типи `SortInt`, `Integer`, `Word`, `LongInt` та інші цілі типи, окрім типу `Byte` або його відрізка. Природна річ, тип `WideChar` або його відрізки також не можуть бути базовими типами для множини, оскільки дані типу `WideChar` займають два байти. Для множин, що містять звичайні символи, подібні труднощі відсутні, оскільки базовим типом для них є `Char` (а в ньому `256` значень із порядковими номерами від `0` до `255`).

Якщо в математиці для позначення множини використовуються фігурні дужки (наприклад, {2, 4, 6}), то в Delphi – квадратні дужки (наприклад, `[2, 4, 6]`) або ж пари складених символів `(.` та `.)`. Як і в математиці, у Delphi порядок елементів у множині може бути будь-яким. Так, записавши `[4, 2, 6]` або `[2, 4, 6]`, ми матимемо справу з однією і тією ж множиною. Більш того, багаторазове повторення одного і того самого елемента не змінює множини (наведену множину можна записати й так: `[2, 4, 2, 6]`).

За формою запису оголошення змінної типу «множина» є подібним оголошенню одновимірного масиву:

```
var
  ім'я: set of тип;
```

Наприклад, оголошення змінної `characters`, що розглядається як множина з базовим типом `Char`, має вигляд

```
var
  characters: set of Char;
```

На відміну від елементів масиву, елементи множини не впорядковані та не мають індексації.

Можна спочатку оголосити тип множини, а потім використати його для оголошення змінних:

```
type
  t_CharSet = set of Char;
var
  CharSet1, CharSet2: t_CharSet;
```

Досить часто як базовий тип множини вживається перелічений тип або деякий його діапазон:

```
type
  week_days = (Monday, Tuesday, Wednesday,
               Thursday, Friday);
var
  Workdays: set of week_days;
  symbols: set of 'A'..'Z';
```

Підкреслимо, що оголошення змінної-множини не надає їй значення.

5.3.2. Побудова множини

Для того щоб у множині з'явилися елементи, необхідно виконати оператор присвоювання, у лівій частині якого стоїть ім'я змінної-множини, а в правій – конструктор множини або деякий вираз над множинами.

Конструктор множини – це укладений у квадратні дужки перелік елементів, розділених комами, причому елементами можуть бути діапазони значень:

```
Workdays := [Monday, Wednesday, Thursday];
symbols := ['C', 'T'..'W', 'A'];
```

При задаванні множини порядок його елементів неважливий, але при задаванні відрізка такий порядок важливий. Тому оператор

```
symbols := ['Z'..'F']
```

не помістить у множину `symbols` елементи, бо в ньому відрізок заданий у зворотному порядку відносно його оголошення.

Множина, у якій немає елементів, називається ***порожньою*** (або інакше ***нуль-множиною***). Вона в Delphi позначається квадратними дужками, між якими не зазначені елементи:

```
Workdays := [];
```

Множину можна оголосити за допомогою типізованої константи, для чого в описі після знака рівності треба вказати конструктор множини. Наприклад, оператор

```
const
  RusLetters: set of ['А'..'Я'] =
       ['А', 'Е', 'И', 'О', 'У', 'Ы', 'Э', 'Ю', 'Я'];
```

описує множину, елементами якої можуть бути великі російські літери, з записом у неї початкового значення, що є множиною великих голосних російських літер.

Конструюючи множини, можна використовувати і змінні за умови, що їхні поточні значення потрапляють у діапазон базового типу множини. Так, якщо `ch1` та `ch2` мають тип `Char`, то припустима наступна послідовність операторів:

```
ch1 := 'S';
ch2 := 'A';
symbols := [ch1, 'F', ch2];
```

Тут результуючою множиною буде `['A', 'F', 'S']`.

Результат обчислення множинного виразу повинен бути сумісним за присвоюванням зі змінною, що стоїть ліворуч в операторі присвоювання.

Елементи множини не можна вводити і виводити звичайним чином. Для організації введення/виведення елементів множини треба застосовувати допоміжні змінні, як це зроблено в програмі, наведеній в прикладі 5.14. У той же час можна використовувати множини як елементи типізованих файлів.

5.3.3. Дії над множинами

Об'єднання, перетин і різниця множин. Над множинами здійснені, насамперед, такі три операції: об'єднання (+), перетин (*) і різниця (−).

Об'єднання двох множин A і B (A+B) – це нова множина, що складається з елементів, які належать одній з множин A і B або тій та інший одночасно:

```
var
  CharSet1, CharSet2, CharSet3: set of Char;
        //...
  CharSet1 := ['a', 'x', 'e'];
  CharSet2 := ['u', 'x'];
  CharSet3 := CharSet1 + CharSet2 + ['s', 'w'];
       //Тут CharSet3 = ['a', 'e', 's', 'u', 'x']
```

Перетин двох множин A і B (A*B) – це множина, що складається з елементів, які одночасно належать множинам A і B:

```
CharSet3 := CharSet1 * CharSet2;    //CharSet3 = ['x']
CharSet1 := CharSet1 * ['n', 'd'];     //CharSet1 = []
```

Різниця двох множин A і B (A−B) – це нова множина, що складається з елементів множини A, які не ввійшли в множину B:

```
CharSet1 := ['a', 'e', 't'];
CharSet2 := CharSet1 - ['e'];  //CharSet2 = ['a', 't']
CharSet3 := ['s', 'e', 't'] - CharSet1;          //Тут
                                      //CharSet3 = ['s']
CharSet2 := CharSet1 - ['s', 'e', 't'];          //Тут
                                      //CharSet2 = ['a']
```

Маніпулюючи операціями над множинами, можна додавати елементи до множин або видаляти їх.

Для вставки і видалення елементів при роботі з множинами в Delphi введено дві процедури:

Include(ім'я_множини, елемент) – вставити елемент;
Exclude(ім'я_множини, елемент) – видалити елемент.

Перша з них забезпечує додавання одного елемента в зазначену множину, а друга – видалення:

```
Include(CharSet1, 'g'); //Можна CharSet:=CharSet+['g']
Exclude(CharSet1, 't'); //Можна CharSet:=CharSet-['t']
```

Ці процедури дають більш ефективний код порівняно з операціями «+» і «−».

Інші операції над множинами. Пріоритет операцій. Крім розглянутих вище дій, над множинами можна виконувати чотири операції порівняння: =, <>, <=, >=.

Дві множини A і B ***рівні*** (A = B), якщо кожен елемент множини A є також елементом множини B і навпаки.

Дві множини A і B ***не рівні*** (A <> B), якщо вони відрізняються хоча б одним елементом.

Множина A є ***підмножиною*** множини B (A <= B або B >= A), якщо кожен елемент з A присутній і в B.

Є також можливість з'ясувати, чи належить даний елемент деякій множині. Для цього служить операція **in**. Нехай A – множина елементів деякого базового типу, а x – змінна (константа, вираз) цього типу. Тоді вираз x **in** A істинний, якщо значення x є елементом множини A.

Всі операції порівняння множин, а також операція **in** виробляють значення True або False.

У складних виразах над множинами операції виконуються відповідно до такої системи пріоритетів:

1 (вищий пріоритет)	*
2	+, –
3 (нижчий пріоритет)	=, <>, <=, >=, **in**

Операції одного пріоритетного рівня виконуються зліва направо (якщо не використані дужки).

У наступному прикладі показано, як описується типізована константа-множина, проілюстровані перевірка приналежності множині й застосування операції об'єднання множин.

```
//Приклад 5.15
//Уводиться рядок. Чи є він ім'ям відповідно до правил
//мови Delphi? Ім'я може починатися тільки з літери або
//символу підкреслення і містити літери, цифри та символ
//підкреслення.
```

Для розв'язання задачі скористаємося без змін формою з прикладу 5.6, описавши в секції **private** класу TForm1 поле s типу **string**. Задача розв'язується таким опрацьовувачем події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
type
  t_set = set of Char;
const
  Letters: t_set = ['a'..'z', 'A'..'Z', '_'];
  Ciphers: t_set = ['0'..'9'];
```

```
var
  i: Byte;
  flag: Boolean;
begin
  s := edInput1.Text;
  mmOutput1.Lines.Add('Початковий рядок:');
  mmOutput1.Lines.Add(s);
  flag := True;
      //Перевірка першого символу
  if not (s[1] in Letters) then
    flag := False;
  //Перевірка допустимості всіх інших символів рядка
  i := 2;
  while (i <= Length(s)) and flag do
  begin
    if not (s[i] in Letters + Ciphers) then
    begin
      flag := False;
      Break;
    end;
    i := i + 1;
  end;
  if flag then
    mmOutput1.Lines.Add('Так')
  else
    mmOutput1.Lines.Add('Ні');
  Button1.Visible := False;
  edInput1.Visible := False;
  lbOutput1.Visible := False;
end;
```

Використання операцій перетину та різниці множин, а також введення/виведення вмісту множини ілюструє опрацьовувач події OnClick кнопки, що наведений нижче.

```
//Приклад 5.16
//Лотерея "5 з 36": угадати 5 загаданих чисел від 1 до 36.
//При програші надрукувати, які числа вгадані, які
//не вгадані, які уведені неправильно.
```

Для розв'язання задачі скористаємося без змін формою з прикладу 5.6. У припущенні, що дані (відгадуванні числа) вводяться одним рядком, який містить 5 розділених пробілом неповторюваних цілих чисел від 1 до 36, задача розв'язується таким опрацьовувачем події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
type
  T_1_36 = 1..36;              //Діапазон чисел, що загадуються
```

```
var
  SInp, SOut: string;
  number: T_1_36;
  n: 0..5;
  comput,                              //Множина загаданих чисел
  player,                             //Множина чисел, що названі
  set1: set of T_1_36;                          //Допоміжна множина
begin
  n := 0;
  Randomize;
  comput := [];                                    //Порожня множина
                    //Генеруємо 5 випадкових чисел від 1 до 36
  repeat
    number := Random(36) + 1;
    if not (number in comput) then begin          //Якщо число...
      n := n + 1;   //...numb не входить в comput, додати його
      Include(comput, number)  //Можна comput:=comput+[number]
    end
  until n = 5;
  player := [];
  SInp := Trim(edInput1.Text);
  mmOutput1.Lines.Add('Уведені дані:');
  mmOutput1.Lines.Add(edInput1.Text);
  for n := 1 to 5 do begin
    SInp := SInp + ' ';
  //Наступні два оператори "вводять" елемент у множину
    number := StrToInt(Copy(SInp, 1, Pos(' ', SInp) - 1));
    Include(player, number);     //Можна обійтися і без number
    Delete(SInp, 1, Pos(' ', SInp));
    SInp := Trim(SInp);
  end;
      //Нижче ілюструються операції над множинами
  if player = comput then                      //Порівняння множин
    mmOutput1.Lines.Add('Вітаю, Ви виграли!')
  else begin
    mmOutput1.Lines.Add('На жаль, Ви програли!');
    for n := 1 to 3 do begin
      case n of
       1: begin
            SOut := 'Вгадані числа: ';
            set1 := comput * player;           //Перетин множин
          end;
       2: begin
            SOut := 'Не вгадані числа: ';
            set1 := comput - player;           //Різниця множин
          end;
       3: begin
            SOut := 'Помилкові числа: ';
```

```
            set1 := player - comput;           //Різниця множин
          end;
      end;
                  //Дописуємо до рядка SOut вмісту множини set1
      for number := 1 to 36 do      //Можливі елементи множини
        if number in set1 then       //Перевірка приналежності
          SOut := SOut + ' ' + IntToStr(number);
      mmOutput1.Lines.Add(SOut);
    end;
  end;
  Button1.Visible := False;
  edInput1.Visible := False;
  lbOutput1.Visible := False;
end;
```

У даному прикладі показано, що замість введення даних у множину здійснюється введення в допоміжну змінну `number` з наступним включенням введеного значення в множину (за допомогою `Include`). Природно, звертання до функції `StrToInt` можна було задати як перший параметр при звертанні до процедури `Include`. Виведення вмісту множини організують у циклі за всіма значеннями базового типу з виведенням тільки тих з них, які належать множині.

Можна рекомендувати самостійно переписати програму, організувавши послідовне введення відгадуваних чисел з виключенням спроби запису в множину раніше включених у нього значень.

5.4. Записи

5.4.1. Звичайні записи

Розглянемо таку задачу. Є інформація про 100 студентів, що включає прізвища та середній бал сесії. Необхідно надрукувати прізвища студентів, успішність яких вища за середню.

У цій задачі інформація про кожного зі студентів складається з даних двох типів – рядкового (прізвище) і числового (середній бал). При написанні програми можна скористатися двома одновимірними масивами, які повинні оброблятися спільно. Це незручно, особливо у випадках, коли подібних характеристик не дві, а значно більше. Багатовимірні масиви тут допомогти не можуть, оскільки кожна з характеристик має свій тип, а масиви поєднують у собі тільки однотипні елементи.

У Delphi є спеціальний комбінований тип даних, що дозволяє поєднувати відразу кілька типів. Такий тип даних називають записом. Формат опису змінної типу «запис» наступний:

```
var
  ім'я_запису: record
                 ім'я1: тип1;
                 ім'я2:тип2;
       ...
                 ім'яN:типN;
               end;
```

Частини запису, призначені для безпосереднього зберігання інформації (ім'я1, ..., ім'яN), називаються полями запису. Для кожного з полів повинні бути зазначені ім'я та тип. Якщо кілька полів мають один тип, то їх можна оголосити спільно, перелічивши імена через кому.

Для сформульованої задачі можна, наприклад, оголосити таку змінну типу «запис»:

```
var
  stud: record
          surname: string[20];              //Прізвище
          AverageEval: Real;          //Середня оцінка
        end;
```

Підкреслимо, що опис запису завершується службовим словом **end**, перед яким можна ставити, а можна не ставити крапку з комою.

Рекомендується спочатку оголосити ім'я типу запису, яке надалі буде використовуватися при описі змінних:

```
type
  T_stud = record
             surname: string;
             AverageEval: Real;
  end;
var
  stud: T_stud;
```

Щоб звернутися до поля запису, необхідно не просто вказати ім'я поля, але перед ним зазначити ім'я запису, з'єднавши їх крапкою:

```
stud.surname := 'Супченко';
stud.AverageEval := 4.8;
```

Альтернативою такому зверненню є звертання з використанням оператора приєднання **with**:

```
with ім'я_запису do оператор;
```

Оператор **with** автоматично приєднує зазначене в ньому ім'я_запису до всіх імен, які входять в оператор, що стоїть після **with**, і збігаються з іменами полів запису. До імен, що не збігаються з іменами

полів запису, приєднання не проводиться. Якщо приєднання потрібно виконати в декількох операторах, то їх за допомогою **begin-end** поєднують у складений оператор. Наприклад, наведені вище оператори можна записати так:

```
with stud do begin
  surname := AnsiUpperCase(surname);
  AverageEval := 0;
end;
```

При оголошенні типізованої константи записного типу її значення задається в такий спосіб: у спільних круглих дужках зазначають імена полів, після кожного з яких через двокрапку подається початкове значення; роздільником є символ «крапка з комою»:

```
const
  stud1:T_stud=(surname: 'Іваненко';AverageEval:0.0);
```

Записи можуть бути елементами масиву, як це має місце у наведеному нижче прикладі.

```
//Приклад 5.17
//Вивести прізвища тих з n студентів, чий середній бал вищий
//від загального середнього бала.
```

Для розв'язання задачі скористаємося формою з прикладу 2.2, помістивши на панелі додатково однорядковий редактор edInput2 з очищеною властивістю Caption і мітку lbOutput2 із записаним у властивість Caption текстом Середній бал. Крім того, у властивість Caption мітки lbOutput1 запишемо текст Уведіть кількість студентів, а у властивість кнопки – текст Введення. У розділі опису типів інтерфейсної частини модуля перед описом класу TForm1 опишемо наступний записний тип:

```
TStud_rec = record
  surname: string;
  AverageEval: Real;
end;
```

У секції **private** опису класу TForm1 опишемо такі поля:

```
Count: Integer;
stud_array: array of TStud_rec;
```

а слідом за описом форми (Form1: TForm1) розмістимо опис змінної c для підрахунку кількості студентів (тут демонструється ініціалізація):

```
c: Integer = 0;
```

Для розв'язання задачі можливе застосування таких двох опрацьовувачів подій:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  sum, ArithmeticMean: Real;
  i: Integer;
begin
  if Count = 0 then begin
    Count := StrToInt(edInput1.Text);
    SetLength(stud_array, Count);
    mmOutput1.Lines.Add('Усього студентів '
                        + IntToStr(Count));
    lbOutput1.Caption := 'Уведіть прізвище';
    lbOutput2.Visible := True;
    edInput2.Visible := True;
    edInput1.SetFocus;
  end
  else begin
                  //Нижче використовується оператор приєднання
    with stud_array[c] do begin
      surname := edInput1.Text;
      AverageEval := StrToFloat(edInput2.Text);
      mmOutput1.Lines.Add(surname + ' '
                          + FloatToStr(AverageEval));
      Inc(c);
    end;
    if c < Count then
      edInput1.SetFocus
    else begin
      sum := 0;
      for i := 0 to Count - 1 do
        sum := sum + stud_array[i].AverageEval;
      ArithmeticMean := sum / Count;
      mmOutput1.Lines.Add('Результат:');
      for i := 0 to Count - 1 do
        if stud_array[i].AverageEval > ArithmeticMean then
          mmOutput1.Lines.Add(stud_array[i].surname);
      Button1.Visible := False;
      edInput1.Visible := False;
      lbOutput1.Visible := False;
      edInput2.Visible := False;
      lbOutput2.Visible := False;
    end;
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Count := 0;
end;
```

У програмі показано, як можна використати оператор **with** (до імені sum приєднання не буде здійснюватися, оскільки це не ім'я поля запису stud_array[c]) і як звертатися до поля запису безпосередньо (в операторах циклу **for**).

Поля запису можуть мати будь-який тип. Наприклад, як поле запису можливий масив

```
type
  session_rec = record
    AverageEval: Real;
    Eval: array[1..5] of 2..5;
  end;
```

або запис

```
type
  stud_rec1 = record
    surname: string;
    session: session_rec;
  end;
var
  report: array[1..100] of stud_rec1;
```

В останньому випадку для звертання до полів запису доводиться будувати цілий ланцюжок приєднань.

Можна також указати через кому список імен, що приєднуються, в операторі **with**. При цьому приєднання починається з імені, що стоїть правіше в списку. Так, наступні чотири оператори еквівалентні:

1) `report[17].session.Eval[3] := 4;`
2) `with report[17], session do Eval[3] := 4;`
3)
```
with report[17] do
   with session do Eval[3] := 4;
```
4) `with report[17].session do Eval[3] := 4;`

Можливість використання записів як полів інших записів дозволяє організовувати оперування складними структурами різнорідних даних.

Щоб оголосити типізовану константу записного типу, необхідно після імені константи слідом за знаком рівності в круглих дужках перелічити імена полів, зв'язавши їх зі значеннями двокрапкою. При цьому як роздільник вживається крапка з комою. Якщо полем запису є масив або запис, то задавання значення для такого поля виконується у своїх круглих дужках.

Це ж має місце й у випадку типізованих констант – масивів з елементами записного типу.

Наприклад:

```
type
  session_rec = record
    AverageEval: Real;
    Eval: array[1..5] of 2..5;
  end;
  Variants = array [1..3] of session_rec;
const
  Etalon = (AverageEval: 5.0; Eval = (5, 5, 5, 5, 5));
  Variants: TVariants =
          ((AverageEval: 2.0; Eval = (2, 2, 2, 2, 2)),
           (AverageEval: 3.0; Eval = (3, 3, 3, 3, 3)),
           (AverageEval: 4.0; Eval = (4, 4, 4, 4, 4)));
```

5.4.2. Записи з варіантами

Нехай тепер необхідно організувати обробку інформації про книги (автори, назва книги, місце видання, назва видавництва, рік видання) і журнальні статті (автори, назва статті, журнал, номер, рік видання). Щоб в одному масиві зосередити інформацію про такі різнорідні записи, потрібно описати в одному записі всі можливі поля і якимось чином обмежити доступ до тих з полів, які не вживаються в конкретному запису. Щоб позбутися від цього, в Delphi є можливість використання записів з варіантами, ідея яких полягає у визначенні в рамках однієї структури декількох різних (альтернативних) записів.

Запис з варіантами складається з двох частин – фіксованої, у якій перелічуються спільні для всіх альтернатив поля, і змінної (варіантної), структура якої змінюється залежно від значення особливого поля, так званого поля тегу (ознаки). Для поставленої вище задачі можна запропонувати таку структуру запису:

```
type
  t_catalog = record
    author: string[20];
    title: string[100];
    year: Integer;
    case tag: Byte of
     0: (town: string[20];
         publishing: string[20]);
     1: (journal: string[20]; num: Byte);
  end;
var
  catalog: array[1..100] of t_catalog;
```

Особливості записів з варіантами:

- варіантна частина може бути тільки одна, і вона повинна бути останньою;
- варіантна частина подібна з оператором **case**, у якого відсутнє службове слово **end**; замість **end** для **case** використовується **end** для **record**;
- полем тегу є ім'я (власне тег) і тип, що стоять за **case**;
- поле тегу може складатися тільки з типу без вказівки імені;
- тег може мати тільки дискретний тип;
- тег входить у фіксовану частину запису і завершує її;
- опис полів, що стосуються окремих значень тегу, беруть у круглі дужки;
- якщо варіант для якого-небудь значення тегу не потрібен, то він задається порожнім полем, укладеним у круглі дужки;
- якщо яке-небудь з полів у варіантній частині – це теж варіантний запис, то воно повинне бути останньою частиною варіанта.

При оголошенні типізованих констант як записів з варіантами подають значення тільки одного з варіантів.

Варіантні записи, полегшуючи оперування даними, у той же час збільшують обсяг використованої пам'яті, оскільки під такого роду змінні завжди виділяється пам'ять, необхідна для зберігання найдовшого запису. Контроль за тим, який з варіантів реалізований у даній змінній, повністю покладається на програміста (можна, наприклад, записати в деяку змінну `catalog[i]` інформацію про книгу, а обробляти її як журнальну статтю). Тому при роботі з варіантними записами треба завжди враховувати значення тегу.

```
//Приклад 5.18
//Є каталог на n книг і журнальних статей (n<=100).
//Вивести інформацію про публікації, видані після
//1990 р. (окремо книги та статті).
```

У зв'язку з тим, що при введенні даних з клавіатури в цій задачі потрібне написання громіздкої програми, а файли ще нами не розглядалися, повний текст опрацьовувачів подій, як і опис форми, наводитися не будуть. Розглянемо тільки фрагмент програми, пов'язаний з обробкою даних і виведенням їх у багаторядковий редактор `mmOutput1`. Будемо вважати, що в модулі перед описом типу форми виконано такий опис типу варіантного запису:

```
type
  T_publication = record
    author: string[20];
```

```
    title: string[100];
    year: Integer;
    case key: Char of
     'B', 'b': (town, publishing: string[20]);       //book
     'A', 'a': (journal: string[50]; num: Byte);  //article
  end;
```

Крім того, вважатимемо, що разом з формою описані змінна `CountRec` типу `Integer` (кількість записів у каталозі) і змінна `catalog` типу **`array of`** `T_publication` (динамічний масив для зберігання інформації про книги та журнальні статті).

У припущенні, що масив `catalog` створений (наприклад, оператором `SetLength(catalog, CountRec)`) і має `CountRec` елементів, у які записані дані, а також що усередині опрацьовувача подій описана локальна змінна `i` типу `Integer`, виведення даних можна організувати за допомогою такої послідовності операторів:

```
mmOutput1.Lines.Add('Інформація про книги');
for i := 0 to CountRec - 1 do
  with catalog[i] do
    if (year > 1990) and (key in ['B', 'b']) then
      mmOutput1.Lines.Add(author + '. ' + title + '.-' +
      town + ': ' + publishing + '.-' + IntToStr(year));
mmOutput1.Lines.Add('Інформація про журнальні статті');
for i := 0 to CountRec - 1 do
  with catalog[i] do
    if (year > 1990) and (key in ['A', 'a']) then
      mmOutput1.Lines.Add(author + '. ' + title + '// ' +
                          journal + '. ' + IntToStr(year)
                          + '. N ' + IntToStr(num));
```

При читанні даних повинні бути прочитані всі поля, що стосуються конкретного джерела інформації, причому в обов'язковому порядку має бути введена інформація, яка дозволяє розрізнити джерело інформації (у цьому випадку поле `key`). При виведенні поле `key` використовується для впізнання запису: що це – книга або стаття? Природно, вводити подібного роду дані з клавіатури недоцільно і практично нереально, а при великих значеннях `n` неможливо. У цьому випадку на допомогу приходять файли.

Запитання для контролю і самоконтролю

1. Що таке масив?
2. Як описується одновимірний масив у Delphi?
3. Чи можна звернутися до елементів поза масивом?

4. Як описується динамічний масив у програмах, написаних мовою Delphi?
5. Як здійснюється створення і знищення динамічних масивів?
6. Що таке багатовимірний масив і як він описується?
7. Чи можна використовувати в програмах масиви розмірності більш двох?
8. Що неправильно в наступному фрагменті програми?

```
var
  i, j: Integer;
  myArray: array[1..5, 1..4] of Integer;
       //Продовження тексту
for i := 1 to 4 do
  for j := 1 to 5 do
    myArray[i, j] := i + j;
```

9. Для чого служить директива компілятору `$R`?
10. Що таке рядкові дані?
11. Які рядкові типи існують у Delphi?
12. У чому особливості коротких рядків?
13. У чому особливості рядків типу `AnsiString`?
14. Які рядки за умовчанням ставляться у відповідність типу **`string`**?
15. Як можна впізнати фактичну довжину рядка?
16. Як проводиться порівняння даних типу **`string`**?
17. Назвіть процедури і функції для роботи з рядками.
18. У чому особливість даних типу `PChar`?
19. Як оголошується множина в Delphi?
20. Які дії можуть виконуватися над множинами в Delphi?
21. Для чого використовуються записи?
22. У чому відмінність звичайних записів від варіантних?

Завдання для практичного відпрацьовування матеріалу

1. Дано натуральне число n ($n \le 15$) і дійсні числа $a_1, a_2, \ldots, a_n$, $b_1, b_2, \ldots, b_n$. Обчислити $(a_1 + b_n) \cdot (a_2 + b_{n-1}) \cdot \ldots \cdot (a_n + b_1)$.
2. Дано ціле число n ($n \le 15$). Скільки різних цифр у його десятковому записі?
3. Дано натуральне число n та послідовність відмінних від нуля цілих чисел $a_1, \ldots, a_n$. Якщо в послідовності числа з різними знаками чергу-

ються, вивести початкову послідовність. У противному разі вивести всі додатні члени послідовності, зберігши порядок їх проходження.

4. Дано натуральне число n та масив із n цілих чисел ($n \le 100$). Чи є в цьому масиві частина із послідовно розташованих елементів, сума елементів якої дорівнює сумі тих елементів, що залишилися? При позитивній відповіді вказати номери першого й останнього елементів знайденої частини послідовності.
5. Дано натуральне число n та цілі числа $a_1, a_2, \ldots, a_n$. З'ясувати, чи є серед чисел $a_1, a_2, \ldots, a_n$ збіжні.
6. Дано натуральне число n та цілі числа $a_1, a_2, \ldots, a_{3n}$. З'ясувати, чи правильно, що для всіх $a_{n+1}, a_{n+2}, \ldots, a_{3n}$ є рівні серед $a_1, a_2, \ldots, a_n$.
7. Дано натуральне число n та послідовність дійсних чисел $a_1, \ldots, a_n$. Упорядкувати цю послідовність за незростанням. Скористатися наступним методом. Знайти елемент масиву, що має найменше значення, і поміняти місцями його з першим елементом, потім серед елементів масиву, починаючи з другого, знайти найменший і поміняти місцями його з другим і т. д.
8. Дано дійсну квадратну матрицю порядку n ($n \le 15$). Замінити в ній нулями всі елементи, що розташовані на головній діагоналі і вище неї.
9. Дано дійсну матрицю розміру $m \times n$ ($m, n \le 15$). Знайти суму найменших значень елементів її стовпців.
10. Дано дійсну квадратну матрицю порядку n ($n \le 15$), натуральні числа p, q $(1 \le p \le n, 1 \le q \le n)$. Видалити з матриці p-й рядок та q-й стовпець.
11. Заповнити цілочисловий масив розміру $n \times n$ ($n \le 15$) числами 1, 2, …, n^2 за спіраллю, що скручується за годинниковою стрілкою, виходячи з верхнього лівого кута масиву.
12. Дано дійсний квадратний масив розміру $n \times n$. Знайти найбільше зі значень елементів, розташованих у зафарбованій частині масиву (рис. 5.2).

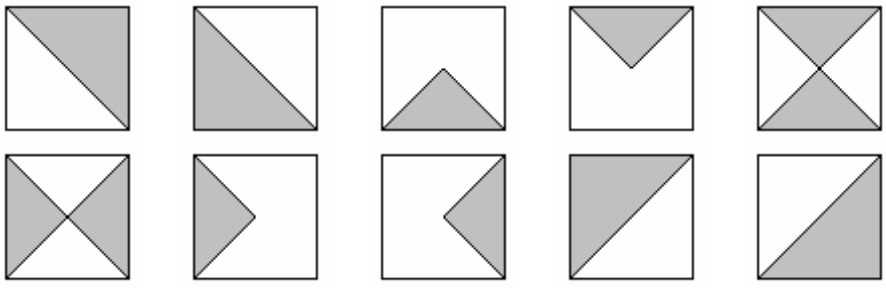


Рис. 5.2. Варіанти частин масивів, що розглядаються у завданні 12

13. Дано прямокутний паралелепіпед з довжинами сторін m, n, k, що задаються натуральними числами. Паралелепіпед утворений з різноко-

лірних елементарних кубиків з ребрами довжиною 1. Чи можна прострелити паралелепіпед трьома спицями, що 1) перетинаються, 2) перпендикулярні різним граням паралелепіпеда і 3) проходять через кубики тільки одного кольору? При позитивній відповіді на запитання, вказати всі різні варіанти проходження спиць.

14. Дано рядок s, що містить n символів $s_1, \ldots, s_n$. Якщо цей рядок є паліндромом, тобто $s_1 = s_n$, $s_2 = s_{n-1}, \ldots,$ то залишити його без зміни, у противному разі доповнити рядок праворуч його «дзеркальним» відбиттям без повторення останнього символу $(s_1, s_2, \ldots, s_{n-1}, s_n, s_{n-1}, \ldots, s_2, s_1)$.
15. Дано рядок. Видалити в ньому найдовший підрядок, що складається тільки з цифр.
16. Дано рядки символів $s1$ та $s2$. Приєднати рядок $s2$ праворуч до рядка $s1$. Якщо початок рядка $s2$ збігається з кінцем рядка $s1$, при приєднанні рядків виключити таке накладення символів, видаливши найбільшу їхню кількість. Наприклад, з рядків *абракадабра* і *брахман* повинен бути отриманий рядок *абракадабрахман*.
17. Дано рядки символів $s1$ та $s2$. Перелічити ті символи, які
    а) містяться в $s1$, але не містяться в $s2$;
    б) містяться і в $s1$, і в $s2$.
18. Дано натуральне число n. Підрахувати кількість різних цифр у його десятковому записі, скориставшись типом даних «множина».
19. Дано рядок. Вивести всі його символи без повторення.
20. Дано рядки символів $s1$ та $s2$.
    а) Які символи одночасно входять в обидва ці рядки?
    б) Які символи входять хоча б в один з рядків $s1$ та $s2$?
    в) Які символи входять у рядок $s1$, але не входять в $s2$?
    г) Які символи входять в $s2$, але не входять в $s1$?
21. Дано рядок s. Вивести всі перші входження в нього великих і малих латинських літер.
22. Дано масив з 10 дат (число, місяць, рік). Зазначити всі весняні дати, а також найпізнішу дату.

6. ФАЙЛИ

6.1. Загальні принципи роботи з файлами

Введення даних з клавіатури і виведення їх на екран не завжди зручні: часто більш доцільним є використання підготовленого заздалегідь набору даних, що зберігається у файлі, і виведення даних у файл з метою подальшого використання.

Стандартними діями, які необхідно виконати для забезпечення можливості роботи з файлом даних, є такі:

1. Опис файлової змінної.
2. Зв'язування файлової змінної з ім'ям файлу (*ініціалізація* файлу).
3. Відкриття файлу.

За формою опису файлових змінних у Delphi розглядають три види файлів:

- текстові файли (їх файлові змінні описуються типом `TextFile`);
- типізовані файли (їх файлові змінні описуються типами виду **`file of`** тип);
- нетипізовані файли (файлові змінні мають тип **`file`**).

Файлова змінна описується звичайним чином:

```
var
  файлова_змінна: файловий_тип;
```

Особливістю файлових змінних усіх файлів є те, що з ними не може працювати оператор присвоювання.

Для роботи з файлом одного оголошення недостатньо, бо оголошена файлова змінна не є самим файлом або хоча б його ім'ям, відомим операційній системі. Це лише внутрішнє ім'я файлу, з яким працює програма. Але програмі повинні бути відомі місце розташування файлу (ім'я дисководу та каталог), а також ім'я та розширення, з якими він записаний на диску. Ці відомості повідомляються за допомогою оператора `AssignFile` (призначити файл), що має такий формат:

AssignFile(файлова_змінна, зовнішнє_ім'я_файлу).

Цей оператор зв'язує ім'я (ідентифікатор) файлової змінної, що фігурує в програмі, із зовнішнім ім'ям файлу, яке задається у вигляді рядка (константи, змінної, виразу). Наприклад, якщо в програмі є опис вигляду

```
var
  f: TextFile;
```

то оператор

```
AssignFile(f, 'a.txt')
```

зв'язує файлову змінну f з файлом a.txt, розташованим у поточному каталозі активного диска.

Для роботи із вмістом файлу недостатньо зв'язати внутрішнє ім'я файлу з зовнішнім – після виконання оператора AssignFile і до першої операції читання з файлу або запису у файл останній повинен бути відкритий. Для читання даних файл відкривається процедурою

Reset(файлова_змінна),

а для запису даних у файл відкриття здійснюють процедурою

Rewrite(файлова_змінна).

Зазначимо, що остання фраза є правильною, тільки тоді якщо мова йде про текстові файли.

Наприклад, для змінної f ці оператори виглядають так:

```
Reset(f)
```

і

```
Rewrite(f).
```

Виконання кожного з цих операторів позиціює файл на позицію його першого символу. Тому відкриття існуючого файлу за допомогою Rewrite забезпечує його повне відновлення.

Якщо при читанні з файлу буде прочитаний символ кінця файлу (#26), то виникне помилка часу виконання, для виключення якої необхідно перед читанням перевіряти, чи немає позиціювання на кінець файлу. Для цього в Delphi уведено функцію

Eof(файлова_змінна),

що повертає значення True, якщо в момент звертання до неї виявлений кінець файлу, і False – у противному разі.

По закінченні роботи з файлом його закривають оператором

CloseFile(файлова_змінна),

який не є обов'язковим, оскільки при завершенні виконання програми всі файли закриваються автоматично. Повторне відкриття файлу без його закриття супроводжується попереднім автоматичним закриттям.

Слід зазначити, що самі по собі файли є просто послідовністю байтів. Тому можливі запис вмісту файлу в одному форматі, а читання в іншому, що в більшості випадків є неправильним (наприклад, файл створювався як типізований, а перед обробкою його вмісту зв'язується з файловою змінною текстового файлу). Дії щодо узгодження вмісту файлу і вибору типу файлової змінної для забезпечення доступу до нього покладають на програміста. Потрібно знати, у якому форматі записувалися дані у файл, і користуватися цією інформацією.

Щоб уникнути помилок, пов'язаних зі спробою відкриття за допомогою `Reset` відсутнього файлу, а також з метою запобігання можливого помилкового видалення існуючого файлу при відкритті за допомогою процедури `Rewrite` рекомендується попередньо перевіряти наявність файлу, для чого можливе використання функції

`FileExists(`ім'я_файлу`)`,

що повертає значення `True`, якщо файл, ім'я якого задане як параметр (тип **`string`**) існує, і значення `False` за відсутності файлу. Ця функція не перевіряє наявність каталогу.

Обмін даними між програмою і файлом проводиться не прямо, а через спеціальний системний буфер розміром 128 байт. При заповненні буфера йде його автоматичне звільнення з занесенням даних у файл. Оператор `CloseFile`, крім інших дій, переписує з буфера залишок даних.

Якщо файл не закритий, то, звернувшись до процедури

`Flush(`файлова_змінна`)`,

можна примусово переписати вміст буфера виведення в дисковий файл.

Крім звичайної методики доступу до вмісту файлу, існує також методика, що базується на понятті дескриптора файлу. Мова йде про те, що при відкритті файлу йому ставиться у відповідність цілочислове значення, яке називають ***дескриптором***. Дескриптор файлу в деякому розумінні є аналогічним номеру файлу.

У Delphi визначено дві функції, що дозволяють відкрити файл із заданим ім'ям з одночасним одержанням дескриптора – `FileCreate` та `FileOpen`.

Функція

`FileCreate(`ім'я_файлу`)`

створює файл із зазначеним ім'ям і повертає його дескриптор, а функція

`FileOpen(`ім'я_файлу, режим_відкриття`)`

відкриває існуючий файл із заданим ім'ям у заданому другим параметром режимі відкриття (тип `LongWord`), також повертаючи дескриптор файлу. Ось деякі з констант, що визначають режими відкриття:

```
fmOpenRead       = $0000;      //Відкриття існуючого файлу
                             //для запису або створення нового
fmOpenWrite      = $0001;   //Відкриття тільки для читання
fmOpenReadWrite  = $0002;    //Відкриття тільки для запису
fmShareCompat    = $0000; //Відкриття для читання і запису
```

При нормальному завершенні функції `FileCreate` та `FileOpen` повертають значення більші або рівні `0`, а при помилці – значення `-1`.

Закриття файлу із застосуванням його дескриптора здійснюється процедурою

`FileClose(`дескриптор`)`.

Зазначимо, що так звані низькорівневі функції відкриття файлів `FileCreate` та `FileOpen` довідкова система Delphi не рекомендує вживати в програмах, написаних мовою Delphi.

6.2. Текстові файли

Одним з видів файлів, якими може оперувати програма, написана мовою Delphi, є текстові файли, які складаються з символьних рядків змінної довжини, що завершуються спеціальною комбінацією, так званим «кінцем рядка». Комбінація «кінець рядка» складається з двох символів – «переведення каретки» (`#13`) і «переведення рядка» (`#10`). Завершується текстовий файл, як і інші файли, символом «кінець файлу» (`#26`). Такий файл можна підготувати будь-яким текстовим редактором, у тому числі редактором, убудованим в інтегроване середовище Delphi (звичайно, його можна створити і за допомогою програми, написаної мовою Delphi).

Опис текстового файлу здійснюється оголошенням змінної типу `TextFile`:

```
var
  файлова_змінна: TextFile;
```

Як це зазначалося раніше, після ініціалізації файлу його необхідно відкрити. При цьому для читання текстовий файл відкривається оператором `Reset`, а для запису – оператором `Rewrite`. Варто пам'ятати, що текстовий файл може бути відкритий тільки для читання або тільки для запису: у текстовий файл, що відкритий для читання, неможливо записати

дані, як і неможливе читання даних з текстового файлу, що відкритий для запису.

Для текстових файлів введено ще одну процедуру відкриття для запису – `Append`, яка має той же формат, що і `Rewrite`:

`Append(`файлова_змінна`)`

Відмінність її полягає в тому, що вона працює з існуючим файлом і позиціює файл на його кінець, у результаті чого не створюється новий файл, а дозаписуються дані у його кінець. Якщо ж за допомогою `Append` відкривається неіснуючий файл, процедура `Append` за аналогією з `Rewrite` створює новий файл.

Для текстового файлу можна створити буфер обміну в самій програмі, а не в системній пам'яті, для чого достатньо скористатися процедурою

`SetTextBuf(`файлова_змінна`,`ім'я_буфера`,`розмір_буфера`)`

Другий параметр у ній – це ім'я будь-якої змінної, під яку відведено обсяг пам'яті, не менший за значення третього параметра (у байтах). Наприклад,

```
SetTextBuf(f, buf, 200);
```

де `buf` – масив, що має такий опис:

```
var
  buf: array[1..200] of Char;
```

Третій параметр у цій процедурі можна опустити, залишивши розмір буфера таким, що дорівнює раніше встановленому (наприклад, установленому за умовчанням розміру в 128 байт). Установка розміру буфера проводиться для закритого файлу.

Читання з текстового файлу виконується операторами `Read` і `ReadLn`, у яких як перший параметр подають ім'я файлової змінної, а далі через кому зазначають змінні, в які здійснюється читання даних з файлу.

У текстовому файлі дані зберігаються в рядковому вигляді; однак, якщо елемент даних може бути перетворений у число, це перетворення проводиться автоматично при введенні в числові змінні. Елементи числових даних у рядках текстового файлу розділяються пробілами або символами табуляції (клавіша <Tab>). Якщо рядок файлу вичерпаний, а в операторі `Read` список введення, який складається з числових або символьних змінних, не вичерпався, то введення продовжується з наступного рядка. При введенні даних з текстового файлу в символьні змінні елементи даних не розділяються. Якщо в списку даних після числової змінної йде рядкова, то пробіл, який іде після числового значення у файлі, зчитується в рядок (це ж справедливо і при зчитуванні в символьну змінну). Треба

враховувати, що при зчитуванні даних у рядок пробіли сприймаються як значущі символи, і в рядок потрапляють усі дані до комбінації «кінець рядка» або ж (у випадку коротких рядків) до вичерпання максимальної довжини рядка. Тому рядкові дані повинні розташовуватися в текстовому файлі або наприкінці рядка, або в окремих рядках і читатися оператором `ReadLn`.

Відмінність операторів `Read` і `ReadLn` при читанні з текстових файлів полягає в тому, що оператор `ReadLn`, помістивши значення в останню змінну списку введення, переходить на початок наступного рядка, не зчитуючи дані, що залишилися в рядку, а оператор `Read` залишається готовим зчитувати дані з наступної позиції поточного рядка. Наприклад, якщо в текстовому файлі `f` є два рядки

```
1 -2
4
```

то два оператори

```
Read(f, m);
Read(f, n);
```

помістять у цілочислові змінні `m` та `n` відповідно значення `1` та `-2`, а два оператори

```
ReadLn(f, m);
ReadLn(f, n);
```

зчитають значення `1` та `4`.

При виведенні даних у текстовий файл використовуються оператори `Write` і `WriteLn`, першим параметром яких є ім'я файлу, а далі через кому іде список значень, що виводяться (констант, змінних, виразів). Відмінність їх полягає у тому, що по закінченні виведення оператор `WriteLn` позиціює файл на наступний рядок, забезпечуючи подальше виведення з нового рядка. Якщо кілька даних виводиться в один рядок текстового файлу, то вони ніяк не розділяються. Дані, що не є рядковими (числові, булівські), автоматично перетворяться до рядкового виду.

```
//Приклад 6.1
//У текстовому файлі f.txt через пробіл та <Enter> записані
//цілі числа. Переписати у файл f1.txt з файлу f.txt всі
//числа за винятком максимальних (у припущенні,
//що їх може бути декілька).
```

Для розв'язання задачі скористаємося формою з двома кнопками:

- Кнопка Button1:
  Caption — Виконати

- Кнопка Close:
  Kind — bkClose
  Name — bbClose

Напишемо для компонента Button1 такий опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f, f1: TextFile;                  //Оголошення файлових змінних
  a, max: Integer;            //Число та максимальне з усіх чисел
  flag: Boolean;
begin
  AssignFile(f, 'f.txt');         //Зв'язування файлової змінної
  Reset(f);           //з файлом і відкриття файлу f для читання
  flag := True;
  while not Eof(f) do begin              //Читати до кінця файлу
    Read(f, a);                       //Читання елемента файлу f
    if flag or (a > max) then max := a;
    flag := False;
  end;
  AssignFile(f1, 'f1.txt');       //Зв'язування файлової змінної
  Rewrite(f1);        //з файлом і відкриття файлу f1 для запису
  Reset(f);//Повторне відкриття файлу f для повторного читання
  while not Eof(f) do begin                    //і цикл читання
   Read(f, a);
   if a <> max then WriteLn(f1, a);//При виведенні пропускаємо
  end;                                        //максимальні елементи
  CloseFile(f1); CloseFile(f);                //Закриття файлів
  Button1.Enabled := False;
end;
```

У наведеній вище підпрограмі замість функції `Eof` краще використати функцію

`SeekEof(`ім'я_файлової_змінної`)`,

бо у випадку наявності наприкінці файлу `f.txt` порожніх рядків або пробілів вони сприймаються як значення `0`, у результаті чого цей `0` вже як число може потрапити у файл `f1.txt`.

Для текстових файлів введено ще три функції:

`Eoln(`файлова_змінна`)`,

`SeekEoln(`файлова_змінна`)`,

`SeekEof(`файлова_змінна`)`.

Перша з них перевіряє, досягнутий кінець рядка (`True`) чи ні (`False`) у файлі, ім'я якого задане як параметр; друга, працюючи аналогічно першій, пропускає при аналізі пробіли та символи табуляції;

третя функція працює аналогічно функції `Eof`, але при аналізі пропускає пробіли і символи табуляції.

Особливістю текстових файлів є те, що вони є файлами послідовного доступу – не можна прочитати який-небудь елемент текстового файлу, не прочитавши всі попередні елементи. Аналогічно не можна записувати інформацію в текстовий файл довільно, писати в нього можна тільки послідовно.

Трохи забігаючи вперед, зазначимо, що при роботі з компонентом Memo можливий простий метод доступу до текстових файлів з метою завантаження їхнього вмісту в компонент (точніше, у властивість Lines) або запису в створюваний новий текстовий файл вмісту властивості Lines. Для цього застосовуються методи (процедури) `LoadFromFile` і `SaveToFile` класу `TStrings`, призначені для занесення вмісту текстового файлу в список рядків та для запису в текстовий файл вмісту списку рядків (про це буде йти митися далі в підрозд. 12.3). Вони мають відповідно такі формати звернення:

```
LoadFromFile(ім'я_файлу);
SaveToFile(ім'я_файлу);
```

При цьому немає необхідності в спеціальних діях, що організують доступ до файлу. При записі в існуючий файл останній знищується.

Розглянемо найпростіший приклад читання вмісту текстового файлу з записом його в багаторядкове текстове поле та зворотного запису вмісту багаторядкового текстового поля в текстовий файл.

```
//Приклад 6.2
//Завантажити текстовий файл і після його редагування
//здійснити збереження в іншому текстовому файлі.
//Передбачити контроль наявності початкового файлу,
//а також неможливість запису у вже існуючий файл.
```

Скористаємося формою з прикладу 3.1, змінивши зміну властивості деяких з її компонентів:

- Мітка:
  Caption — Початковий файл
- Багаторядкове поле:
  Lines — очистити
  Name — mmInpOut1
- Кнопка Button1:
  Caption — Завантажити

Напишемо для компонента Button1 такий опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Tag = 0 then begin
    if FileExists(edInput1.Text) then begin
      mmInpOut1.Lines.LoadFromFile(edInput1.Text);
      lbOutput1.Caption := 'Результуючий файл';
      Button1.Caption := 'Зберегти';
      Tag := 1;
    end;
    edInput1.SetFocus;
  end
  else begin
    if FileExists(edInput1.Text) then begin
      MessageBeep(MB_OK);
      edInput1.SetFocus;
    end
    else begin
      mmInpOut1.Lines.SaveToFile(edInput1.Text);
      Button1.Enabled := False;
    end;
  end;
end;
```

У наведеній вище процедурі для організації переключення режиму роботи компонента Button1 використано властивість `Tag`, що є притаманною всім компонентам (у даному випадку мова йде про форму). Ця властивість має тип `Integer` і може застосовуватися програмістом на власний розсуд (за умовчанням вона дорівнює `0`). На початку роботи програми виконується умова `Tag = 0`, і клік мишкою над компонентом Button1 забезпечує читання файлу, ім'я якого задане у властивості `edInput1.Text`. При цьому за допомогою функції `FileExists` перевіряється наявність файлу, і завантаження файлу виконується тільки за його наявності. Якщо файл завантажений, то змінюються властивості деяких компонентів, а властивість `Tag` дістає значення `1`, що відповідає режиму запису файлу при кліку мишкою над компонентом Button1. Після редагування вмісту вікна багаторядкового редактора він може бути збережений у текстовому файлі після введення імені файлу у вікні однорядкового редактора. Перед збереженням перевіряється наявність файлу з введеним ім'ям. Якщо такий файл уже існує, видається звуковий сигнал, а запис не проводиться (відзначимо, що можна було б організувати циклічне введення імені результуючого файлу). У противному випадку вміст редакторського вікна переписується в текстовий файл, а компонент Button1 стає недоступним для виключення повторного виконання процесів завантаження і збереження файлу.

6.3. Опрацювання помилок введення/виведення

За умовчанням в Delphi виконується автоматична перевірка на наявність помилок введення/виведення з видачею діагностичного повідомлення. Цей контроль можна відключити, для чого служить директива компілятору `$I`:

`{$I+}` – автоматичний контроль введення/виведення включений;

`{$I-}` – автоматичний контроль введення/виведення відключений.

Для обробки помилок введення/виведення користувачем директиву `$I` необхідно встановити в пасивний стан `{$I-}` і відразу ж після виконання операції введення/виведення звернутися до функції `IOResult`, що не має параметрів. Якщо помилки введення/виведення немає, то функція повертає нульове значення; при виникненні ж помилки повертається цілочислове значення – код помилки, і програма продовжує виконуватися (якщо це можливо).

Загалом кажучи, коди помилок розбиті на два діапазони: 0–99 (помилки Win32) і 100–149 (помилки VCL).

З першого діапазону звичайно зустрічаються такі помилки:

`2` – файл не знайдений;

`3` – маршрут не знайдений;

`5` – доступ заборонений;

`32` – порушення роздільного доступу.

Коди помилок введення/виведення VCL такі:

`100` – кінець файлу;

`101` – диск заповнений;

`102` – файлу не присвоєне ім’я;

`103` – файл не відкритий;

`104` – файл не відкритий для введення;

`105` – файл не відкритий для виведення;

`106` – помилка у форматі даних, що вводяться;

`107` – файл уже відкритий.

Код помилки може бути проаналізований користувачем для організації відповідної реакції.

Слід пам’ятати, що при `{$I-}` у випадку виникнення помилок введення/виведення всі наступні операції роботи з файлами ігноруються до звертання до функції `IOResult`. Звертання до функції `IOResult` без виконання операцій введення/виведення, у тому числі повторне для однієї і тієї ж операції, дає значення `0`. Відзначимо більш сучасним для контролю помилок є використання механізму обробки винятків (див. розд. 13).

Після фрагмента програми, у якому можливі помилки введення/виведення, необхідно відновити активний стан директиви: `{$I+}`.

```
//Приклад 6.3
//Фрагмент опрацьовувача події OnClick для Button1 з
//відкриттям текстового файлу для читання при введенні його
//імені з клавіатури.
```

Як форму пропонується використати форму з прикладу 2.2 з такими змінами властивостей деяких її компонентів:

- властивість Caption мітки – текст Уведіть ім'я файлу;
- властивість Text однорядкового редактора – очистити;
- властивість Caption кнопки Button1 – текст Виконати.

Напишемо для компонента Button1 такий опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f: TextFile;
  NameOfFile: string;
  code: Integer;
begin
  {$I-}                    //Відключення автоматичного контролю
  NameOfFile := edInput1.Text;
  AssignFile(f, NameOfFile);
  Reset(f);
  code := IOResult;
  if code <> 0 then begin
    mmOutput1.Lines.Add('Помилка відкриття файлу');
    mmOutput1.Lines.Add('Повторіть введення імені файлу');
    edInput1.SetFocus;
  end
  else begin
  {$I+}                    //Включення автоматичного контролю
    mmOutput1.Lines.Add('Файл ' + NameOfFile +
                        ' відкритий для введення');
        //Продовження програми
  end;
end;
```

У наведеному тексті показано, що для відключення автоматичного контролю правильності відкриття файлу у пасивний стан переведено директиву компілятору `$I` (`{$I-}`), а далі після відкриття файлу за допомогою функції `IOResult` зчитується код помилки введення/виведення з наступним його аналізом (відсутності помилки відповідає нульове значення коду). При успішному відкритті файлу директива контролю введення/виведення встановлюється в активний стан. Підкреслимо також,

що у даному випадку контролюється просто помилка введення/виведення без її конкретизації.

Описана методика контролю введення/виведення широко застосовувалася в мові Turbo Pascal і, звісно, може бути використана в програмах, написаних мовою Delphi. У той же час більш сучасним і розповсюдженим є механізм виняткових ситуацій (див. розд. 13).

6.4. Типізовані файли

Більш характерними є типізовані файли, або файли довільного доступу, фундаментальною властивістю яких є те, що ця структура даних являє собою послідовність компонентів одного й того самого типу. Описують подібний файл словосполученням **file of** з наступним зазначенням типу компонентів файлу, кількість яких (довжина файлу) не фіксується:

```
var
  файлова_змінна: file of тип_компонентів;
```

Оскільки відомий тип компонентів файлу, а отже, й обсяг пам'яті, що відводиться під кожний з них, то можна обчислити позицію кожного з компонентів усередині файлу, що дозволяє організувати безпосередній доступ до довільного компонента типізованого файлу. Наприклад, опис

```
var
  FileInt: file of Integer;
```

свідчить про те, що компонентами файлу є дані типу `Integer`, які займають 4 байт. При цьому відпадає необхідність у спеціальному поділі окремих компонентів файлу, як це має місце в текстових файлах, і можливий довільний доступ до них (у цьому розумінні типізований файл дещо нагадує одновимірний масив).

Деякі типи даних (наприклад, тип **string** або динамічні масиви) не можуть бути записані у файл. Це пояснюється тим, що під час виконання обсяг пам'яті, що відводиться під них, може змінюватися, і компілятор не може організувати дії по завершенню процесу запису таких даних у файл або читання їх з файлу. У такому випадку краще працювати з нетипізованими файлами, визначивши власні підпрограми для їх обробки.

Щоб можна було працювати з типізованим файлом, потрібно, як і у випадку текстових файлів, спочатку зв'язати ім'я файлової змінної з зовнішнім ім'ям файлу (оператор `Assign`), а потім відкрити його (оператори `Reset` та `Rewrite`, але не `Append`). Оператори `Reset` і `Rewrite` відкривають файл і для читання, і для запису (а не тільки для читання або тільки для запису, як це має місце в текстових файлах). Відмінність їх

полягає у тому, що оператор `Reset` відкриває тільки існуючий файл (якщо такого файлу немає, то буде помилка часу виконання), а оператор `Rewrite` створює новий файл (якщо файл з таким ім'ям уже є, то він буде знищений і створений заново). При відкритті файлу з ним зв'язується поточний вказівник файлу, що позиціюється на його перший компонент. Оперувати можна тільки тим компонентом файлу, на який указує вказівник файлу. При читанні або записі компонента файлу відбувається автоматичне переміщення вказівника на наступний компонент. Читання з типізованого файлу проводиться оператором `Read` (але не `ReadLn`), а запис у нього – оператором `Write` (але не `WriteLn`), що збігаються за форматом з аналогічними операторами для текстових файлів; однак у списку виведення оператора `Write` можуть бути тільки змінні. Типи компонентів файлу і типи змінних у списках введення/виведення повинні бути сумісними за присвоюванням. Компонентами типізованих файлів можуть бути числові та символьні значення, значення логічного типу, короткі рядки, масиви, множини, записи, але не файли або структури з файловими компонентами.

Дізнатись кількість компонентів типізованого файлу (розмір файлу) можна, звернувшись до функції

`FileSize(`файлова_змінна`).`

Наприклад, якщо змінна `k` має тип `Integer`, а `f` – файлова змінна типізованого файлу, то оператор `k:=FileSize(f)` записує в змінну `k` розмір файлу `f` (функція `FileSize` повертає значення типу `Integer`).

Компоненти типізованого файлу нумеруються, причому починаючи з `0` (порядковий номер останнього елемента файлу на одиницю менший від розміру файлу). Щоб довідатися, на якому компоненті розташовується вказівник файлу, використовують функцію `FilePos`, яка повертає значення типу `LongInt` і як параметр має файлову змінну:

`FilePos(`файлова_змінна`).`

Положенням поточного вказівника можна керувати, для чого служить процедура `Seek`, що має формат

`Seek(`файлова_змінна`,` номер_компонента`).`

Другий параметр (тип `LongInt`) задає номер компонента (відлік від 0), на який повинен переміститися вказівник файлу:

`Seek(f, 5)` – перейти до компонента файлу `f`, що має номер 5 (фактично шостий компонент);

`Seek(f,FilePos(f)-1)` – перейти до попереднього компонента;

`Seek(f, FileSize(f))` – перейти на кінець файлу.

Закриття типізованого файлу здійснюється раніше розглянутою процедурою `FileClose`.

Як і для текстових файлів, для типізованих можна використовувати функцію

`Eof(`ім'я_файлу`)`,

що повертає значення `True`, якщо поточний вказівник розташований на ознаці кінця файлу, тобто при виконанні рівності

`FilePos(`ім'я_файлу`) = FileSize(`ім'я_файлу`)`.

Процедура `Seek` і функції `FilePos` та `FileSize` дозволяють легко коректувати компоненти файлу й організовувати дозаписування у кінець файлу зі збільшенням його розміру.

Процедура

`Truncate(`файлова_змінна`)`

знищує всі компоненти типізованого файлу, ім'я якого зазначене як її параметр, починаючи з компонента, на якому розташований поточний вказівник. Однак знищити компонент усередині файлу не можна, для цього файл повинен бути переписаний.

У той час як текстові файли можуть бути створені текстовим редактором, типізовані файли створюються в результаті роботи якої-небудь програми, найпростіший приклад якої наводиться нижче.

```
//Приклад 6.4
//Записати у файл прізвища і телефони 10 осіб.
```

Як форму використаємо форму з прикладу 2.2, розмістивши на панелі додатково ще одну мітку з властивістю Name, що дорівнює lbOutput2, і властивістю Caption, що містить текст Номер телефону (у вигляді числа), а також ще один однорядковий редактор з властивістю Name, яка дорівнює edInput2. Внесемо також наступні зміни в значення властивостей компонентів:

- властивість Caption однорядкового редактора edInput1 – текст Прізвище абонента;
- властивість Text обох однорядкових редакторів – очистити;
- властивість Caption кнопки Button1 – текст Виконати.

У секції **`interface`** модуля опишемо такий тип:

```
type
  t_subscriber = record
    surname: string[20];
    tel: LongInt
  end;
```

Крім того, у розділі **var** секції **implementation** модуля опишемо файл такого типу:

```
f: file of t_subscriber;
```

Опрацьовувач події OnClick компонента Button1 може мати вигляд

```
procedure TForm1.Button1Click(Sender: TObject);
var
  subscriber: t_subscriber;                          //Абонент
begin
  if Tag = 0 then begin
    AssignFile(f, 'notebook.dat');
    Rewrite(f);                                     //Новий файл
  end;
  with subscriber do begin
    surname := edInput1.Text;
    tel := StrToInt(edInput2.Text);
    Write(f, subscriber);              //Запис компонента у файл
    mmOutput1.Lines.Add(surname + #9 + IntToStr(tel));
    edInput1.SetFocus;
  end;
  Tag := Tag + 1;
  if Tag = 10 then begin
    CloseFile(f);                                  //Закриття файлу
    Button1.Visible := False;
    lbOutput1.Visible := False;
    edInput1.Visible := False;
    lbOutput2.Visible := False;
    edInput2.Visible := False;
  end;
end;
```

В опрацьовувачі події OnClick для організації циклу введення використано наявну у форми властивість Tag, значення якої за умовчанням дорівнює 0 (див. зауваження до прикладу 6.2). Передбачається, що перед кожним кліком мишкою над кнопкою введення користувач повинен увести в перший і другий однорядковий редактори відповідно прізвище абонента і номер його телефону у вигляді цілого числа. Після першого спрацьовування кнопки введення (коли Tag=0) відкривається новий файл з ім'ям notebook.dat, а якщо Tag=10, то виконується закриття файлу. Дані, що вводяться, записуються спочатку в поля surname і tel змінної subscriber, а потім запис subscriber переписується у файл. У програмі також проілюстроване застосування оператора **with**. Оголошення поля surname коротким рядком пояснюється тим, що довгі рядки не можуть бути компонентами типізованого файлу або описувати ті поля

записів, які є компонентами типізованого файлу (відзначимо, що перевірка існування файлу notebook.dat у програмі не передбачена, що може призвести до знищення вже існуючого файлу з таким ім'ям).

Організоване запропонованим способом введення даних з клавіатури не зовсім зручне через необхідність постійного переходу від одного редактора введення до іншого, а також багаторазової подачі команд за допомогою кнопки введення. Більш розумною уявляється попередня підготовка всього набору даних з подальшим записом його у файл. З цією метою для введення зручно використовувати багаторядковий редактор Memo. Розглянемо це на прикладі тієї самої задачі, знявши обмеження на кількість даних, що вводяться.

```
//Приклад 6.5
//Записати у файл прізвища і телефони декількох абонентів.
```

Видалимо з описаної в прикладі 6.3 форми обидві мітки й обидва однорядкових редактори введення, присвоїмо властивості Name багаторядкового редактора ім'я mmInput1 і скористаємося наступним опрацьовувачем події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
type
  t_subscriber = record
    surname: string[20];
    tel: LongInt;
  end;
var
  subscriber: t_subscriber;                          //Абонент
  f: file of t_subscriber;
  s: string;
  i: Integer;
begin
  AssignFile(f, 'notebook.dat');
  Rewrite(f);
  for i := 0 to mmInput1.Lines.Count - 1 do begin
    s := Trim(mmInput1.Lines[i]);
    if s <> ' ' then
      with subscriber do begin
        surname := Copy(s, 1, Pos(' ', s) - 1);
        tel := StrToInt(Trim(Copy(s, Pos(' ', s),
                                  Length(s))));
        Write(f, subscriber);          //Запис компонента у файл
      end;
    end;
  CloseFile(f);                                    //Закриття файлу
  Button1.Visible := False;
end;
```

У даному випадку запропонована така методика введення даних: дані у вигляді рядків, у яких записані розділені одним або декількома пробілами прізвище абонента і його номер телефону у вигляді цілого числа, набираються у вікні багаторядкового редактора mmInput1, після чого виконується клік мишкою над кнопкою введення. Особливістю багаторядкового редактора є те, що одна з його властивостей – це властивість Lines типу TStrings, яку можна інтерпретувати як масив елементів типу **string** з індексуванням від 0 за допомогою значень типу Integer. Кількість елементів цього масиву (загалом кажучи, списку) визначається значенням властивості Count, що змінюється автоматично при зміні кількості рядків. У циклі за змінною i рядки по черзі переписуються в змінну s типу **string** з видаленням кінцевих пробілів за допомогою функції Trim, після чого виділяється прізвище абонента (як частина рядка s до пробілу) і номер телефону (як частина рядка s після пробілу). Прізвище і перетворений до цілочислового виду номер телефону записуються відповідно в поля surname і tel змінної subscriber записного типу t_subscriber, що є базовим типом файлової змінної f, яка на самому початку зв'язується зі знову створюваним файлом notebook.dat, після чого вміст змінної subscriber за допомогою оператора Write записується у файл f. З метою недопущення надходження на обробку помилково набраних порожніх рядків (у тому числі порожнього рядка, що може з'явитися в результаті натискання клавіші <Enter> після набору останнього введеного рядка) перед обробкою рядок s перевіряється на його порожність.

Природно, при необхідності типізованими файлами можна оперувати, розглядаючи їх як файли послідовного доступу. Це ілюструє наступна програма.

```
//Приклад 6.6
//На телефонній станції шестизначні номери, що починаються
//з 37, замінюються семизначними номерами, що починаються
//з 337. Внести зміни у файл notebook.dat у поточній папці.
```

Скористаємося формою з прикладу 6.5, присвоївши властивості Name багаторядкового редактора ім'я mmOutput1 і записавши у властивість Caption компонента Button1 текст Виконати. Текст опрацьовувача події OnClick компонента Button1 може бути таким:

```
procedure TForm1.Button1Click(Sender: TObject);
type
  t_subscriber = record
    surname: string[20];
    tel: LongInt;
  end;
```

```
var
  subscriber: t_subscriber;                              //Абонент
  f: file of t_subscriber;
  s: string;
begin
  AssignFile(f, 'notebook.dat');
  Reset(f);                                          //Існуючий файл
       //Нижче файл розглядається як файл послідовного доступу
  while not Eof(f) do begin
                                        //Читання компонента з файлу
    Read(f, subscriber);
    with subscriber do begin
      Str(tel, s);
      if (Length(s) = 6) and (Copy(s, 1, 2) = '37') then begin
        Insert('3', s, 1);
        tel := StrToInt(s);
      end;
    end;
                              //При читанні вказівник змістився
                  //Повертаємо його назад (вже довільний доступ)
    Seek(f, FilePos(f) - 1);
                                      //Змінюємо компонент файлу
    Write(f, subscriber)
  end;
  CloseFile(f);                                     //Закриття файлу
  Reset(f);
  mmOutput1.Lines.Add('Контрольне виведення');
  while not Eof(f) do begin
    Read(f, subscriber);
    with subscriber do
      mmOutput1.Lines.Add(surname + #9 + IntToStr(tel));
  end;
  CloseFile(f);
  Button1.Visible := False;
end;
```

У наведеній програмі типізований файл обробляється і як файл послідовного доступу, і як файл довільного доступу.

Як компоненти файлів можуть бути не тільки звичайні записи, але й варіантні.

6.5. Нетипізовані файли

У Delphi, крім розглянутих, існують також нетипізовані файли, які сумісні з усіма типами файлів і використовуються тоді, коли тип елементів файлу не важливий (наприклад, при копіюванні). Такі файли мають опис

```
var
  файлова_змінна: file;
```

Наприклад, можливий такий опис:

```
var
  FileOneType: file;
```

Файл без типу розглядається як послідовність компонентів довільного типу, але обумовленого розміру: у нього можна записати значення будь-якої змінної, що має заданий розмір, а при читанні з такого файлу допускається довільна інтерпретація вмісту чергового компонента.

Відкриваються файли без типу тими ж операторами `Reset` і `Rewrite`, але в цьому випадку є другий параметр – розмір запису (компонента файлу, блок), заданий у байтах. Попередньо потрібно за допомогою оператора `AssignFile` зв'язати внутрішнє ім'я файлу із зовнішнім:

```
AssignFile(FileOneType, 'f.dat');
Reset(FileOneType, 1);
```

Другий параметр операторів `Reset` і `Rewrite` може бути опущений, що означає задавання розміру запису в 128 байт. Найбільша швидкість обміну даними забезпечується при довжині запису, кратній 512 байт (розміру сектора на диску).

Варто 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 при задаванні розміру запису рівним 1 байт.

Обмін даними при роботі з нетипізованими файлами виконується за участі робочого буфера. Як такий буфер застосовується оголошена в програмі змінна. Її розмір повинен бути достатнім для розміщення даних, які читаються (записуються) за один сеанс читання (запису).

Перед читанням нетипізований файл має бути відкритий за допомогою `Reset`, а саме читання здійснюється процедурою `BlockRead`, до якої можна звертатися згідно з одним із двох форматів:

```
BlockRead(файлова_змінна, змінна_буфер, кількість_записів)
```

або

```
BlockRead(файлова_змінна, змінна_буфер, кількість_записів,
                                передана_кількість).
```

За допомогою процедури `BlockRead` за один сеанс читання можна прочитати декілька записів. Їх кількість задається третім параметром процедури, який має тип `Integer`.

При виконанні процедури BlockRead дані поміщаються в оперативну пам'ять, починаючи з першого байта змінної, зазначеної як другий параметр процедури BlockRead. Тому змінна_буфер повинна мати розмір, який дорівнює добутку кількості записів, що читаються за один раз (третій параметр), і розміру запису, заданого в процедурі Reset. У процедурі BlockRead можливе задавання четвертого параметра (тип Integer), у який записується кількість фактично прочитаних записів.

Запис даних у нетипізований файл проводиться тільки після його відкриття за допомогою Rewrite. Для запису даних використовується процедура BlockWrite, що має ті ж три (або чотири) параметри, що й BlockRead. При цьому в змінну_буфер потрібно попередньо помістити записи, кількість яких має збігатися зі значенням третього параметра 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 й четвертого параметрів будуть відрізнятися.

Якщо при читанні з диска виявиться, що розмір буфера буде меншим від зазначеного вище розміру або при записі на диск недостатньо вільного місця, то за відсутності четвертого параметра в процедурах BlockRead і BlockWrite буде зафіксована помилка. За наявності четвертого параметра помилка не фіксується.

```
//Приклад 6.7
//Скопіювати файл.
```

Для розв'язання задачі скористаємося формою з прикладу 2.2, змінивши текст, що міститься у властивості Caption кнопки запуску, на текст Копіювати і помістивши на панелі додаткове поле введення та ще одну мітку.

Установимо для додаткових компонентів такі властивості:

- Мітка:
  Caption — Файл-копія
  Name — lbOutput2
- Поле введення:
  Name — edInput2
  Text — очистити

У першої ж мітки замінимо текст, що міститься в її властивості Caption, на текст Файл-оригінал.

Як опрацьовувач події OnClick компонента Button1 використаємо таку процедуру:

```
procedure TForm1.Button1Click(Sender: TObject);
const
  CountRead = 512;           //Кількість записів, що читаються
              //(при довжині записів в 1 байт - розмір буфера)
var
  FileIn, FileOut: file;           //Файл-оригінал, файл-копія
  NumRead,                           //Число фактично зчитаних
  NumWritten: Integer;          //і фактично записаних записів
  Buf: array[1..CountRead] of Char;              //Змінна-буфер
  Name1,                                  //Імена файлу-оригіналу
  Name2: string;                                  //і файлу-копії
begin
  AssignFile(FileIn, edInput1.Text);
  {$I-}
  Reset(FileIn, 1);                       //Довжина запису 1 байт
  {$I+}
  if IOResult <> 0 then
    mmOutput1.Lines.Add('Файл-оригінал не знайдений')
  else begin
    AssignFile(FileOut, edInput2.Text);
    Rewrite(FileOut, 1);                  //Довжина запису 1 байт
    repeat
      BlockRead(FileIn, Buf, CountRead, NumRead);
      BlockWrite(FileOut, Buf, NumRead, NumWritten);
    until (NumRead = 0) or (NumWritten <> NumRead);
    mmOutput1.Lines.Add('Копіювання завершене');
    if NumRead <> NumWritten then
      mmOutput1.Lines.Add('Немає місця на диску для файлу '
                          + Name2);
    CloseFile(FileOut);
    CloseFile(FileIn);
    Button1.Visible := False;
  end;
end;
```

У даному випадку передбачена перевірка правильності відкриття існуючого файлу, для чого вжито функцію `IOResult` і директиву `{$I-}`. Аналогічні дії щодо файлу-копії не передбачені, тому якщо, наприклад, для файлу-копії буде введене ім'я вже існуючого, але захищеного від запису файлу, програма завершиться помилково. Крім того, якщо ім'я файлу-копії виявиться збіжним з ім'ям існуючого файлу, останній буде знищений.

У процесі читання/запису за допомогою `BlockRead` і `BlockWrite` поточний вказівник переміщається на ту кількість записів, яка була оброблена. Можливе також застосування функцій `FileSize` і `FilePos` та процедури `Truncate`, які в цьому випадку оперують записами оголошеного в `Reset` або `Rewrite` розміру.

6.6. Інші підпрограми для роботи з файлами

Крім згаданих підпрограм, для роботи з файлами будь-якого виду в Delphi існує ще кілька підпрограм. Розглянемо деякі з них.

Процедура `Rename` служить для перейменування файлу або каталогу. Її формат наступний:

`Rename(`файлова_змінна, нове_ім'я`).`

Другий параметр задається рядковим виразом і зазначає нове зовнішнє ім'я файлу або каталогу.

Для видалення файлу в Delphi використовується процедура `Erase`, єдиним параметром якої є внутрішнє ім'я файлу:

`Erase(`файлова_змінна`).`

Ці дві процедури працюють тільки з закритим файлом, але попередньо за допомогою оператора `AssignFile` файлова змінна (тип якої непринциповий) повинна бути пов'язана з зовнішнім ім'ям файлу (або каталогу, якщо перейменовується каталог).

Для видалення файлу можна застосовувати також функцію

`DeleteFile(`ім'я_файлу`),`

що працює не з файловою змінною, а безпосередньо з ім'ям файлу (параметр типу **`string`**) і не потребує попереднього виконання процедури `AssignFile`. Функція повертає значення `True` при її нормальному завершенні і значення `False`, якщо видаляється відсутній файл або видалення не відбулося.

Чотири процедури (`ChDir`, `MkDir`, `RmDir` і `GetDir`) у Delphi забезпечують роботу з каталогами (папками).

Перші три процедури мають один і той самий формат:

`ChDir(`каталог`), MkDir(`каталог`), RmDir(`каталог`).`

У всіх трьох випадках параметр задається рядковим виразом і містить ім'я каталогу в інтерпретації операційної системи.

Процедура `ChDir` змінює поточний каталог на зазначений як параметр, процедура `MkDir` створює новий каталог із ім'ям, зазначеним як її параметр, а процедура `RmDir` знищує каталог із зазначеним ім'ям за умови, що він порожній.

Процедура `GetDir` дозволяє визначити ім'я поточного каталогу на певному диску. Формат процедури:

`GetDir(`диск, каталог`),`

де диск – вираз типу `Word`, що задає номер диска (`0` – активний диск, `1` – диск A, `2` – диск B і т. д.); каталог – змінна типу **`string`**, що служить для повернення в точку виклику шляху до поточного каталогу на диску, номер якого подано як перший параметр процедури.

```
//Приклад 6.8
//Застосування різних процедур для роботи з файлами.
//Передбачається, що в поточному каталозі знаходиться
//файл або підкаталог з ім'ям fd.
```

Оскільки в постановці задачі не передбачається введення і виведення даних (задача є чисто ілюстративною), обмежимося формою з кнопкою Button1, властивість Caption якої містить текст Виконати, і кнопкою із зображенням BitBtn, присвоївши їй ім'я bbClose і задавши властивість Kind рівною bkClose. Напишемо для компонента Button1 наступний опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   f: file;          //Можна використати будь-який файловий тип
   s: string;
begin
  AssignFile(f, 'fd');
  Rename(f, 'ttt');         //Перейменування файлу або каталогу
  MkDir('TMP');//У поточному каталозі створений підкаталог TMP
  ChDir('TMP');                       //Перехід у підкаталог TMP
  GetDir(0, s);   //Запис в s повного імені поточного каталогу
           //Якщо вихідний каталог C:\USER, то s='C:\USER\TMP'
  AssignFile(f, 'f.tmp');
  Rewrite(f);               //Створення допоміжного файлу f.tmp
          //Продовження програми для обробки...
          //...файлу f.tmp - потрібна постановка задачі
  CloseFile(f);             //Закриття файлу для його видалення
  Erase(f);                              //Видалення файлу f.tmp
  ChDir('..');                            //Перехід у надкаталог
  Button1.Enabled := False;
end;
```

Особливістю даної процедури є те, що в ній не передбачається контроль можливих помилок, спричинених некоректною роботою з файлами. Так, якщо в поточному каталозі відсутній файл (або підкаталог) з ім'ям `fd`, то підпрограма `Rename` завершиться з помилкою. Уникнути помилки можна, звернувшись до функції `FileExists` перед тим, як виконати оператори `AssignFile` і `Rename` (однак при цьому не буде перевірятись існування каталогу з ім'ям `fd`). З наведеного тексту видно, що перед звертанням до підпрограми `Rename` треба виконати підпрограму

AssignFile. У той же час до звертання до процедури Erase закрито файл за допомогою процедури CloseFile; це пояснюється тим, що файл, який видаляється, повинен бути закритий перед видаленням.

Функція

FindFirst(шлях, атрибути, перший_запис)

виконує пошук першого запису, що відповідає заданим імені (параметр шлях типу **string**) і набору атрибутів (атрибути, тип Byte).

Запис може відшукуватися не тільки за конкретним ім'ям (з можливим розширенням), але і серед файлів та каталогів, що задаються за маскою в першому параметрі. При цьому маска задається звичайним для Windows чином, а саме, у масці можна подавати символ *, який служить для зазначення того, що замість нього може стояти будь-яка допустима для імені файлу або розширення послідовність символів (у тому числі і нульовій довжині), а також символ ?, що служить для позначення будь-якого допустимого символу (одного). Ім'я, що задається, може бути повним (містити ім'я диска і маршрут доступу) або скороченим. В останньому випадку діє звичайний для Windows принцип умовчання.

Набір атрибутів (ознак) – це деяке числове значення. Окремим атрибутам відповідають наступні константи модуля SysUtils (у коментарях зазначено зміст констант):

```
faReadOnly  = $01,                  //Тільки для читання
faHidden    = $02,                     //Прихований файл
faSysFile   = $04,                      //Системний файл
faVolumeID  = $08,                          //Мітка тому
faDirectory = $10, //Ім'я директорія (каталогу, папки)
faArchive   = $20,                       //Архівний файл
faAnyFile   = $3F.                         //Усякий файл
```

Комбінації атрибутів задаються іншими числовими значеннями, які звичайно отримують за допомогою застосування бітової операції **and** (або інших бітових операцій) до операндів, що задаються наведеними вище значеннями атрибутів. Результати роботи функції записуються в третьому параметрі, тип якого описано у модулі SysUtils так:

```
type
  TSearchRec = record
    Time: Integer;
    Size: Integer;
    Attr: Integer;
    Name: TFileName;
    ExcludeAttr: Integer;
    FindHandle: THandle;
```

```
    FindData: TWin32FindData;
end;
```

У цьому параметрі відображаються характеристики першого знайденого файлу, що зареєстрований у зазначеному каталозі: упакований час створення або останнього оновлення (поле `Time`), розмір файлу в байтах (поле `Size`), значення атрибута (поле `Attr`), ім'я й розширення файлу або каталогу (поле `Name`), додаткова інформація про час створення та останнього доступу, а також про повне і скорочене ім'я (поле `FindData`). Значення поля `Time` за допомогою функції

`FileDateToDateTime(`час_створення_файлу`)`

можна перетворити до типу `TDateTime`, у якому повертає результат своєї роботи дана функція.

Зазначимо, що зворотне перетворення виконує функція

`DateTimeToFileDate(`значення_типу_`DateTime)`,

яка повертає результат типу `Integer`, у якому задається системний час створення файлу.

Функція

`FindNext(`наступний_запис`)`

служить для пошуку на диску наступного запису з тими ж параметрами, що й у раніше виконаній функції `FindFirst` або `FindNext`. Її параметр має тип `SearchRec`.

Оскільки для роботи підпрограм `FindFirst` і `FindNext` виділяється додаткова пам'ять, її рекомендується звільняти по закінченні пошуку. Це забезпечується використанням функції

`FindClose(`запис_про_файл`)`,

параметром якої є змінна, що вживалося в раніше виконаній функції `FindNext` або `FindFirst`.

При нормальному завершенні функції `FindFirst` і `FindNext` повертають значення `0`, інакше – код помилки.

Функція

`FileAge(`ім'я_файлу`)`

значенням типу `Integer` повертає час створення (останньої корекції) файлу, ім'я якого (але не ім'я файлової змінної) зазначене як її параметр. За відсутності зазначеного файлу функція повертає значення `-1`. Повернуте функцією значення за допомогою функції `FileDateToDateTime` може бути перетворене до типу `TDateTime`.

Атрибути файлу можна визначити, скориставшись функцією

`FileGetAttr(`ім'я_файлу`)`,

що повертає значення `-1` при помилковому завершенні.

Установка атрибутів проводиться за допомогою функції

`FileSetAttr(`ім'я_файлу, комбінація_атрибутів`)`,

яка повертає значення `0` при нормальному своєму завершенні і код помилки у випадку завершення з помилкою.

Знаючи дескриптор відкритого файлу, можна визначити або змінити дату і час його створення, для чого використовуються відповідно функції

`FileGetDate(`дескриптор`)`

і

`FileSetDate(`дескриптор, час`)`.

При цьому функція `FileGetDate` повертає час створення файлу в упакованому системному форматі як ціле число (аналогічно значенню, записуваному в поле `Time` запису типу `SearchRec`). У цьому ж форматі задається час при звертанні до функції `FileSetDate`.

При нормальному завершенні функція `FileSetDate` повертає значення `0`, інакше – код помилки.

```
//Приклад 6.9
//Якщо в поточному каталозі є файл test.dat, установити час
//його створення на 2 хвилини після 0 годин 01.01.2020.
//Крім того, у поточному каталозі для всіх файлів
//з розширенням .txt, що не мають атрибут "тільки для
//читання", додатково встановити атрибут прихованості.
```

Скористаємося формою з прикладу 6.1 і наступним опрацьовувачем події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Age: TDateTime;
  at: Integer;
  file_rec: TSearchRec;
  f: file;
  Handle, Code: Integer;
begin
      //Упаковуємо і запам'ятовуємо новий час створення файлу
  Age := StrToDateTime('01.01.2020 00:02');
       //Відкриваємо файл і визначаємо його дескриптор Handle
  Handle := FileOpen('test.dat', fmOpenReadWrite);
```

```
  if Handle <> -1 then begin            //Якщо файл відкрився,
                    //то встановлюємо новий час створення файлу
    FileSetDate(Handle, DateTimeToFileDate(Age));
    FileClose(Handle);                      //і закриваємо файл
  end;
      //Пошук першого файлу (каталогу, мітки тому)
             //з розширенням txt
  Code := FindFirst('*.txt', faAnyFile, file_rec);
  while Code = 0 do begin                //Поки пошук успішний
    at := FileGetAttr(file_rec.Name);      //Зчитуємо атрибут
                       //Якщо це не каталог і не мітка тому та
    if at and (faDirectory or faVolumeID) = 0 then
                              //якщо немає захисту від запису,
      if at and faReadOnly = 0 then     //то змінюємо атрибути
        FileSetAttr(file_rec.Name, at or faHidden);
    Code := FindNext(file_rec);       //Шукаємо наступний файл
  end;
  FindClose(file_rec);                     //Звільняємо пам'ять
end;
```

У наведеному тексті показано методику відкриття файлу на низькому рівні (за допомогою функції `FileOpen`) й одержання його дескриптора (змінна `Handle`). Задані у вигляді рядка дата та час за допомогою функції `StrToDateTime` перетворюються до типу `TDateTime`, після чого отримане значення, додатково перетворене за допомогою функції `DateTimeToFileDate` до системного формату часу створення файлу, використовується у функції `FileSetDate` для установки нового часу створення файлу `test.dat`.

У другій половині тексту процедури спочатку за допомогою функції `FindFirst` шукається перший запис про файл або каталог з розширенням `.txt` (константа `faAnyFile` свідчить про необхідність розгляду всіх файлів). При виявленні такого запису (`Code = 0`) виконується цикл продовження пошуку з переходом до нового такого файлу по функції `FindNext`, що при виявленні нового запису про файл, який задовольняє масці, також повертає значення `0`. Застосування функції `FileGetAttr` забезпечує зчитування комбінації атрибутів знайденого файлу або папки (у даному випадку це не потрібно, оскільки комбінація атрибутів уже зберігається в полі `Attr` запису про знайдений файл, – демонструється метод звертання до функції). Два оператори **if** забезпечують виключення з розгляду записів про каталоги та мітку тому, а також про файли, що мають атрибут захисту від запису. Далі за допомогою функції `FileSetAttr` у комбінацію атрибутів знайденого файлу додається атрибут прихованості. Цикл повторюється до вичерпання всіх файлів і папок.

Запитання для контролю і самоконтролю

1. Для чого використовуються файли даних?
2. Опишіть загальні принципи роботи з файлами.
3. Які файли використовуються в Delphi?
4. У чому особливість текстових файлів?
5. Як оголошується текстовий файл у програмі, написаній мовою Delphi?
6. Назвіть процедури і функції для роботи з текстовими файлами.
7. У чому особливість типізованих файлів? Як вони оголошуються?
8. Які засоби для роботи з типізованими файлами існують у Delphi?
9. Для чого служить поточний вказівник файлу?
10. Чи можна управляти положенням поточного вказівника файлу?
11. Як довідатися про кількість компонентів типізованого файлу?
12. Що таке нетипізований файл?
13. Які засоби для роботи з нетипізованими файлами існують у Delphi?
14. Як можна виконати перейменування і видалення файлів?
15. Як у Delphi проводиться створення і зміна каталогу?
16. Для чого служить набір атрибутів файлу?
17. Назвіть константи, що задають окремі атрибути.
18. Які підпрограми можуть використовуватися для пошуку файлів?
19. Як можна дізнатися набір атрибутів файлу?
20. Яким способом можна установити файлу набір атрибутів?
21. Як зчитати набір атрибутів файлу?
22. Як визначити час створення файлу?
23. Чи можна змінити час створення (корекції) файлу без зміни його вмісту?

Завдання для практичного відпрацьовування матеріалу

1. Дано текстовий файл *f*. Переписати в інший файл усі рядки вихідного файлу, що не містять латинських літер.
2. Дано текстові файли f_1, f_2, f_3, f_4, f_5. Використовуючи не більше одного додаткового файлу, організувати обмін вмістом між файлами відповідно до наступної схеми:

$$\begin{array}{ccccc} f_1 & f_2 & f_3 & f_4 & f_5 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ f_3 & f_4 & f_5 & f_2 & f_1 \end{array}$$

3. Дано текстовий файл. Переписати в інший файл частини рядків, починаючи з останнього слова, що не містить цифр.
4. Дано файл *f*, компоненти якого є дійсними числами. Знайти суму його додатних компонентів.
5. Дано файл цілих чисел. Записати в інший файл найбільше значення перших 10 компонентів, потім – наступних десяти компонентів і т. д. Якщо в останній групі менше 10 чисел, то розглядати неповну групу чисел.
6. Дано символьний файл *f*. Одержати файл *g*, утворений з файлу *f* заміною всіх його великих латинських літер відповідними малими.
7. Дано файл цілих чисел *f*. Записати у файл *g* парні компоненти файлу *f* у зворотному порядку.
8. Дано файл *f*, компоненти якого є цілими числами. Одержати файл *g*, утворений з файлу *f* виключенням повторних входжень одного і того самого числа.
9. Дано файл *f*, компоненти якого є цілими числами. Записати у файл *g* усі прості числа, що входять у файл *f*. Числа у файлі *g* повинні іти:
   а) за неубуванням;
   б) у порядку зростання без повторень.
10. Дано впорядковані за незростанням файли цілих чисел *f* та *g*. Записати у файл *h* всі числа файлів *f* та *g* без повторення. Значення, записані у файл *h*, повинні іти за не убуванням.
11. Дано символьний файл, що містить тільки малі латинські літери і пробіли. Переписати в інший символьний файл всі слова вихідного файлу, упорядкувавши їх за зростанням (з використанням звичайного правила порівняння рядків).
12. Дано файл *f* дійсних чисел, кількість яких кратна десяти. Переписати компоненти файлу *f* у файл *g*, змінюючи порядок проходження чисел у кожній десятці таким чином, щоб спочатку йшли від'ємні числа десятки, а потім усі інші. Порядок проходження самих десяток має бути змінений на зворотний
13. Дано файл *f* цілих чисел. Розглядаючи його компоненти у вигляді послідовності груп з 10 чисел (остання група не обов'язково повинна бути повною), переписати їх у файл *g*, змінюючи порядок проходження чисел у кожній групі таким чином, щоб спочатку йшли числа, що діляться на 3, потім числа, що діляться на 3 з остачею 1, а потім усі інші. Порядок проходження самих десяток має залишитися незмінним.
14. Скористатися методом «бульбашки» для упорядкування за неубуванням вмісту файлу *f*, що містить дійсні числа.
15. Багаж пасажира характеризується кількістю речей і загальною їхньою вагою. Дано файл *f*, що містить інформацію про багаж декількох

пасажирів. Інформація про багаж кожного окремого пасажира являє собою відповідну пару чисел.

а) Знайти багаж, середня вага однієї речі в якому відрізняється не більш ніж на 0,3 кг від загальної середньої ваги речей.

б) Знайти число пасажирів, що мають більше двох речей, і число пасажирів, кількість речей яких перевищує середнє число речей.

в) Визначити, чи є два пасажири, багажі яких збігаються за числом речей і розрізняються за вагою не більш як на 0,5 кг.

16. Дано файл, компоненти якого записані у форматі, що відповідає типу `Integer`. Числа у файлі йдуть у такому порядку: десять парних, десять непарних, десять парних, десять непарних і т. д. Переписати в інший файл вміст вихідного файлу так, щоб дані в ньому розміщалися в такому порядку:

а) п’ять парних, п’ять непарних, п’ять парних, п’ять непарних і т. д.;

б) двадцять непарних, двадцять парних, двадцять непарних, двадцять парних і т. д.

7. ВКАЗІВНИКИ І ДИНАМІЧНІ ДАНІ

7.1. Посилальні типи і вказівники. Тип Pointer

Пам’ять комп’ютера являє собою послідовність комірок, призначених для зберігання інформації. Як це відзначалося в підрозд. 3.5, мінімальною областю пам’яті, що може адресуватися, є байт, а кожен з байтів має свій номер. Для роботи з адресами в Delphi введено спеціальний посилальний тип, що описується символом ^ з наступним ім’ям базового типу, на значення якого здійснюється посилання:

```
type
  ім’я_посилального_типу = ^ім’я_базового_типу;
```

Наприклад, для визначення типу `TptI`, що посилається на тип `Integer`, у програмі повинне зустрітися таке оголошення типу:

```
type
  TptI = ^Integer;
```

Опис змінних посилального типу проводиться стандартно:

```
type
  TptI = ^Integer;
  TptRec = record
     Day: Byte;
     Month: string;
     Year: Word;
  end;

var
  p1, p2: TptI;
  DatePtr: ^TptRec;
```

Посилальний тип може бути заданий навіть у випадку, коли базовий тип ще не введений (єдиний виняток у правилі опису даних, яке вимагає обов’язкового опису імені **перед** першим його використанням). Однак тоді базовий тип має бути описаний у тій же послідовності оголошення типів:

```
type
  PtrType = ^BaseType;
  BaseType = record
               x, y: Real;
  end;
```

Значенням посилального типу є ***вказівник***, тобто адреса першого байта змінної того типу, що є базовим. Термін ***вказівник*** звичайно застосовують й у випадку, коли мова йде про змінну посилального типу. Якщо елемент даних займає в пам'яті більше одного байта, то вказівник містить адресу тільки першого з них.

Присвоєння змінній посилального типу значення виконується з використанням ***операції узяття вказівника*** (адреси) `@`. Для забезпечення доступу до вмісту адреси, на яку посилається змінна посилального типу, призначена ***операція розіймення*** вказівника, що позначається символом `^` **після** імені такої змінної. Так, у наведеному нижче фрагменті програми посилання вигляду `p^` еквівалентне вживанню змінної `k`:

```
type
  TptI = ^Integer;
var
  p: TptI;
  k: Integer;
  // ...
begin
  p := @j;            //У p - адреса j; p^ - те ж, що j
  j := Random(100);        //Те ж, що p^:=Random(100);
  p^ := p^ + 1;                        //Аналог j:=j+1
  // ...
```

Над змінними посилального типу можна виконувати тільки дві операції порівняння (= та <>) й операцію узяття адреси `@`. У Delphi є константа посилального типу **`nil`**, яку задає вказівник, що є порожнім. Вона сумісна з усіма посилальними типами.

Незважаючи на те що всі посилальні типи фізично однакові (адреси пам'яті), для різних базових типів вони вважаються різними. Так, помилковим є оператор присвоювання в такому фрагменті програми:

```
type
  TptI = ^Integer;
  TptR = ^Real;
var
  p1: TptI;
  p2: TptR;
```

```
begin
  p1 := p2;
  // ...
```

При роботі з вказівниками можна користуватися явним перетворенням типу. Так, в останньому прикладі оператор `p1:=p2` повинен бути записаний у вигляді `Tpt(p1):=p2`. Однак при цьому жодна перевірка на коректність звертань до пам'яті не проводиться.

Нехай, наприклад, у секції **`interface`** модуля присутні такі описи:

```
type
  TptB = ^Byte;
  TptR = ^Real;
var
  b: Byte;
  p1: TptR;
```

У цьому випадку компілятор не знайде помилку в наступному фрагменті програми:

```
p1 := @b;
TptR(p1)^ := -0.5;
// ...
```

Тут вказівник `Tpt(p1)^` посилається на ділянку з 8 байт, початок якої збігається з адресою змінної `b`, що займає 1 байт. У ці 8 байт записане значення `−0.5`, у результаті чого псується вміст двох описаних підряд змінних `b` і `p1`.

Крім ***типізованих вказівників***, про які йшлося раніше у цьому підрозділі, в Delphi є спеціальний тип `Pointer`, значення якого називають ***нетипізованим вказівником***. Він сумісний з усіма посилальними типами:

```
type
  TptI = ^Integer;
  TptR = ^Real;
var
  p1: TptI;
  p2: TptR;
  p3: Pointer;
begin
  p3 := p1;                          //Є припустимим
  p3 := p2;                          //Є припустимим
  p1 := p3;                          //Є припустимим
  p2 := p3;                          //Є припустимим
  p1 := p2;                        //Є неприпустимим
  // ...
```

Відзначимо також, що результат операції `@` має тип `Pointer`.

Єдиним можливим значенням типізованої константи-вказівника є **`nil`**, наприклад:

```
const
  PReal: ^Real = nil;
```

У наведеному нижче прикладі показана можливість використання явного перетворення типу, а також операції взяття адреси для узгодження вказівників різних типів.

```
//Приклад 7.1
//Узгодження вказівників різних типів.
```

Скористаємося формою з прикладу 5.13 з наступним опрацьовувачем події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
type
  TptB = ^Byte;
  TptS = ^ShortString;
var
  p1: TptB;
  p2: TptS;
  s1, s2: ShortString;
begin
  p2 := @s1;                              //p2 посилається на s
  p2^ := 'pascal';                                  //Запис в s1
  p1 := TptB(p2);                      //p1 посилається на s1[0]
  p1^ := 3;          //Змінюємо вміст s1[0]; тепер Length(s)=3
  mmOutput1.Lines.Add(p2^);
  s2 := 'Pascal';
  p1 := @s2;       //Заносимо в p1 адресу першого байта s2,...
  p1^ := 3;   //... тобто посилання на s2[0], і змінюємо s2[0]
  mmOutput1.Lines.Add(s2);
  Button1.Enabled := False;
end;
```

У програмі в змінну `p1` (посилання на тип `Byte`) заноситься в першому випадку посилання на рядок, на який вказує змінна `p2`, а в другому – на рядкову змінну `s2`. Далі за рахунок зміни вмісту комірки, на яку посилається `p1`, в обох випадках змінюється довжина рядка (`p1` посилається на початок рядка, де записана його довжина). У результаті оператор

```
mmOutput1.Lines.Add(p2^)
```

виводить рядок `pas`, а оператор

```
mmOutput1.Lines.Add(s2)
```

– рядок `Pas`.

Нагадаємо, що посилальними типами є типи **string**, PChar і **array of** ТИП.

7.2. Адресна арифметика і вказівники

Як це відзначалося в попередньому підрозділі, над вказівниками можна виконувати тільки дві операції порівняння (= та <>) й операцію узяття адреси @. Операції порівняння вказівників означають порівняння адрес областей пам'яті, на які ці вказівники посилаються (навіть при порівнянні потрібен збіг базових типів вказівників, за винятком порівняння з вказівником типу Pointer).

Змінні посилального типу можуть використовуватися як параметри процедур Inc та Dec в обох форматах. При цьому змінювання значення такої змінної на 1 відповідає зсуву вказівника в пам'яті на число байт, що відповідає розміру базового типу. Наприклад, якщо в секції **interface** модуля є описи вигляду

```
var
  m, n, k: LongInt;
  Pt: ^LongInt;
```

то в результаті виконання послідовності операторів

```
Pt := @k;Dec(Pt, 2);
```

вказівник Pt буде посилатися на змінну m (Pt=@m), бо оператор Dec(Pt, 2) «зсуне» вказівник Pt на 2*SizeOf(LongInt) байт убік зменшення адрес.

```
//Приклад 7.2
//Дано масив з n цілих чисел (можливо повторюваних),
//де n>0. Видалити в ньому всі максимальні елементи.
```

Скористаємося формою з прикладу 5.3, записавши у властивість Caption компонента Button3 текст Видалити. Включимо також у секцію **public** опису форми такі описи:

```
n: Integer;                    //Кількість елементів масиву
a: array of Integer;
```

Тоді задача розв'язується наступними опрацьовувачами події OnClick компонентів Button1, Button2 та Button3:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  n := StrToInt(edInput1.Text);          //Кількість елементів
  mmOutPut1.Lines.Add('n = ' + IntToStr(n));
```

```
  SetLength(a, n);              //Створюємо масив з n елементів
  lbOutput1.Caption := 'Уведіть елемент a[0]';
  edInput1.SetFocus;
  Button1.Visible := False;      //Button1 тепер невидимий,...
  Button2.Visible := True;           //...a  Button2 – видимий
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
begin
  if Tag < n then                    //Умова припинення введення
    a[Tag] := StrToInt(edInput1.Text);            //Зчитування
  Tag := Tag + 1;      //Змінюємо номер елемента, що вводиться
  if Tag = N then begin              //Якщо введення закінчене,
    mmOutPut1.Lines.Add('Вихідний масив');
    for i := 0 to n - 1 do   //...виводимо початкові дані і...
      mmOutput1.Lines.Add('a[' + IntToStr(i) + '] = '
                          + IntToStr(a[i]));
    edInput1.Visible := False;      //...готуємося до обробки,
    lbOutput1.Visible := False;        //перемикаючи видимість
    Button2.Visible := False;
    Button3.Visible := True;
  end
  else begin                                //Продовжуємо введення
    lbOutput1.Caption:='Уведіть елемент a[' +
                       IntToStr(Tag) + ']';
    edInPut1.SetFocus;
  end;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  i: Integer;
  max: Integer;
  Pt: ^Integer;
begin
  max := a[0];
  for i := 1 to n - 1 do
    if a[i] > max then max := a[i];
  Pt:=@a[0]; //У вказівник записуємо адресу нульового елемента
  for i := 0 to n - 1 do
    if a[i] <> max then begin
              //Запис в ділянку пам'яті за допомогою вказівника
      Pt^ := a[i];
      Inc(Pt);   //Зсув на 4 байти, тобто на наступний елемент
    end
    else Dec(n);
```

```
  mmOutPut1.Lines.Add('Результуючий масив');
  for i := 0 to n - 1 do
    mmOutput1.Lines.Add('a[' + IntToStr(i) + '] = '
                        + FloatToStr(a[i]));
  Button3.Visible := False;
end;
```

Процедура `TForm1.Button1Click` здійснює введення кількості елементів масиву `a` і його створення, а процедура `TForm1.Button3Click` забезпечує введення вмісту масиву.

У процедурі `TForm1.Button2Click` вказівник `Pt` спочатку посилається на `a[0]`. Надалі за допомогою процедури `Inc` забезпечується послідовне збільшення адреси, на яку посилається вказівник, на `SizeOf(Integer)` байт, що забезпечує послідовне переміщення його по елементах масиву. При цьому в масив `a` за вказівником `Pt` записуються елементи, що не збігаються з максимальним значенням. Зсув вказівника проводиться тільки у випадку здійснення перезапису; якщо ж перезапис не відбувається, то за допомогою процедури `Dec` зменшується кількість елементів у результуючому масиві.

Треба зазначити, що в процедурі `TForm1.Button1Click` змінюється значення змінної `n`, що входить у вираз для кінцевого значення параметра циклу. Це допустимо, оскільки кінцеве значення параметра циклу **`for`** обчислюється і запам'ятовується до початку виконання циклу.

Від інших вказівників дещо відрізняється тип `PChar`.

До змінних типу `PChar` застосовні не тільки процедури `Inc` та `Dec`, але над ними також здійсненні такі операції адресної арифметики:

- віднімання двох вказівників (−);
- віднімання з вказівника цілого числа (+);
- додавання вказівника та цілого числа (+).

Різниця двох елементів типу `PChar` визначається як ціле число, що дорівнює кількості байт, на яку зміщений один вказівник щодо іншого. Тому якщо дві змінні типу `PChar` посилаються на два сусідніх об'єкти, різниця між ними дорівнює ±1:

```
var
  PCh1, PCh2, Ch3: PChar;
  Ch1, Ch2: Char;
  k: Integer;
        ...
  PCh1 := @Ch1; PCh2 := @Ch3;
  k := PCh1 - PCh2;           //k=-2 або 2 залежно від
                    //місця розташування операторів опису
```

З вказівника типу PChar можна віднімати цілі значення. При цьому віднімання з вказівника типу PChar деякого цілого додатного значення *k* відповідає зсуву вказівника в пам'яті на *k* байт убік молодших адрес (числове значення вказівника зменшується).

Додавання вказівників заборонене, а додавання вказівника типу PChar і цілого значення виконується аналогічно відніманню: якщо до елемента даних типу PChar додати ціле значення n, то в результаті буде отриманий вказівник типу PChar, зміщений у пам'яті на n байт.

Характерно те, що сума або різниця вказівника типу PChar і цілочислового значення також є вказівником типу PChar, тобто припустимою буде, наприклад, конструкція вигляду (PCh1 + k)^.

До символьних масивів **array of** Char з нульовою базою також можна застосовувати адресну арифметику, однак ім'я такого масиву не може використовуватися як параметр процедур Inc та Dec (як і у лівій частині оператора присвоювання), оскільки воно є вказівником-константою.

```
//Приклад 7.3
//Випадковим способом сформувати завершуваний символом #0
//рядок, що не містить символів керування. Довжина
//рядка не повинна перевищувати половини відведеної під
//нього області пам'яті. Якщо в цьому рядку кількість
//символів з непарними кодами менша від кількості символів
//з парними кодами, доповнити його праворуч "дзеркальним"
//відбиттям; інакше залишити рядок без зміни.
```

Скористаємося формою з прикладу 5.13, включивши в секцію **public** опису форми оголошення таких полів:

```
z: array[0..100] of Char;
n: Integer;
```

Задачу розв'язують наступні опрацьовувачі подій:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  k, m1, m2: Integer;
  PCh1, PCh2: PChar;
begin
  PCh1 := @z[n];//Запам'ятовуємо позицію завершального символу
  m1 := 0;
  m2 := 0;
  k := 0;
        //До символьних масивів застосовна адресна арифметика.
           //Ім'я z – це вказівник на початок масиву, z+k – це
         //вказівник на елемент, що зміщений на k позицій від
         //початку масиву, (z + k)^ – значення цього елемента
```

```
  while (z + k)^ <> #0 do
  begin
    if Odd(Ord((z + k)^)) then
      Inc(m1)
    else
      Inc(m2);
    Inc(k);
  end;
  if m1 < m2 then begin
    PCh2 := PCh1 - 1;       //Застосування адресної арифметики
                    //Вказівник PCh2 вказує на попередній байт
                        //щодо до символу, на який вказує PCh1
                //(тут PCh1 вказує на останній символ рядка z)
    while PCh2 >= z do begin    //Вказівники можна порівнювати
      PCh1^ := PCh2^;
      Inc(PCh1);       //До всіх типізованих вказівників можна
      Dec(PCh2)            //застосовувати процедури Inc і Dec
    end;
    PCh1^ := #0;             //Дописуємо завершальний символ #0
  end;
  n := PCh1 - z;  //Різниця – кількість байт між PCh1 та @z[0]
  mmOutput1.Lines.Add('Довжина рядка дорівнює ' +
                       IntToStr(n));
  mmOutput1.Lines.Add(z);
  Button1.Enabled := False;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  k: Integer;
begin
  Randomize;
  n := Random(SizeOf(z) div 2) + 1;        //n – довжина рядка
             //Наприкінці рядка розміщаємо завершальний символ
  z[n] := #0;
  for k := n - 1 downto 0 do         //Тут працюємо з масивом z
    z[k] := Chr(Random(223) + 32);   //як зі звичайним масивом
  mmOutput1.Lines.Add(z);         //Символьні масиви з нульовою
        //базою можна виводити на екран і вводити з клавіатури
end;
```

У програмі показана можливість застосування адресної арифметики при роботі з символьними масивами та даними типу `PChar`: вираз `z+n` служить для організації переходу по елементах символьного масиву, вираз `PCh1-1` служить для одержання вказівника на попередній байт щодо `PCh1`, а конструкція `PCh1-z` служить для визначення кількості елементів у символьному масиві.

7.3. Динамічний розподіл пам'яті

7.3.1. Поняття динамічних змінних

Дотепер ми в основному розглядали випадок, коли пам'ять під змінні виділяється статично. Подібні змінні характеризуються тим, що вони мають один і той самий формат і розмір протягом усього часу їх існування. Під час компіляції для кожного статичного елемента даних виділяється постійний обсяг пам'яті, який не може бути змінений. Альтернативою статичному розподілу пам'яті є динамічний розподіл, коли пам'ять під змінні виділяється в міру необхідності. Ці питання частково розглядалися в п. 5.1.2, присвяченому динамічним масивам. Динамічно пам'ять виділяється і при роботі з довгими та широкими рядками. Особливістю динамічного розподілу пам'яті є ускладнення звертання до змінних, що спричиняє зменшення швидкості обчислень. Однак при цьому в розпорядженні програміста виявляється значно більший обсяг пам'яті для розміщення в ній даних порівняно з тим обсягом пам'яті, яким він може оперувати, якщо динамічна пам'ять ним не використовується явно (загалом кажучи, динамічна пам'ять у Delphi використовується і неявно). Для розміщення динамічних змінних у Delphi виділяється спеціальна ділянка пам'яті, що розподіляється динамічно, або інакше «купа» (heap). Робота з нею здійснюється за допомогою вказівників.

Ідея, що лежить в основі концепції вказівників, полягає у зв'язуванні певного типу даних з конкретним вказівником. Сам вказівник, у свою чергу, теж є елементом даних, являючи собою посилання на певну комірку пам'яті. Оскільки вказівник може посилатися на будь-яку адресу пам'яті, можна здійснити посилання і на адреси в динамічній області пам'яті. У цьому випадку забезпечується оперування вмістом адрес так, ніби вони є звичайними змінними, причому ці змінні в програмі не оголошуються, а створюються і знищуються по ходу виконання програми. Такі змінні називаються ***динамічними***.

Розглянемо, наприклад, таку послідовність описів:

```
type
   TDateRec = record
                Day: 1..31;
                Month: string[8];
                Year: Integer;
   end;
   TDatePtr = ^TDateRec;
var
   DateVar: TDatePtr;
```

Другий опис типу констатує, що будь-який об'єкт типу `TDatePtr` є вказівником, що буде вказувати на змінну типу `TDateRec`. Жодні змінні типу `TDateRec` можуть не оголошуватися: вони будуть спеціальним способом створюватися в купі за ходом виконання програми. Якщо змінна вже створена, то конструкція `DateVar^` (ім'я змінної-вказівника з символом `^`) розглядається як своєрідне ім'я цієї змінної, тобто `DateVar^` – це посилання на динамічну змінну типу `TDateRec`. Посилаючись за допомогою вказівника на динамічні змінні, можна оперувати ними, не знаючи при цьому, де вони розташовуються.

7.3.2. Процедури та функції, що використовуються при роботі з динамічними даними

Для роботи з динамічними змінними в Delphi введено процедури `New`, `Dispose`, `GetMem`, `FreeMem`, `Mark` і `Release`. Крім того, перед розміщенням у пам'яті динамічних даних дуже часто вживають функцію `SizeOf`.

Процедура

`New(`ім'я_вказівника`)`

розміщає в пам'яті динамічну змінну з ім'ям, що задається параметром процедури (наприклад, оператор `New(DateVar)` створює в динамічній пам'яті нову динамічну змінну з адресою, записаною в змінній `DateVar`; для звертання до цієї змінної потрібно розійменувати вказівник: `DateVar^`).

При відведенні динамічної пам'яті необхідно контролювати її наявність, для чого застосовується технологія обробки виняткових ситуацій (див. розд. 13). Якщо для розміщення динамічної змінної бракує достатньої кількості доступної пам'яті, то процедура `New` генерує виняток `EOutOfMemory`.

Протилежною процедурі `New` є процедура

`Dispose(`ім'я_вказівника`)`,

що звільняє ділянку пам'яті, на яку посилається вказівник, заданий як параметр (наприклад, оператор `Dispose(DateVar)` звільняє пам'ять, що займала змінна `DateVar^`). При цьому звільнена ділянка пам'яті повертається в системну ділянку, що називається *пулом вільної пам'яті*. У результаті застосування процедури `Dispose` у пам'яті з'являються вільні ділянки (кластери), які не завжди можуть бути надалі використані через те, що вони можуть бути маленькими за обсягом.

```
//Приклад 7.4
//У файлі f.dat зберігається 18000 дійсних чисел,
//що є елементами масиву з 10 рядків та 1800
//стовпців. Переписати в інший файл рядок
//з максимальною сумою елементів.
```

Скористаємося формою з прикладу 6.1 і напишемо такий опрацьовувач події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
type
   T1 = array[1..1800] of Real;
   T2 = array[1..10] of ^T1;
var
  a: T2;                    //Масив з 10 вказівників на масиви
  i, j, n: Integer;
  max, s: Real;
  f: file of Real;
begin
  AssignFile(f, 'f.dat');
  Reset(f);
        //Нижче імітується і заповнюється «двовимірний» масив
  for i := 1 to 10 do begin
    New(a[i]);      //Створення і-го рядка двовимірного масиву
    s := 0;
    for j := 1 to 1800 do begin
      Read(f, a[i]^[j]);//Читаємо в елемент динамічного масиву
      s := s + a[i]^[j];
    end;
    if (i = 1) or (s > max) then n := i;
  end;
  CloseFile(f);
  AssignFile(f,'f1.dat');
  Rewrite(f);
       //Цикл виведення динамічного масиву a[n]^ за елементами
  for j := 1 to 1800 do
    Write(f, a[n]^[j]);
  CloseFile(f);
  for i := 1 to 10 do       //Цикл видалення масиву за рядками
    Dispose(a[i]);             //(десяти одновимірних масивів)
  Button1.Enabled := False;
end;
```

Тип T2 описує масив з 10 елементів, кожний з яких є вказівником на одновимірний масив з 1800 елементів типу Real. За допомогою цього типу описано масив вказівників a, елементи якого (що є одновимірними масивами) створюються в динамічній пам'яті процедурою New. Доступ до створених одновимірних масивів здійснюється за допомогою операції ^.

Таким чином, конструкція вигляду `a[i]^` є уточненим ім'ям одновимірного динамічного масиву, конструкція вигляду `a[i]^[j]` – уточненим ім'ям елемента двовимірного масиву, що імітується за допомогою масиву вказівників `a`. Звільнення динамічної пам'яті здійснюється процедурою `Dispose` у циклі, що забезпечує видалення всіх одновимірних динамічних масивів.

Зазначимо, що розглянута задача є чисто ілюстративною задачею на використання процедур `New` та `Dispose`. Для створення та знищення динамічних масивів (у тому числі і багатовимірних) у Delphi передбачено процедури `SetLength` та `Finalize` (див. п. 5.1.2)

Процедури `New` та `Dispose` виділяють і звільняють пам'ять відповідно до того, який розмір пам'яті визначає створюваний тип (наприклад, у наведеній вище процедурі кожна зі змінних `a[i]^` займає по $1800 \times 8 = 14400$ байт). Дещо інакше працюють процедури `GetMem` та `FreeMem`.

Процедура

```
GetMem(ім'я_вказівника, кількість_байт)
```

забезпечує виділення пам'яті під динамічну змінну розміром, що задається другим параметром. Процедура ж

```
FreeMem(ім'я_вказівника, кількість_байт)
```

звільняє задану кількість байт динамічної пам'яті, починаючи з адреси, на яку вказує змінна, що задана як перший параметр. Перший параметр у цих двох процедурах має тип `Pointer`, що дозволяє підставляти замість нього вказівник-змінну з будь-яким базовим типом.

Необхідно дотримуватися такого правила: змінні, розподілені за допомогою `New`, повинні знищуватися процедурою `Dispose`, а пам'ять, відведена за допомогою `GetMem`, повинна звільнятися процедурою `FreeMem` з тим же значенням другого параметра, що й у відповідному звертанні до процедури `GetMem` (змінні, створені за допомогою `New`, можуть знищуватися і процедурою `FreeMem`).

Пам'ять, що відводиться за допомогою `GetMem`, є блоком пам'яті, який можна адресувати за допомогою вказівника. Цей блок пам'яті розглядається як послідовність елементів одного і того самого типу, причому тип цих елементів визначається не їхнім вмістом і методом запису в них значень, а базовим типом вказівника, за допомогою якого забезпечується доступ до цих елементів. Тому потрібно забезпечувати відповідність базового типу вказівника формату, у якому зберігаються записані у пам'ять значення (контроль у цьому випадку не проводиться в

припущенні, що він забезпечується програмістом на етапі написання програми).

Розглянемо найпростіший приклад відведення блоку динамічної пам'яті за допомогою процедури GetMem і доступу до цього блоку пам'яті за допомогою вказівника. Даний приклад є чисто ілюстративним, оскільки подібні задача звичайно розв'язують із використанням динамічних масивів, доступ до елементів яких забезпечується за допомогою індексування.

```
//Приклад 7.5
//У файлі f.dat зберігаються дані у форматі дійсних
//чисел типу Real. Видалити з цього файлу найбільше
//значення (якщо їх декілька, то перше з них).
```

Скористаємося формою з прикладу 6.1 і напишемо такий опрацьовувач події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Pt1, Pt2, Pt:^Real;
  i, n, nmax: Integer;
  max: Real;
  f: file of Real;
begin
  AssignFile(f, 'f.dat');
  Reset(f);
  n := FileSize(f);       //Визначаємо кількість чисел у файлі
  GetMem(Pt1, n * SizeOf(Real));           //Відводимо пам'ять
  Pt2:=Pt1; //На початок області пам'яті вказують 2 вказівники
  for i := 1 to n do begin       //Читаємо з файлу й пишемо за
    Read(f, Pt2^); //вказівником Pt2^, зсуваючи його в пам'яті
    Inc(Pt2);      //збільшенням його значення на SizeOf(Real)
  end;
  Pt2 := Pt1;//Повертаємо у Pt2 адресу початку області пам'яті
  max := Pt1^;
  Pt := Pt1;   //У Pt запишеться адреса максимального елемента
  for i := 1 to n do begin
    if Pt2^ > Pt^ then Pt := Pt2;
    Inc(Pt2);
  end;
             //Значення PChar(Pt)-PChar(Pt1) визначає відстань
            //у байтах від початку області пам'яті (масиву) до
                                          //найбільшого елемента
  nmax := (PChar(Pt) - PChar(Pt1)) div SizeOf(Real) + 1;
                           //nmax – номер найбільшого елемента
  for i := nmax to n - 1 do begin           //Працюючи з двома
    Pt2 := Pt;       //вказівниками, зсуваємо елементи на одну
    Inc(Pt2);           //позицію вліво, починаючи з елемента,
    Pt^ := Pt2^;                    //що стоїть за максимальним
```

```
    Inc(Pt);
  end;
  Rewrite(f);
  Pt2 := Pt1;      //Переписуємо n-1 елементів, що залишилися,
  for i := 1 to n - 1 do begin    //у файл, адресуючи елементи
    Write(f, Pt2^);              //за допомогою вказівника Pt2
    Inc(Pt2);                                 //і зсуваючи його
  end;
  CloseFile(f);
  FreeMem(Pt1, n * SizeOf(Real));          //Звільняємо пам'ять
  Button1.Enabled := False;
end;
```

У процедурі `TForm1.Button1Click` спочатку виділяється блок пам'яті відповідно до розміру файлу, а адреса початку цього блоку запам'ятовується у вказівнику `Pt1`. Надалі цей вказівник зберігає своє значення з метою забезпечення наступного доступу до початку цього блоку і його звільнення за допомогою `FreeMem`. Вказівник `Pt2` використовується для переміщення по блоку пам'яті з трактуванням його вмісту як послідовності елементів типу `Real`. Вказівник `Pt2` змінює своє значення за допомогою процедури `Inc`, що забезпечує послідовний його зсув на 8 байт (відповідно до розміру базового типу). Адреса найбільшого елемента фіксується у вказівнику `Pt`. Якщо при цьому виділений блок пам'яті трактувати як масив елементів типу `Real`, то для визначення місця розташування максимального елемента (значення змінної `nmax`) потрібно визначити зсув у байтах вказівника `Pt` щодо вказівника `Pt1` (визначається різницею `PChar(Pt) - PChar(Pt1)`) і розділити його на розмір базового типу. Необхідність приведення типу вказівника до типу `PChar` пояснюється тим, що різниця двох вказівників визначена тільки для типу `PChar`. Звільнення динамічної пам'яті проводиться процедурою `FreeMem` зі значенням другого параметра, що збігається зі значенням другого параметра при виділенні пам'яті за допомогою процедури `GetMem`.

Слід, по можливості, знищувати динамічні змінні відразу ж, як тільки відпадає необхідність у них, інакше може досить швидко вичерпатися вільна пам'ять.

7.4. Зв'язані списки

Вказівники є ефективним засобом для побудови зв'язаних списків, тобто таких упорядкованих структур, кожен елемент яких містить посилання, що зв'язує його з попереднім або наступним елементом (можливе використання й більш складних структур).

У Delphi існують значно ефективніші способи побудови впорядкованих структур даних, які також називаються списками. Фактично ці структури даних являють собою масиви вказівників, які забезпечують довільний доступ до їх елементів. Мало того, для таких структур даних існує сукупність підпрограм, що забезпечують керування ними. У той же час освоєння методів роботи зі зв'язаними списками дозволяє, з одного боку, поглибити розуміння вказівників і, з іншого боку, відпрацювати техніку побудови алгоритмів і написання програм. Крім того, зв'язані списки носять прикладний характер, і необхідність у їх використанні зустрічається в багатьох задачах керування.

Для організації зв'язаних списків застосовують записи, що складаються з двох змістовних частин – основної і додаткової. Основна частина містить інформацію, що підлягає обробці, а в додатковій частині розміщено вказівник на інший елемент списку. Вказівник на кінець списку зберігається в змінній, котра завжди присутня у програмі обробки списків. Якщо в списку немає елементів, тобто він порожній, значення цієї змінної повинне дорівнювати **nil**. У противному випадку його перший (останній) елемент містить у додатковій частині **nil**.

Основними операціями над списками є формування списку, додавання, вставка, видалення і перегляд (обробка) елементів.

Найпростішими є два види списків – стек і черга.

Стек – це список з однією точкою доступу до його елементів, яка називається вершиною стека. Додати або видалити елемент можна тільки через вершину стека. Принцип обслуговування стека коротко формулюється так: «останній прийшов – перший вийшов». Це правило інакше називається правилом LIFO (від *Last In – First Out*). У цьому випадку йдеться про обслуговування елементів стека і їх надходження в стек.

Черга – це структура даних, в один кінець якої додаються елементи, а з іншого вилучаються. Принцип роботи черги: «перший прийшов – перший вийшов». Дане правило називається правилом FIFO (від *First In – First Out*).

Щоб забезпечити можливість роботи зі списком, потрібно включити вказівник до складу запису, що можна зробити, наприклад, так:

```
type
  TPt = ^T;
  T = record
    Pt: TPt;
    surname: string;
  end;
var
  PtTop, start: TPt;
```

Тут фігурує виняток із загального правила Delphi (не можна користуватися яким-небудь ім'ям доти, доки воно не буде описане): в описі типу `TPt` міститься посилання на тип `T`, який визначається пізніше. Саме завдяки цьому винятку можливе створення зв'язаних структур даних.

Тепер непотрібно створювати масив з вказівників, оскільки кожен новий запис містить посилання на попередній.

Для роботи зі стеком, крім вказівника усередині запису, необхідний вказівник для зберігання адреси, за якою розташовується вершина стека (таким вказівником, наприклад, може бути змінна `start`).

Розглянемо процес створення, обробки та знищення стека на прикладі наступної програми.

```
//Приклад 7.6
//Водяться прізвища. Ознака закінчення введення – символ @
//як перша літера прізвища. Розташувати дані в стеку.
//Вивести вміст стека без знищення списку (стека).
//Якщо в стеку є прізвище "Іванов", вставити в список
//перед цим прізвищем (у порядку обслуговування) прізвище
//"Іванова", якщо немає – у кінець. Вивести вміст стека
//зі знищенням його елементів.
```

Розмістимо на формі з прикладу 3.1 ще один компонент Button (його властивість Name автоматично дістане значення Button2), наклавши цей компонент на раніше розміщену кнопку введення, і встановимо для властивості Caption цього компонента значення Вставити. Змінимо також значення властивостей деяких інших компонентів:

- Мітка:
  Caption — Уведіть прізвище або @
- Поле введення:
  Text — @
- Кнопка Button1:
  Caption — Введення

Крім того, в секції **`interface`** модуля перед описом класу `TForm1` оголосимо наступні типи:

```
type
  TPtElement = ^TElement;
  TElement = record
     surname: string;                    //Основне поле
     Pt: TPtElement                      //Поле-вказівник
  end;
```

а в секції **`public`** опису класу `TForm1` опишемо поле `PtTop` типу `TPtElement`.

Тоді тексти опрацьовувачів події OnClick компонентів Button1 і Button2 можуть бути наступними:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Pt1: TPtElement;
  s: string;
begin
  if Tag = 0 then begin
    PtTop := nil;                                  //Стек порожній
    Tag := 1;
  end;
  s := edInput1.Text;
  if s[1] <> '@'
  then begin
    New(Pt1);                            //Створення елемента стека
    Pt1^.surname := s;                    //Значення основного поля
    Pt1^.Pt := PtTop;           //Посилання на попередній елемент
    PtTop := Pt1;                      //Переміщаємо вершину стека
    edInput1.SetFocus;
  end
  else begin
    if PtTop = nil then mmOutput1.Lines.Add('Стек порожній')
    else
      mmOutput1.Lines.Add('Вміст стека');
      Pt1 := PtTop;               //Переміщаємося у вершину стека
      while Pt1 <> nil do begin          //Цикл виведення стека
        mmOutput1.Lines.Add(Pt1^.surname);
        Pt1 := Pt1^.Pt;      //Перехід до попереднього елемента
      end;
      edInput1.Visible := False;
      lbOutput1.Visible := False;
      Button1.Visible := False;
      Button2.Visible := True;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  Pt1, Pt2, v: TPtElement;
begin
  New(v);          //Створення елемента, що вставляється в стек
  v^.surname := 'Іванова';
  if (PtTop = nil) or (PtTop^.surname = 'Іванов') then begin
   //Якщо стек порожній або перше прізвище - 'Іванов', вставка
    v^.Pt := PtTop; //у початок стека в порядку обслуговування
    PtTop := v;
  end
```

```
  else begin          //Підтримуємо два вказівники (Pt1 і Pt2)
    Pt1 := PtTop;                    //на два сусідніх елементи
    Pt2 := PtTop^.Pt;
     //Пошук елемента до знаходження його або вичерпання стека
    while not ((Pt2 = nil) or (Pt2^.surname = 'Іванов')) do
    begin
      Pt1 := Pt2;
      Pt2 := Pt2^.Pt
    end;
    Pt1^.Pt := v;                     //Вставка нового елемента
    v^.Pt := Pt2;                                  //між Pt1 і Pt2
  end;
  Pt1 := PtTop;                     //Повернення у вершину стека
  mmOutput1.Lines.Add('Результат вставки');
  while PtTop <> nil do begin       //Цикл до вичерпання стека
     mmOutput1.Lines.Add(PtTop^.surname); //Виведення елемента
     PtTop := PtTop^.Pt;    //Перехід до попереднього елемента
     Dispose(Pt1);                                    //Знищення
     Pt1 := PtTop;                //Суміщення двох вказівників
  end;
  Button2.Enabled := False;
end;
```

Створення стека проводиться в неявному циклі, пов'язаному з кліками мишкою над компонентом Button1. При створенні стека використовуються дві змінні (у програмі `PtTop` і `Pt1`), одна з яких (`PtTop`) підтримує посилання на вершину стека, а друга (`Pt1`) – посилання на створений елемент. Перед створенням стека в `PtTop` заноситься значення **`nil`** (рис. 7.1, *а*). На першому кроці циклу спочатку створюється новий елемент (змінна `Pt1^`), адреса якого (позначимо її через ***A1***) заноситься в змінну `Pt1` (рис. 7.1, *б*). Після виконання оператора `Pt1^.Pt:=PtTop` створена динамічна змінна дістає посилання на попередній елемент (на першій ітерації – це **`nil`**, результатом чого є те, що перший елемент стека посилається на **`nil`**). Наприкінці ітерації (`PtTop:=Pt1`) змінна `PtTop` також переміщається у вершину стека і буде зберігати адресу ***A1*** (рис. 7.1, *в*). При виході з процедури `TForm1.Button2Click` змінна `Pt1` знищується, але доступ до вершини стека зберігається завдяки наявності змінної `PtTop` (рис. 7.1, *г*). На другій ітерації знову створюється елемент `Pt1^` (з адресою ***A2*** у змінній `Pt1`, див. рис. 7.1, *д*), а оператор `Pt1^.Pt:=PtTop` заносить у її поле `Pt` адресу ***A1*** попереднього елемента, після чого змінна `PtTop` дістає значення ***A2*** (рис. 7.1, *е*). У підсумку два елементи стека виявляються зв'язаними так, що знову створений елемент «знає», де розташовується раніше створений. Перший елемент посилається на **`nil`**, завер-

шуючи тим самим стек, а на останній елемент (вершину стека) посилається вказівник PtTop (рис. 7.1, *ж*).

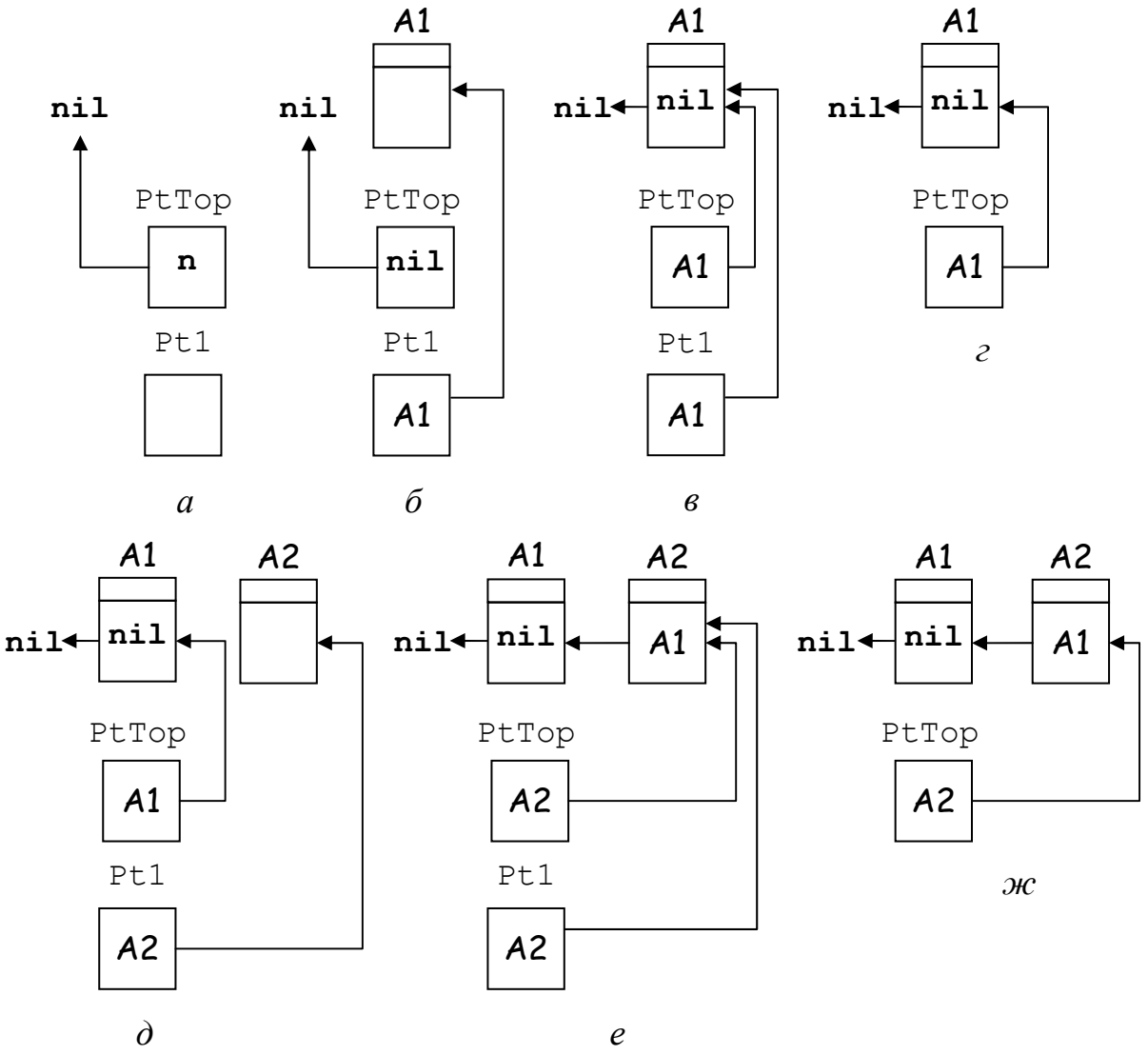


Рис 7.1 – Процес створення стека

Вставка нового елемента виконується за кліком мишкою над компонентом Button2 за допомогою процедури TForm1.Button2Click. Якщо до моменту вставки нового елемента стек порожній (про що свідчить виконання умови PtTop=**nil**), то новостворений елемент стає вершиною стека (оператори v^.Pt := PtTop та PtTop := v). Для забезпечення правильної вставки новоствореного елемента усередину стека застосовують два вказівники (Pt1 і Pt2), які адресують два сусідніх елементи стека.

При обробці даних, що зберігаються в стеці (прикладом чого є виведення вмісту стека), проводиться перебір елементів (адрес) до досягнення значення **nil**. Якщо перегляд елементів повинен супроводжуватися їх

знищенням (як це має місце в циклі виведення в процедурі TForm1.Button2Click), то при переборі необхідно використовувати два вказівники, один з яких (PtTop) призначений власне для переходу від одного елемента до іншого, а другий (Pt1) – для знищення розглянутого елемента після того, як виконано перехід до попереднього (у порядку створення) елементу.

Процеси створення й обробки черги подібні з аналогічними процесами для стека. Відмінність зумовлена, насамперед, необхідністю підтримання в черзі двох точок входу (вказівників) – адреси початку (для обробки) й адреси кінця (для поповнення) черги.

```
//Приклад 7.7
//Вводяться дійсні числа до введення першого
//від'ємного. Вивести  в порядку введення всі
//введені числа, попередньо виключивши мінімальний
//елемент (якщо їх декілька, то виключити перший).
```

Скористаємося формою з попередньої задачі, змінивши значення властивостей деяких компонентів:

- Мітка:
  Caption — Уведіть дійсне число (кінець введення – число <0)
- Поле введення:
  Text — -1
- Кнопка Button2:
  Caption — Видалити

Крім того, у секції **interface** модуля перед описом класу TForm1 оголосимо наступні типи:

```
type
  TPtElement = ^TElement;
  TElement = record
         value: Real;                    //Основне поле
         Pt: TPtElement;                 //Поле-вказівник
  end;
```

а в секції **public** опису класу TForm1 опишемо поля start і finish типу TPtElement.

Тоді тексти опрацьовувачів події OnClick компонентів Button1 і Button2 можуть бути наступними:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Pt1, Pt2: TPtElement;
  v: Real;
begin
```

```
  if Tag = 0 then begin
    start := nil; finish := nil;                //Черга порожня
    Tag := 1;
    DecimalSeparator := '.';
  end;
  v := StrToFloat(edInput1.Text);
  if v >= 0 then begin
    New(Pt1);                             //Створюємо новий елемент
    Pt1^.value := v;                      //Визначаємо основне поле
    if start = nil then start := Pt1           //Якщо це перший
                        //елемент, запам'ятовуємо початок черги
    else finish^.Pt := Pt1;          //Інакше попередній елемент
                                              //зв'язуємо з новим
    Pt1^.Pt := nil;           //Останній елемент зв'язуємо з nil
    finish := Pt1;           //Запам'ятовуємо адресу кінця черги
    edInput1.SetFocus;
  end
  else begin
    if start = nil then
      mmOutput1.Lines.Add('Черга порожня')
    else begin
      mmOutput1.Lines.Add('Вміст черги');
      Pt1 := start;
      while Pt1 <> nil do begin
        mmOutput1.Lines.Add(FloatToStr(Pt1^.value));
        Pt1 := Pt1^.Pt;
      end;
      Button2.Visible := True;
    end;
    edInput1.Visible := False;
    lbOutput1.Visible := False;
    Button1.Visible := False;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  min: Real;
  Pt1, Pt2: TPtElement;
begin
  min := start^.value;                  //Починаємо шукати мінімум
  Pt1 := start^.Pt;      //Переходимо до другого елемента черги
  while Pt1 <> nil do begin                 //Цикл до кінця черги
    if Pt1^.value < min then min := Pt1^.value;
    Pt1 := Pt1^.Pt;             //Перехід до наступного елемента
  end;
  Pt2 := start;                    //Запам'ятовуємо початок черги
  if start^.value = min then     //Перший елемент - мінімальний
```

```
    start := start^.Pt                    //Зміщуємо початок черги
  else                              //Перший елемент - не мінімальний
  begin
    Pt1 := start;                         //Знову на початок черги
    Pt2 := start^.Pt;
    while Pt2^.value <> min do begin
      Pt1 := Pt2; Pt2 := Pt2^.Pt;
    end;              //Pt1 - перед мінімумом, Pt2 - на мінімумі
    Pt1^.Pt := Pt2^.Pt;               //Зв'язуємо елементи перед
  end;                               //мінімумом і після мінімуму
  Dispose(Pt2);   //Видаляємо елемент із мінімальним значенням
  Pt1 := start;                           //Знову на початок черги
  mmOutput1.Lines.Add('Черга після видалення');
  if start = nil then
    mmOutput1.Lines.Add('Черга порожня')
  else    //Для видалення потрібні два вказівники: Pt1 і start
    while start <> nil do begin          //Цикл до кінця черги
      mmOutput1.Lines.Add(FloatToStr(Pt1^.value));
      start := start^.Pt;               //Перехід до наступного
      Dispose(Pt1);                        //Знищення елемента
      Pt1 := start;                     //Суміщення вказівників
    end;
  Button2.Enabled := False;
end;
```

На початку роботи програми (при `Tag=0`) змінні `start` і `finish` дістають значення **`nil`**, що свідчить про порожність черги. Властивість `Tag` отримує значення `1`, яке говорить про те, що хоча б одне значення вже вводилося. При створенні першого елемента змінна `start` дістає посилання на нього і більше не змінюється, зберігаючи посилання на початок черги. У першого елемента немає попереднього, тому зв'язок попереднього елемента з новостворюваним здійснюється тільки з моменту створення другого елемента черги (коли значення вказівника `start` є відмінним від **`nil`**). У всіх випадках останній створений елемент отримує посилання на **`nil`** (`Pt1^.Pt := nil`), а змінна `finish` – посилання на останній створений елемент, зберігаючи тим самим посилання на кінець черги. Для перегляду (обробки) елементів черги (виведення і пошук мінімуму) достатньо однієї змінної (`Pt1`), у той час як обробка черги з її знищенням (в опрацьовувачі `TForm1.Button2Click` – виведення на екран) вимагає використання двох посилальних змінних (в опрацьовувачі `TForm1.Button2Click` – це `start` і `Pt1`).

У процедурі `TForm1.Button2Click` показано принцип видалення елементів з черги: для цього достатньо зв'язати два елементи – перед елементом, що видаляється, і безпосередньо наступний за ним.

Зазначимо, що знищення за допомогою `Dispose` вилучених з черги (або стека) елементів обов'язкове, але воно бажане через те, що в противному випадку в динамічній пам'яті залишаються зайняті ділянки пам'яті, на які відсутні посилання.

Можуть бути побудовані й інші зв'язані списки. Такими, наприклад, є двозв'язаний список (дек), що є комбінацією стека і черги, кільцевий список (у ньому останній елемент зв'язаний з першим).

Запитання для контролю і самоконтролю

1. Що таке динамічна змінна?
2. Що таке вказівник?
3. Яка операція дозволяє отримати значення, записане за адресою, що зберігається у вказівнику?
4. У чому різниця між значенням, що зберігається у вказівнику, і значенням, що адресується вказівником?
5. Що таке операція розіймення?
6. У чому різниця між операціями `^` та `@`?
7. Як описуються типізовані та нетипізовані вказівники?
8. Поясніть зміст фрагменту програми:

```
var
  v: Integer;
  pInt: ^Integer;
     //...
  pInt := @v;
```

9. Знайдіть помилку в такому фрагменті програми:

```
var
  pInt: ^Integer;
     //...
  ^pInt := 13;
  Memo1.Lines.Add(IntToStr(^pInt));
```

10. Які засоби існують у Delphi для роботи з динамічними змінними?
11. Що таке динамічні списки і як вони організуються?

Завдання для практичного відпрацьовування матеріалу

1. За допомогою вказівника запишіть у змінну `myAge`, що має тип `Integer`,значення 17.

2. Дано n цілих чисел, де n – натуральне число. Сформувати новий масив, що містить тільки ті елементи вихідного масиву, які є простими числами.
3. Дано матрицю (прямокутну таблицю чисел) $\boldsymbol{A}$ розміру $m \times n$, де m і n – натуральні числа (відповідно кількість рядків і стовпців). Одержати транспоновану матрицю $\boldsymbol{A}^T$ та добутки $\boldsymbol{A}\boldsymbol{A}^T$ і $\boldsymbol{A}^T\boldsymbol{A}$. Де сума елементів більша?

   Транспонована матриця утворюється з початкової записом рядків у вигляді стовпців (або, що те ж саме, стовпців у вигляді рядків).

   Добутком двох матриць $\boldsymbol{A}$ (розміру $m \times n$) і $\boldsymbol{B}$ (розміру $n \times p$) називається така матриця $\boldsymbol{C}$ (розміру $m \times p$), елементи якої дорівнюють сумі попарних добутків відповідних елементів рядка першої матриці та стовпця другої матриці:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, i = 1, 2, \ldots, m, j = 1, 2, \ldots, p.$$

4. Створити в пам'яті структуру, що імітує двовимірний масив з рядками різної довжини, записати в цей «масив» трикутник Паскаля і вивести його на екран.

   Трикутником Паскаля називається числовий трикутник, у якому по краях рядків розташовуються одиниці, а кожне число, що стоїть усередині рядка дорівнює сумі двох чисел, що стоять над ним у найближчому рядку зверху:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
. . . . . .

5. Дано числовий масив розміру $m \times n$, де m та n – натуральні числа. Не переміщаючи елементи масиву в пам'яті, виконати впорядкування рядків масиву за убуванням суми їх елементів.
6. Дано натуральне число n і цілі числа $a_1, a_2, \ldots, a_n$. Організувати стек, що містить значення $x_1, y_1, x_2, y_2, \ldots, x_k, y_k$, де $x_1, x_2, \ldots, x_m$ – узяті в порядку проходження парні члени послідовності $a_1, a_2, \ldots, a_n$, а $y_1, y_2, \ldots, y_l$ – непарні члени, $k = \min(m, l)$.
7. Дано натуральне число n. Одержати послідовність $d_k, d_{k-1}, \ldots, d_0$ десяткових цифр числа $n!$, тобто таку послідовність, у якій кожен член d_i задовольняє умовам $0 \le d_i \le 9$ і $d_k \cdot 10^k + d_{k-1} \cdot 10^{k-1} + \ldots + d_0 = n!$. Скористатися для запису послідовності зв'язаним списком.

8. Ввести список академічної групи за алфавітом та помістити його в динамічній пам’яті у вигляді черги. У групу зарахований ще один студент. Помістити його прізвище в черзі, зберігши алфавітний порядок.
9. Розмістити в черзі n цілих чисел, де n – натуральне число, після чого перед кожним парним числом помістити значення, що дорівнює половині цього числа.
10. Дано натуральне число n. Розмістити в динамічному стеку $2n$ дійсних чисел. Розглядаючи елементи стека відповідно до порядку обслуговування, виконати таке перетворення списку: якщо число з парним порядковим номером менше за попереднє в списку, змінити їх порядок проходження в списку, не змінюючи розташування в пам’яті.
11. Дано натуральні значення n і m. Припускається, що n людей встають у коло й отримують номери, лічачи проти годинникової стрілки. Потім, починаючи з першого, також проти годинникової стрілки відлічується m-та людина (за m-ю людиною стоїть перша, оскільки люди стоять по колу). Ця людина виходить з кола і її номер поміщається в чергу, після чого, починаючи з наступного, знову відлічується m-та людина. Визначити початковий номер останньої вибулої людини та вивести вміст сформованої черги. Підказка: тут може допомогти кільцевий список.

8. ПІДПРОГРАМИ

8.1. Функції

Нехай потрібно написати програму обчислення значення факторіала числа 12 (12! = 1·2·…·12).

```
//Приклад 8.1.1
//Обчислити факторіал числа 12 (12! = 1·2·…·12).
```

Для розв'язання задачі скористаємося формою з прикладу 2.2, видаливши з неї мітку та однорядкове редаговане поле й оголосивши в секції **public** опису класу TForm1 поле factorial типу Integer.

Процес розв'язання задачі реалізує наступний опрацьовувач події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  factorial := 1;
  for i := 1 to 12 do
     factorial := factorial * i;
  mmOutput1.Lines.Add('12! = ' + IntToStr(factorial));
  Button1.Enabled := False;
end;
```

Якщо нам знадобиться обчислити не 12!, а 7!, то в програмі всюди потрібно буде замінити 12 на 7. Це легко зробити в даній програмі, але як бути у випадку написання програми для розв'язання складної задачі, особливо коли змінюване значення використовується в програмі в декількох змістах? Тут на допомогу можуть прийти константи, адже зміни тоді виконуватимуться тільки в розділі опису констант.

```
//Приклад 8.1.2
//Обчислити факторіал числа 12 (12! = 1·2·…·12).
```

Скористаємося тією же формою, що й у прикладі 8.1.1, не змінюючи опис класу TForm1.

У секції **interface** модуля одразу ж за розділом **uses** опишемо константу N:

```
const
  N = 12;
```

Внесемо також невеликі зміни в текст опрацьовувача події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  factorial := 1;
  for i := 1 to N do
    factorial := factorial * i;
  mmOutput1.Lines.Add(IntToStr(N)+ '!=' +IntToStr(factorial));
  Button1.Enabled := False;
end;
```

Тепер, якщо буде потрібно обчислити факторіал числа, відмінного від 12, достатньо змінити значення константи N на початку модуля.

Найчастіше більш розумним є використання змінної, що забезпечує незалежність тексту програми від вихідних даних.

```
//Приклад 8.1.3
//Обчислити факторіал числа N, 0 <= N <= 12 (N! = 1·2·…·N).
```

Скористаємося формою з прикладу 2.2, записавши у властивість Caption мітки текст Уведіть ціле невід'ємне число до 12. Оголосимо в секції **public** опису класу TForm1 поля N і factorial типу Integer і створимо такий опрацьовувач події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  N := StrToInt(edInput1.Text);
  if (N < 0) or (N > 12) then begin
    mmOutput1.Lines.Add('Повторіть введення');
    edInput1.SetFocus;
  end
  else begin
    factorial := 1;
    for i := 1 to N do
      factorial := factorial * i;
    mmOutput1.Lines.Add(IntToStr(N)+'!='+IntToStr(factorial));
    Button1.Enabled := False;
  end;
end;
```

Тепер у програмі передбачене введення того значення, для якого обчислюється факторіал. Мало того, реалізується контроль на допустимість значення, що вводиться.

Але як бути, якщо в якій-небудь складній програмі в різних місцях потрібно буде обчислювати факторіали різних значень? Що ж, повторювати ті самі оператори, які реалізують метод обчислення факторіала багаторазово? Так, можлива і така реалізація, але це нерозумно. Більш доцільно застосувати підпрограму-функцію. Напишемо подібну програму, але таку, що використовує функцію.

```
//Приклад 8.2
//Дано N об'єктів, N<=12. Скільки можна сформувати з
//них різних сполучень по M об'єктів. Число сполучень
//з N по M визначається формулою N!/((N-M)!*M!).
```

Скористаємося формою з прикладу 8.1.3, записавши у властивість Caption мітки текст Уведіть через пробіл N та M (N>=M) й оголосивши в секції **public** опису класу TForm1 поля N, M і CNM типу Integer. Крім того, у секції **implementation** опишемо так звану підпрограму-функцію:

```
function factorial(k: Integer): Integer;
var
  i, f: Integer;
begin
  f := 1;
  for i := 1 to k do
    f := f * i;
  Result := f;
end;
```

Створимо також наступний опрацьовувач події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  s: string;
begin
  s := Trim(edInput1.Text);
  if Pos(' ', s) > 0 then begin
    N := StrToInt(Copy(s, 1, Pos(' ', s) - 1));
    Delete(s, 1, Pos(' ', s));
    M := StrToInt(Trim(s));
  end
  else N := 0;                    //Ознака помилки при введенні
  if (N <= 0) or (M <= 0) or (N > 12) or (M > 12) then begin
    mmOutput1.Lines.Add('Повторіть введення');
    edInput1.SetFocus;
```

```
  end
  else begin
    CNM := factorial(N) div factorial(N - M) div factorial(M);
    mmOutput1.Lines.Add('Число сполучень з ' +
                        IntToStr(N) + ' по ' + IntToStr(M) +
                        ' дорівнює ' + IntToStr(CNM));
    Button1.Enabled := False;
  end;
end;
```

Зазначимо, що в опрацьовувачі події OnClick компонента Button1 передбачено контроль правильності введення, зокрема, якщо в поле введення однорядкового редактора замість двох цілих чисел випадково буде введене одне, змінній `N` буде присвоєне значення `0`, що свідчитиме про помилку введення (але формат даних, які вводяться, не контролюється).

Як це видно з прикладу 8.2, використання функцій дає дві основних переваги:

1) відокремлюється хід розв'язання від конкретної реалізації підпрограми: немає потреби знати, як реалізована підпрограма, потрібно тільки вміти до неї звернутися;

2) опис функції в програмі дається один раз, а використовувати її можна багаторазово (у прикладі 8.2 замість того, щоб тричі реалізовувати метод обчислення факторіала, виконане триразове звертання до оголошеної на початку розділу **implementation** функції `factorial`).

Розглянемо особливості опису функції і звертання до неї.

1. Функція оголошується в розділі описів. Її можна оголосити як у секції **implementation** модуля (у цьому випадку вона буде відома всюди нижче точки опису), так і у розділі опису будь-якої іншої підпрограми (правда, у цьому випадку вона буде відома тільки усередині цієї підпрограми).

2. Функція може бути оголошена також і в секції **interface** модуля. У цьому випадку в секції **interface** оголошується тільки заголовок функції, а в секції **implementation** дається її повне визначення. Такий спосіб опису функції робить її відомою у всіх модулях, до яких підключений даний модуль (див. підрозд. 9.1).

3. Функції можуть бути описані й у розділі описів проекту, але доступ до них у такому разі можливий тільки за допомогою операторів, записаних у виконуваній частині проекту. Оскільки програміст змінює проект дуже рідко, подібного роду опис є практично безглуздим.

4. Заголовок функції починається зі службового слова **function**, зразу за яким йде ім'я функції.

5. Після імені функції в круглих дужках зазначаються формальні параметри та їхні типи (у прикладі 8.2 функція `factorial` має один параметр `k` типу `Integer`). Якщо параметрів декілька, то вони оголошуються через крапку з комою. Якщо кілька параметрів мають один і той самий тип, то ці параметри можна подати через кому з вказівкою спільного типу; наприклад,

```
function fun(a, b, c: Real; d: Integer): Real;
```

6. Якщо функція не має формальних параметрів, то круглі дужки в заголовку не ставляться (в Delphi дозволено ставити круглі дужки й у цьому випадку).

7. Після дужок, у які укладені формальні параметри, через двокрапку вказують тип функції (тип значення, що повертається). У праматері Delphi – мові Turbo Pascal функція служила для обчислення тільки одного значення, що передавалося в програму через ім'я функції, внаслідок чого типом значення, що поверталося, міг бути будь-який скалярний тип (числовий, символьний, булівський, перелічений), рядковий тип або вказівник. У Delphi типом функції може бути довільний тип (крім файлових).

8. Тип значення, що повертається, може бути заданий тільки ім'ям. Не можна, наприклад, вказати тип **string**`[20]` – у цьому випадку у програмі слід ввести новий тип і використати його при зазначенні типу функції (наприклад, **type** `t_st20 =` **string**`[20]`).

9. Тип формальних параметрів теж задається ім'ям (можливе також застосування типу «відкритий масив»; наприклад, **array of** `Real`). За необхідності використання як параметрів масивів, записів та інших елементів даних, типи яких не задаються одним ім'ям, у розділі опису типів визначають користувальницькі типи даних (наприклад, **type** `t_array =` **array**`[1..10]` **of** `Real`) і вживають їх при описі формальних параметрів (див. приклад 8.3).

10. Деякі з параметрів можуть не мати типу (див. підрозд. 8.3).

11. Усередині функції перед її виконуваною частиною можуть бути оголошені змінні, які називаються ***локальними***. Ці змінні поза функцією не існують, оскільки під них виділяється пам'ять щоразу при виклику функції, а при виході з функції ці змінні знищуються.

12. Всі змінні, описані в модулі до оголошення функції, також відомі усередині функції. Ці змінні називаються ***глобальними***. У зв'язку з загальнодоступністю глобальних змінних у складних випадках можлива ситуація, коли одна функція може змінити значення змінної непомітно для іншої, результатом чого стає поява помилок, які досить важко виявляються. Щоб уникнути цього, рекомендується обмежувати використання глобальних змінних. Глобальні змінні необхідні в тому випадку, коли програмісту

потрібно зробити дані доступними для декількох функцій, а передача даних з однієї функції в іншу є проблематичною.

13. Результат роботи функції повертається в точку виклику тільки якщо у функції він буде записаний у внутрішню змінну `Result`, яку має будь-яка функція і яка завжди має тип, що збігається з типом функції, – інакше результат роботи функції не буде переданий у точку виклику. Змінну `Result` спеціально **описувати не потрібно**. Повернення результату забезпечується також за допомогою його присвоювання імені функції (запис результату роботи функції в її ім'я, успадкований Delphi від мови Turbo Pascal).

14. Результат роботи функції передається в точку виклику і через так звані **`var`**- та **`out`**-параметри (див. підрозд. 8.2), але це не найкращий стиль при програмуванні в середовищі Delphi.

15. Функція завершується службовим словом **`end`**, після якого ставиться крапка з комою.

16. Для звертання до функції необхідно згадати її ім'я в якому-небудь операторі (присвоювання, виведення і т. д.). При цьому замість формальних параметрів повинні бути підставлені фактичні параметри, якими можуть виступати змінні, константи, вирази. Типи формальних і фактичних параметрів мають бути сумісними за присвоюванням. Якщо в програмі активізована директива компілятору `$X` (стан `{$X+}` – розширений синтаксис), то до функції можна звернутися як до окремого оператора (наприклад, `factorial(10);`). Однак це можна робити тільки у випадках, коли значення, що повертається, не використовується. За умовчанням установлена директива `{$X+}`.

17. Формальні параметри є локальними змінними.

18. Імена локальних змінних не можуть збігатися з іменами формальних параметрів (це є наслідком з попереднього).

19. Імена формальних параметрів і локальних змінних можуть збігатися з різними іменами, оголошеними поза функцією. У цьому випадку в тілі функції активними будуть імена, оголошені в ній.

20. Функції можуть бути методами класів (див. підрозд. 11.1).

```
//Приклад 8.3
//Обчислити слід (суму елементів, розташованих на головній
//діагоналі) квадратної матриці розміру не більш ніж 20x20.
//Визначити функцію обчислення сліду квадратної матриці.
```

Скористаємося формою з прикладу 8.2, внісши такі зміни в значення властивостей:

- Мітка:
  Caption — Уведіть розмірність матриці (N<=20)

- Багаторядкове поле:
  Name — mmInpOut1
  Enabled — False
- Кнопка Button1:
  Caption — Введення

У секції **interface** модуля перед описом класу TForm1 визначимо тип для наступного оголошення двовимірного масиву:

```
type
  T20x20 = array[1..20, 1..20] of Real;
```

У визначення ж типу TForm1 додамо опис трьох полів:

```
N: Byte;
matrix: T20x20;
sum: Real;
```

Крім того, перед кодом процедури TForm1.Button1Click опишемо таку функцію:

```
function TraceOfMatr(a: T20x20; N: Byte): Real;    //Заголовок
var
  i: Byte;
  sum: Real;
begin
  sum := 0;
  for i := 1 to N do
    sum := sum + a[i, i];
  Result := sum;        //Результат заноситься в змінну Result
end;                                              //Кінець функції
```

Для розв'язання задачі скористаємося такими двома опрацьовувачами подій:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DecimalSeparator := '.';
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  i, j: Integer;
  s: string;
begin
  if Tag = 0 then begin
    N := StrToInt(Trim(edInput1.Text));
    if (N < 1) or (N > 20) then
      edInput1.SetFocus
    else begin
      mmInpOut1.Enabled := True;
```

```
      mmInpOut1.Lines.Add(
          'Уведіть через пробіл за рядками матрицю порядку '
          + IntToStr(N));
      edInput1.Visible := False;
      lbOutput1.Visible := False;
      Tag := 1;
      mmInpOut1.SetFocus;
    end;
  end
  else begin          //У рядку 0 mmInpOut1 виводилася підказка
    for i := 1 to N do begin
      s := mmInpOut1.Lines[i] + ' ';
      for j := 1 to N do begin
        matrix[i, j] := StrToInt(Copy(s, 1, Pos(' ', s) - 1));
        Delete(s, 1, Pos(' ', s));
      end;
    end;
    sum := TraceOfMatr(matrix, N);  //При звертанні до функції
           //замість формальних параметрів a та N підставлені
                              //фактичні параметри matrix та N
    mmInpOut1.Lines.Add('Слід матриці дорівнює ' +
                         FloatToStr(sum));
    Button1.Enabled := False;
  end;
end;
```

Оскільки в список формальних параметрів функції не можна включити опис

```
a: array[1..20, 1..20] of Real;
```

у програмі оголошений тип `T20x20`, що використаний як при описі поля `matrix` класу `TForm1`, так і при описі параметра `a` функції `TraceOfMatr` з метою забезпечення еквівалентності типів формальних і фактичних параметрів.

Незважаючи на те що це не стосується теми даного підрозділу, розглянемо особливості організації процесу введення даних у підпрограмі `TForm1.Button1Click`. За умовчанням властивість `Tag` має значення `0`, яке вказує на введення розмірності задачі. Якщо введене значення `N` з діапазону від `1` до `20`, то `Tag` набуває значення `1`, а це свідчитиме про те, що далі має вводитись вміст масиву `matrix`. Крім того, активізується багаторядковий редактор (`mmInpOut1.Enabled:=True`), і після виведення в мітці підказки про необхідність введення матриці він отримує фокус введення (`mmInpOut1.SetFocus`). Особливістю редагованого багаторядкового текстового поля є те, що його властивість `Lines` може розглядатися як рядковий масив з нумерацією від `0`. Оскільки перед

введенням масиву передбачене виведення підказки, що буде записана в рядок з номером `0`, то за умови суворого дотримання обумовленого порядку введення нумерація елементів властивості `Lines` багаторядкового редактора і рядків масиву `matrix` відповідатимуть одна одній (контроль правильності введення в цьому випадку не передбачений). Цикл за змінною `i` від `1` до `N` забезпечує перегляд записаних у поле `Lines` багаторядкового редактора рядків, а внутрішній цикл за змінною `j` служить для організації процесу одержання і запису значень окремих елементів в `i`-й рядок масиву `matrix`.

8.2. Процедури. Види параметрів

Як уже говорилося, правила гарного тону в програмуванні мовою Delphi вимагають, щоб функція повертала тільки одне значення. А як бути, якщо підпрограма в процесі своєї роботи або взагалі не повинна повертати яке-небудь значення (наприклад, підпрограма для виводу таблиці), або повинна змінювати значення відразу декількох змінних? У цьому випадку підпрограму оформляють у вигляді процедури.

Опис процедури є подібним до опису функції. По суті, існують тільки такі відмінності:

- у заголовку замість службового слова **`function`** вживається слово **`procedure`**;
- у заголовку не вказується тип імені процедури;
- у тілі процедури не виконується присвоювання значення імені процедури (в імені процедури немає типу) або змінній `Result` (ця змінна в процедурі взагалі відсутня).

За винятком цих моментів і методики виклику, все сказане про методику опису функції може бути перенесене і на процедури.

Важливим моментом при використанні процедур є застосування так званих **`var`**-параметрів (їх можна використовувати й у функціях, але це поганий стиль програмування мовою Delphi). Якщо в процесі роботи необхідно змінити значення декількох змінних, то відповідні імена можуть бути не згадані в списку формальних параметрів (тобто відповідати глобальним змінним). Однак можна в списку формальних параметрів перед деякими іменами задати службове слово **`var`**, яке свідчить про те, що дані імена визначають змінні, які змінюють свої значення в процесі роботи підпрограми. У списку формальних параметрів процедури можуть зустрічатися в будь-якому сполученні параметри-значення і параметри-змінні (**`var`**-параметри).

Приклад запису заголовка процедури:

```
procedure FindMaxMin(x, y: Integer;
                     var max, min: Integer);
```

У чому ж відмінність між двома видами формальних параметрів? Дотепер ми мали справу тільки з **передачею параметрів за значенням**. Цей метод полягає в тому, що при звертанні до підпрограми значення фактичного параметра записується у відповідний формальний параметр. Як тільки починається виконання процедури або функції, жодні зміни значення формального параметра не впливають на значення відповідного фактичного параметра. Такий спосіб діє за умовчанням, тобто у всіх випадках, коли не зазначено інший спосіб.

Використання **`var`**-параметра забезпечує **передачу параметра за адресою**. У цьому випадку в процедуру (або функцію) пересилається вже не значення фактичного параметра, а його місце розташування в пам'яті (адреса). Якщо формальний параметр має атрибут **`var`**, то будь-які зміни формального параметра будуть відбиватися в значенні фактичного параметра, оскільки вони тепер мовби займають одну і ту саму ділянку пам'яті. Насправді в цьому випадку при звертанні до формального параметра відбувається звертання до ділянки пам'яті, на яку він посилається; а оскільки цією ділянкою пам'яті є та, яку займає фактичний параметр, відбувається оперування (у тому числі й змінювання) фактичним параметром.

Важливою особливістю **`var`**-параметрів є те, що замість них як фактичні параметри можна підставляти тільки змінні (але не константи або вирази). Це, природно, потрібно враховувати при написанні програм.

Якщо виклик функції може з'являтися в будь-якому виразі, то цього не можна сказати про процедуру. Щоб звернутися до процедури, необхідно згадати її ім'я разом зі списком фактичних параметрів як **окремий оператор**. Як фактичні параметри, що підставляються замість формальних параметрів, оголошених зі службовим словом **`var`**, можна використовувати тільки змінні (але не константи й вирази).

Якщо необхідно вийти з процедури або функції, не досягши її кінця, то рекомендується застосовувати процедуру без параметрів `Exit`. Перед виконанням процедури `Exit` **`var`**-параметри і (при примусовому виході з функції) змінна `Result` або ім'я функції повинні дістати значення.

Підкреслимо: всі методи об'єктів і класів є процедурами і функціями (див. підрозд. 11.1).

Процедурами є й опрацьовувачі подій, про що вже говорилося вище. Правда, до опрацьовувачів подій звичайно не звертаються спеціально, оскільки вони викликаються автоматично.

```
//Приклад 8.4
//Скласти процедуру, що видаляє з першого рядка
//всі символи, які входять у другий рядок.
```

Скористаємося формою з прикладу 5.6, записавши у властивість Caption мітки текст Уведіть рядок, а властивість Caption кнопки введення текст Введення. Опишемо також у секції **implementation** модуля змінні s (рядок) і subs (підрядок) типу **string**. Тоді для розв'язання задачі може бути запропонована така процедура:

```
procedure DelSymbols(var st1: string; st2: string);
var
  i, p: Integer;
begin
  for i := 1 to Length(st2) do begin
    p := Pos(st2[i], st1);
    while p <> 0 do begin
      Delete(st1, p, 1);
      p := Pos(st2[i], st1);
    end;
  end;
end;                                          //Кінець процедури
```

Опрацьовувач події OnClick компонента Button1 може мати такий вигляд:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Tag = 0 then begin
    s := edInput1.Text;
    lbOutput1.Caption := 'Уведіть підрядок';
    edInput1.SetFocus;
    Tag := 1;
  end
  else begin
    subs := edInput1.Text;
    mmOutput1.Lines.Add('Початковий рядок:');
    mmOutput1.Lines.Add(s);
    mmOutput1.Lines.Add('Підрядок:');
    mmOutput1.Lines.Add(subs);
    mmOutput1.Lines.Add('Результуючий рядок:');
    DelSymbols(s, subs);
    mmOutput1.Lines.Add(s);
    Button1.Enabled := False;
  end;
end;
```

Процедура DelSymbols має два формальних параметри, перший з яких є **var**-параметром. При звертанні до неї замість формальних

параметрів st1 і st2 підставляються фактичні параметри s і subs. При цьому усередині процедури параметр st1 змінює своє значення, а оскільки він є **var**-параметром, то ці зміни приводять до зміни фактичного параметра s.

Схожими з **var**-параметрами у плані реалізації та застосування є так звані **out**-параметри. Якщо параметр підпрограми використовується тільки для передачі результатів роботи підпрограми в точку виклику і при цьому не використовується для передачі даних ззовні усередину підпрограми, то його можна описати з попереднім зарезервованим словом **out**:

```
procedure FindMaxMin(x, y: Integer;
                     out max, min: Integer);
```

При виклику підпрограми замість **out**-параметрів необхідно підставляти змінні. При цьому вони є тільки вихідними (результуючими).

Наголосимо: усе, що може виконати процедура, здійсненне і за допомогою функції (оскільки апарат **var**- і **out**-параметрів використовується й у функціях). Справедливим є і зворотне ствердження.

Як це зазначалося раніше, локальні змінні (у тому числі й формальні параметри) створюються в пам'яті в момент активізації підпрограми і знищуються при виході з неї. Якщо підпрограма викликає іншу підпрограму, то локальні змінні тієї підпрограми, що викликається, розміщаються в пам'яті зразу після локальних змінних першої підпрограми, у результаті чого утворюється ланцюжок груп локальних змінних. Знищення локальних змінних виконується з хвоста цього ланцюжка. Цей принцип відведення та звільнення пам'яті відповідає дисципліні обслуговування, відповідно до якої формується стек (див. підрозд. 7.4). Ділянка пам'яті, у яку записуються формальні параметри і локальні змінні підпрограм, також називається стеком. Стек не може займати більше 2147483647 байт і менше 1024 байт. За умовчанням розмір стека дорівнює 16384 байт. Якщо в програмі не використовуються підпрограми, можна зменшити розмір стека, а якщо вони використовуються активно, – збільшити.

Керування розміром стека здійснюється за допомогою директиви компілятору $M, що є директивою з параметрами. Її перший параметр задає мінімальний розмір стека, а другий – максимальний розмір (наприклад: {$M 10000, 2000000}). За умовчанням ця директива задана у вигляді {$M 16384, 1048576}.

У Delphi введено так звані константні параметри:

```
procedure proc(const param: Real);
```

Особливістю **const**-параметрів є те, що фактичні параметри, які підставляють замість них, передаються не за значенням, а за адресою (як і

у випадку **var**-параметрів). Однак оголошення формального параметра зі службовим словом **const** забороняє зміни відповідного параметра усередині підпрограми, і поява операторів, що модифікують такі параметри, зумовлює діагностику помилки на етапі компіляції. Константні параметри можуть використовуватися як при описі процедур, так і при описі функцій. Параметри-масиви, якщо їхній вміст не змінюється усередині підпрограми, рекомендується оголошувати як константні параметри, оскільки в цьому випадку копії фактичних параметрів при звертанні не створюються, що сприяє економії пам'яті, займаної в стеку локальними змінними.

Відзначимо, що якщо формальний параметр має який-небудь файловий тип, то він повинен бути описаний зі службовим словом **var** (можливе вживання одного зі слів **out** або **const**).

8.3. Нетипізовані параметри

Формальні параметри, описані з попереднім службовим словом **var** або **const**, можуть бути ***нетипізованими*** (***безтиповими***). Наприклад, з використанням таких параметрів описано багато стандартних процедур і функцій:

```
procedure BlockRead(var F: File; var Buf;
                    Count: Integer);
```

Фактичний параметр, що відповідає такому формальному параметру, повинен являти собою змінну будь-якого типу (але не константу або вираз). Для забезпечення роботи з безтиповим параметром його потрібно погодити в підпрограмі з фактичним параметром відповідно до типу останнього. Це забезпечується або приведенням типу формального параметра до типу відповідного фактичного параметра, або накладенням у пам'яті на безтиповий параметр деякої локальної змінної підпрограми.

Застосування безтипових параметрів робить підпрограми гнучкішими, але від програміста вимагає більшої відповідальності.

```
//Приклад 8.5
//Дано два цілочислових масиви однакового розміру.
//Знайти позицію першого елемента, яким відрізняються ці
//масиви. Описати функцію визначення номера першого
//елемента, яким відрізняються будь-які два одновимірних
//масиви одного типу з однаковою розмірністю. Якщо
//масиви еквівалентні, то функція повинна повертати
//значення -1.
```

Скористаємося формою з прикладу 3.1, змінивши значення деяких з властивостей:

- Мітка:
  Caption — Уведіть кількість елементів масивів
- Кнопка Button1:
  Caption — Введення

У секції ж **implementation** модуля опишемо такі змінні:

```
var
  a1, a2: array[1..100] of Integer;
  n: Integer;
```

Крім того, там же опишемо таку функцію:

```
function Numb(var a1, a2; n: Integer;
              SizeElem: Byte): Integer;
type
  TMaxByteArray = array[0..2000000000] of Byte;
var
  i: Integer;                        //Номер порівнюваних байтів
begin
  i := 0;
  while (i <= n * SizeElem) and          //Цикл, поки не будуть
       //зіставлені всі байти або не буде виявлена відмінність
   (TMaxByteArray(a1)[i] = TMaxByteArray(a2)[i]) do Inc(i)
  if i > n * SizeElem then
    Result := -1                          //Всі байти збігаються
  else Result := i div SizeElem;            //Обчислення номера
end;                                 //елементів, що відрізняються
```

Скористаємося наступним опрацьовувачем події OnClick компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i, p: Integer;
begin
  n := StrToInt(edInput1.Text);
  Randomize;
  for i := 1 to n do
    a1[i] := Random(10);
  for i := 1 to n do
    a2[i] := Random(10);
  p := Numb(a1, a2, n, SizeOf(a1[1]));
  if p = -1 then
    mmOutput1.Lines.Add('Масиви збігаються')
  else
    mmOutput1.Lines.Add('Відмінність із номера '
                        + IntToStr(p + Low(a1)));
  Button1.Enabled := False;
end;
```

У даному прикладі функція Numb містить два безтипові формальні параметри a1 і a2. Їхні типи приводяться до визначеного усередині функції типу TMaxByteArray, що дозволяє працювати з ними як з байтовими масивами, у яких порівнюються відповідні елементи (байти). Параметрами функції є два безтипові параметри (a1 і a2) для передачі порівнюваних масивів, кількість порівнюваних елементів у масивах (n) і розмір кожного елемента в байтах (SizeElem). Значення параметра SizeElem визначається в програмі за допомогою функції SizeOf. Оскільки у функції Numb передбачається розгляд масивів з нумерацією елементів від 0, значенням, яке вона повертає, і є номер знайденого елемента (або значення -1, якщо масиви збігаються). У процедурі TForm1.Button1Click результат роботи функції Numb коректується з урахуванням того, що у фактичних параметрів нумерація елементів інша і починається від номера, який повертає функція Low. Підкреслимо: функція Numb буде працювати не тільки у випадку, коли базовим типом обох масивів є тип Integer, але й при інших базових типах.

Наступний приклад дає уявлення про можливу методику роботи з безтиповими параметрами за допомогою накладення на них локальних змінних підпрограми.

```
//Приклад 8.6
//Процедура піднесення в цілий невід'ємний степінь
//даних типу Integer та Real. Результат має тип
//основи степеня. Для інших числових типів обчислення
//не виконуються. Контроль діапазону не проводиться.
```

Скористаємося формою з прикладу 8.5, записавши у властивість Caption мітки текст Уведіть ціле число і описавши в секції **implementation** модуля такі змінні:

```
var
  x, Res1: Integer;
  r, Res2: Real;
```

Крім того, у секції **implementation** опишемо процедуру піднесення в степінь:

```
procedure Involution(var x, Res; n: Integer; Size: Byte);
    { Смисл параметрів: x – основа степеня,
      Res – результат, n – показник степеня,
      Size – кількість байт, що відводиться під основу
      степеня (результат)    }
var
  xI: Integer absolute x;                 //Якщо x та Res мають
  ResultI: Integer absolute Res;                  //тип Integer
```

```
  xR: Real absolute x;                    //Якщо x та Res мають
  ResultR: Real absolute Res;                          //тип Real
  i: Integer;
begin
  case Size of              //Залежно від значення Size працюємо
    SizeOf(Integer): if xI = 0 then ResultI := 0    //з даними
                     else begin                   //типу Integer
                            ResultI := 1;
                            for i := 1 to n do
                              ResultI := ResultI * xI;
                     end;
    SizeOf(Real):    if xR = 0 then ResultR := 0    //або Real
                     else begin
                            ResultR := 1;
                           for i := 1 to n do
                              ResultR := ResultR * xR;
                    end;
   end;
end;
```

Наведений нижче опрацьовувач події OnClick компонента Button1 призначений для введення одного цілого й одного дійсного числа, а також піднесення їх у **третій** степінь:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Tag = 0 then begin
    x := StrToInt(edInput1.Text);
    edInput1.SetFocus;
    lbOutput1.Caption := 'Уведіть дійсне число';
    Tag := Tag + 1;
  end
  else begin
    DecimalSeparator := '.';
    r := StrToFloat(edInput1.Text);
    Involution(x, Res1, 3, SizeOf(Integer));       //Цілі дані
    Involution(r, Res2, 3, SizeOf(Real));     //Дані типу Real
    mmOutput1.Lines.Add(IntToStr(Res1));
    mmOutput1.Lines.Add(FloatToStr(Res2));
    Button1.Enabled := False;
  end;
end;
```

У процедурі Involution за допомогою директиви **absolute** проводиться накладення в пам'яті локальних змінних на її параметри. Усередині процедури виконується переключення режиму її роботи (обробка цілих або дійсних чисел) за допомогою переданого усередину неї розміру пам'яті, займаного фактичними параметрами. Природно, якщо буде

передане неправильне значення цього параметра, процедура буде виконуватися з помилкою.

8.4. Випереджальний опис

Ще один момент у використанні процедур і функцій – можливість випереджального опису, який полягає в тому, що використовувана підпрограма може бути описана тільки заданням свого заголовка, зразу за яким йде стандартна директива **forward**. Опис тексту такої підпрограми може бути розташований в будь-якому місці розділу описів без повторення списку формальних параметрів (хоча його можна і повторити). Випереджальний опис потрібен у випадку взаємного звернення процедур або функцій одна до одної:

```
procedure first(x, y: Real); forward;//Випереджальний опис

procedure second(x, y: Integer);//Початок процедури second
begin
          //Текст процедури second
  first(-1, 2.7);                  //Виклик процедури first
          //Продовження тексту процедури second
end;                               //Кінець процедури second

procedure  first;              //Опис тексту процедури first
begin
          //Текст процедури first
  second(-7, 12);                 //Виклик процедури second
          //Продовження тексту процедури first
end;                                //Кінець процедури first
```

Можна вважати, що методи класів, у тому числі опрацьовувачі подій, конструктори та деструктори (див. підрозд. 11.3) описуються з випереджальним описом, щоправда, без зазначення службового слова **forward**.

8.5. Рекурсивні підпрограми

Кожна з підпрограм може викликати інші підпрограми, у тому числі звертатися до самої себе. Підпрограма, що звертається до самої себе, називається ***рекурсивною***. Рекурсія може бути і неявною, коли підпрограма `first` викликає підпрограму `second`, а та у свою чергу викликає підпрограму `first`. У зв'язку з наявністю взаємних посилань підпрограм однієї на іншу виникає проблема, що зумовлена вимогою Delphi до описів:

спочатку описати, потім використовувати. При неявній рекурсії підпрограми описуються з використанням випереджального опису за допомогою описової директиви **forward** (див. підрозд. 8.4).

Обов'язковим елементом в описі будь-якого рекурсивного процесу є деяке твердження (оператор), що визначає умову завершення рекурсії; іноді вона називається опорною умовою. В ній може бути задане яке-небудь фіксоване значення, що обов'язково буде досягнуте в ході рекурсивного обчислення, дозволяючи тим самим організувати своєчасну зупинку процесу. Крім того, повинен бути другий елемент – спосіб вираження одного кроку розв'язання за допомогою іншого, простішого. Кількість рівнів вкладеності може бути досить великою.

```
//Приклад 8.7
//Рекурсивна функція обчислення факторіала:
//0! = 1, k! = k * (k - 1)!.
```

Скористаємося без змін формою з прикладу 8.1.3, оголосивши в секції **public** опису класу `TForm1` поля `N` та `factorial` типу `Integer` і вставивши в текст перед описом процедури `TForm1.Button1Click` опис функції `factorialR`:

```
function factorialR(k: Integer): Integer;
begin
  if k = 0 then Result := 1       //Якщо опорна умова виконана
  else Result := k * factorialR (k - 1);             //Якщо ні
end;
```

Сам опрацьовувач події OnClick для компонента Button1 дещо модифікуємо:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  N := StrToInt(edInput1.Text);
  if (N < 0) or (N > 12) then begin
    mmOutput1.Lines.Add('Повторіть введення');
    edInput1.SetFocus;
  end
  else begin
    mmOutput1.Lines.Add(IntToStr(N) + '! = ' +
                        IntToStr(factorialR(N)));
    Button1.Enabled := False;
  end;
end;
```

У наведеному тексті в процедурі `TForm1.Button1Click` функція `factorial` після першого її виклику багаторазово звертається сама до

себе зі значенням параметра k, яке зменшується від заданого при першому виклику початкового значення N до 0. При досягненні параметром значення 0 виконується опорна умова і здійснюється вихід з функції. При цьому відбувається повернення в точку виклику функції на попередньому рівні з продовженням обчислення значення виразу k*factorial(k-1) і наступним виходом з функції з поверненням на попередній рівень.

Рекурсивною може бути й процедура.

```
//Приклад 8.8
//Визначити множення цілих чисел через додавання:
//x*0=0, x*y=x+x(y-1) при y>0.
```

Скористаємося тією ж формою, що й у прикладі 8.7. У секції **public** опису класу TForm1 оголосимо поля N1, N2 та z типу Integer, а перед описом підпрограми TForm1.Button1Click вставимо опис процедури Multiplication:

```
procedure Multiplication(x, y: LongInt; var z: LongInt);
begin
  if y < 0 then begin
    y := -y;
    x := -x;
  end;
  if y = 0 then z := 0                              //Опорна умова
  else begin                                      //Рекурсивний виклик
    Multiplication(x, y - 1, z);
    z := x + z;
  end;
end;
```

Опрацьовувач події OnClick для компонента Button1 у цій задачі може бути наступним:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  s: string;
begin
  s := Trim(edInput1.Text);
  if Pos(' ', s) > 0 then begin
    N1 := StrToInt(Copy(s, 1, Pos(' ', s) - 1));
    Delete(s, 1, Pos(' ', s));
    N2 := StrToInt(Trim(s));
    Multiplication(N1, N2, z);
    mmOutput1.Lines.Add(IntToStr(N1) + '*' + IntToStr(N2) +
                        '=' + IntToStr(z));
    Button1.Enabled := False;
  end
```

```
  else begin
    mmOutput1.Lines.Add('Повторіть введення');
    edInput1.SetFocus;
  end;
end;
```

У процедурі `Multiplication` для правильної організації рекурсії доводиться нарощувати змінну `z` після виходу з попереднього рівня рекурсії.

Незважаючи на простоту та наочність коду рекурсивних підпрограм, вони можуть вимагати дуже великих обсягів пам'яті для свого виконання. Це пояснюється тим, що, як і у випадку звичайних підпрограм, локальні змінні створюються в програмному стеку на кожному рівні рекурсивного виклику, що може привести до ситуації, коли обсяг стека просто вичерпається, оскільки знищення локальних змінних здійснюється тільки при виході з підпрограми.

8.6. Відкриті масиви та рядки. Масиви констант

У Delphi введено так звані відкриті агрегатні параметри – масиви та рядки. Так, якщо як формальний параметр використовується одновимірний масив, то при його описі можна не задавати межі зміни індексу, маючи справу з так званим ***відкритим масивом***. Сам формальний параметр при цьому може бути описаний і зі службовим словом **`var`**. Наприклад, припустимим є такий заголовок функції:

```
function Test(var a: array of Integer): Boolean;
```

Як відповідний фактичний параметр підпрограми в цьому випадку може бути одновимірний масив будь-якого розміру (однак типи елементів повинні збігатися). Нижня межа відкритого масиву завжди дорівнює `0`. Тому реальні межі переданого масиву визначають усередині підпрограми за допомогою функцій `Low` і `High`: функція `Low` дає в цьому випадку завжди значення `0`, а `High` – відмінне від `0` верхнє значення індексу при звертанні до масиву – фактичного параметра (кількість елементів масиву мінус `1`).

Розглянемо найпростіший приклад, що демонструє використання відкритих масивів. У прикладі діапазон зміни індексів оброблюваного масиву заданий константами (від `-1` до `7`), на які здійснюється орієнтація і при введенні даних.

Мета цього прикладу – показати можливу методику використання функцій `Low` і `High` для визначення діапазону зміни індексу масиву усередині підпрограми.

```
//Приклад 8.9
//Чи правда, що сума парних елементів масиву
//цілих чисел більша за суму його непарних елементів.
```

Скористаємося формою з прикладу 2.2, надавши властивості Caption мітки значення Введіть елемент масиву, а цій же властивості кнопки введення – значення Введення. Крім того, надамо властивості Tag форми значення -1 (нижня межа виміру масиву). У коді модуля в секції **public** опису форми оголосимо поле a типу **array**[-1..7] **of** Integer. Крім того, у секції **implementation** перед описом процедури TForm1.Button1Click опишемо функцію Test:

```
function Test(a: array of Integer): Boolean;
var
  i, SumEv, SumUnev: Integer;          //SumEv – сума парних,
begin                          //SumUnev – сума непарних елементів
  SumEv := 0;
  SumUnev := 0;                    //Нижче цикл від нижньої межі
  for i := Low(a) to High(a) do begin     //масиву до верхньої
    if Odd(a[i]) then SumUnev := SumUnev + a[i]
    else SumEv := SumEv + a[i];
  end;
  Result := SumEv > SumUnev;
end;
```

Опрацьовувач події OnClick для компонента Button1 у цій задачі такий:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin                          //Нижче - умова припинення введення
  if Tag < 8 then
    a[Tag] := StrToInt(edInput1.Text);             //Зчитування
  Tag := Tag + 1;      //Змінюємо номер елемента, що вводиться
  edInput1.SetFocus;
  if Tag = 8 then begin           //Якщо введення закінчене, то
    mmOutput1.Lines.Add('Початковий масив');        //виводимо
    for i := -1 to 7 do                        //початковий масив
      mmOutPut1.Lines.Add('a[' + IntToStr(i) + ']='
         + IntToStr(a[i]));       //Нижче виведення результату
    if Test(a) then mmOutput1.Lines.Add('Правда')
    else mmOutput1.Lines.Add('Неправда');
    edInput1.Hide; lbOutput1.Hide;     //Перемикаємо видимість
    Button1.Enabled := False;                   //і активність
  end;
end;
```

У функції Test нижня межа масиву a, переданого в неї для обробки, визначається функцією Low, верхня – функцією High. Зазначимо також,

що при введенні вмісту масиву як індекс елемента, що вводиться, використовується поле `Tag` форми з початковим значенням `-1`, заданим на етапі проектування програми.

Відкриті масиви можуть бути ***масивами констант***, для чого їх оголошують, указавши замість імені типу службове слово **`const`**, що означає наступне: замість такого масиву як фактичний параметр підставляється масив, що містить дані не обов'язково одного типу. Звичайно як фактичний параметр у цьому випадку підставляють так званий *конструктор масиву*.

Конструктор масиву являє собою укладений у квадратні дужки список значень, розділених комами. Наприклад, до функції з прикладу 8.7 можна звернутися в такий спосіб:

```
if Test([1, 13, 22, 17]) then
  mmOutput1.Lines.Add('Правда');
```

Природно, відкритий масив констант не може бути оголошений зі службовим словом **`var`** або **`out`** (хоча компіляція підпрограми в цьому випадку проходить нормально). Якщо при звертанні до підпрограми замість відкритого масиву, описаного зі службовим словом **`var`** або **`out`**, підставити конструктор масиву, то компілятор діагностує помилку. Так, якщо заголовок функції `Test` має вигляд заголовка, наведеного на початку даного підрозділу, то компілятор видасть діагностичне повідомлення про помилку при звертанні до функції `Test` у наведеному вище операторі **`if`**.

Особливістю відкритого масиву констант є те, що він створюється в пам'яті як масив варіантних записів (див. п. 5.4.2) типу `TVarRec`, що визначений у модулі `System` у такий спосіб:

```
type
  TVarRec = record
    case VType: Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
```

```
      vtVariant:     (VVariant: PVariant);
      vtInterface:   (VInterface: Pointer);
      vtWideString: (VWideString: Pointer);
      vtInt64:       (VInt64: PInt64);
end;
```

При цьому деякі з полів запису `TVarRec` є звичайними полями, а деякі – вказівниками (імена їх типів починаються з літери `P`). Треба враховувати, що при звертанні до полів-вказівників потрібно використовувати операцію розіймення вказівника `^`. Перед обробкою елемента відкритого масиву необхідна перевірка типу елемента, для чого треба звернутися до поля `VType`, у якому зберігається цілочислове значення від `0` до `16`, що кодує тип поля (відповідно до наведеного вище порядку перерахування полів у визначенні типу `TVarRec`). При цьому рекомендується використовувати іменовані константи, перелічені вище у визначенні типу `VType` (їх імена починаються з префікса `vt`).

```
//Приклад 8.10
//Обробити відкритий масив констант, виділивши елементи,
//що являють собою значення типу Integer та дійсні числа.
```

Скористаємося формою з прикладу 2.2, видаливши з неї однорядковий редактор введення та мітку виведення й записавши у властивість Caption компонента Button1 текст Виконати. Опишемо також перед процедурою `TForm1.Button1Click` таку процедуру:

```
procedure Division(a: array of const; var si, sd: string);
var
  i, vt: Integer;
begin
  si := '';
  sd := '';
  for i := Low(a) to High(a) do begin
    if a[i].VType = vtInteger          //Елемент – ціле число?
    then si := si + IntToStr(a[i].VInteger) + ' '
    else
      if a[i].VType = vtExtended     //Елемент – дійсне число?
        then sd := sd + FloatToStr(a[i].VExtended^) + ' ';
  end;
end;
```

Сам опрацьовувач `TForm1.Button1Click` може мати, наприклад, такий програмний код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  si, sd: string;            //Для накопичення цілих і дійсних
```

```
begin
  Division([1, 1.2, 23, -1.2, -111111, True, 2.5, 'Ok'],
           si, sd);
  mmOutput1.Lines.Add('Цілі числа');
  mmOutput1.Lines.Add(si);
  mmOutput1.Lines.Add('Дійсні числа');
  mmOutput1.Lines.Add(sd);
end;
```

У процедуру Division відкритий масив передається у вигляді конструктора масиву, що містить значення різних типів (дійсні, цілі, булівське та рядкове). Оскільки перший параметр процедури Division – відкритий масив констант, у ній можливий аналіз типу значень переданих як елементи масиву. У циклі, що забезпечує перебір елементів масиву, насамперед, перевіряється тип елемента. Для цього аналізується значення, записане в полі VType елемента (нагадаємо, що в кожного елемента відкритого масиву є таке поле), і на подальшу обробку надходять тільки ті елементи, у яких у полі VType записане одне зі значень – vtInteger або vtExtended. При цьому доступ до самих переданих у процедуру Division значень здійснюється через варіантні поля VInteger (якщо значення має тип Integer) і VExtended (у випадку дійсного значення). Відзначимо також, що поле VExtended є вказівником на тип Extended, у зв'язку з чим для одержання самого значення вказівник повинен бути розіменований (здійснюється звертання до VExtended^).

Відкриті рядки в Delphi описуються типом OpenString і також повинні супроводжуватися службовим словом **var**. У такому разі максимальна довжина формального параметра завжди буде збігатися з максимальною довжиною будь-якого короткого рядка, переданого в підпрограму як фактичний параметр. Замість типу OpenString, **var**-параметр може бути описаний з типом ShortString. Тип OpenString є застарілим й у сучасному програмуванні не використовується.

8.7. Процедурний тип

У Delphi можливе використання змінних, що набувають значень процедур і функцій. Тип таких змінних задається службовим словом **procedure** або **function** зі списком параметрів (якщо він передбачається) і типом значення, яке повертається, для функції. Ім'я процедури або функції в описі типу, природно, не ставиться:

```
type
  TProc1 = procedure;
```

```
  TProc2 = procedure(a: Integer; var b: Real);
  TFun = function(x, y: Real): Real;
var
  Pr1: TProc1;
  Pr2: TProc2;
  Fun: TFun;
```

Тут `Pr1` – процедура без параметрів, `Pr2` – процедура з параметрами, `Fun` – функція.

Якщо тепер у користувальницькій програмі описати процедуру або функцію зі структурою заголовка, аналогічною опису типу якої-небудь зі змінних `Pr1`, `Pr2`, `Fun`, то цій змінній можна «присвоїти» відповідну процедуру або функцію (зв'язати змінну з процедурою або функцією). Після цього можна працювати з такою змінною точно так само, якби вона фігурувала як ім'я відповідної процедури або функції. Так, якщо в програмі описана дійсна змінна `h`, то до функції

```
function hypoten(a, b: Real): Real;
begin
  hypoten := Sqrt(a * a + b * b);
end;
```

можна звернутися двома способами:

```
h := hypoten(1.2, 10));
```

або

```
Fun := hypoten; h := Fun(1.2, 10));
```

У загальному випадку процедурний тип визначається в такий спосіб:

```
type
  ім'я_типу = procedure(специфікація_параметрів);
```

або

```
type
  ім'я_типу = function(специфікація_параметрів): тип;
```

Процедурний тип дозволяє за допомогою однієї змінної виконувати різні процедури (функції), якщо ці процедури (функції) мають однакову структуру заголовка.

Вимоги, які необхідно виконати при використанні процедурного типу:

1. Процедурній змінній не можна присвоїти стандартну процедуру або функцію.

2. Процедури і функції, що присвоюються, не можуть бути вкладеними в інші підпрограми.

Змінні процедурного типу можуть оголошуватися як типізовані константи. Для цього в оголошенні після знака рівності потрібно зазначити ім'я процедури або функції (без задавання параметрів), яке є початковим значенням процедурної змінної, що оголошується. Наприклад, після того як будуть описані наведені вище тип TFun і функція hypoten, можна включити в програму такий опис:

```
const
  fn: Tfun = hypoten;
```

```
//Приклад 8.11
//Написати процедури, які видаляють із заданого рядка
//всі символи, що входять в інший рядок (перша
//процедура), або всі підрядки, що збігаються з іншим
//рядком (друга процедура).
```

Скористаємося формою з прикладу 2.2, внісши такі зміни у властивості деяких з компонентів

- Мітка:
  Caption — Уведіть оброблюваний рядок
- Однорядкове редаговане текстове поле:
  Caption — очистити
- Кнопка Button1:
  Caption — Введення

Оголосимо також у секції **interface** змінні s1 і s2 типу **string**, а в секції **implementation** опишемо дві процедури:

```
 //Процедура 1 не може бути вкладеною в іншу підпрограму
procedure del_symb(var st1: string; st2: string);
var
  i, j: Integer;
begin
  for i := 1 to Length(st2) do
  begin
    j := 1;
    while j <= Length(st1) do
      if st1[j] = st2[i]
        then Delete(st1, j, 1)
        else j := j + 1;
  end;
end;
//Процедура 2 не може бути вкладеною в іншу підпрограму
procedure del_substr(var st1: string; st2: string);
begin
  while Pos(st2, st1) <> 0 do
```

```
    Delete(st1, Pos(st2, st1), Length(st2));
end;
```

Опрацьовувач події OnClick компонента Button1 може мати, наприклад, такий вигляд:

```
procedure TForm1.Button1Click(Sender: TObject);
type                              //Оголошення процедурного типу
  TProc = procedure(var st1: string; st2: string);
var
  v: TProc;              //Оголошення змінної процедурного типу
  c: Char;
begin
  case Tag of
   0: begin
        s1 := edInput1.Text;
        lbOutput1.Caption := 'Уведіть рядок для порівняння';
      end;  {0}
   1: begin
        s2 := edInput1.Text;
        lbOutput1.Caption := 'Видаляти символи/рядок? y/n';
      end;  {1}
   2: begin
        c := edInput1.Text[1];
        mmOutput1.Lines.Add('    Оброблюваний рядок');
        mmOutput1.Lines.Add(s1);
        mmOutput1.Lines.Add('    Рядок для порівняння');
        mmOutput1.Lines.Add(s2);
        if c in ['y', 'Y'] then begin
          mmOutput1.Lines.Add('    Видалення символів');
                    //Визначення значення процедурної змінної
          v := del_symb
        end
        else begin
          mmOutput1.Lines.Add('    Видалення підрядка');
          v := del_substr;
        end;
                  //Нижче процедурна змінна v використовується
                                    //для виклику підпрограми
        v(s1, s2);                        //Виклик підпрограми
        mmOutput1.Lines.Add('Результат обробки');
        mmOutput1.Lines.Add(s1);
      end;  {2}
  end;  {case}
  Tag := Tag + 1;
  edInput1.SetFocus;
end;
```

У даному прикладі показана методика оголошення процедурного типу (тип TProc) й опису процедурних змінних (змінна v). Процедурним змінним можна присвоювати значення, що є ім'ям раніше оголошеної підпрограми (оператори v:=del_symb і v:=del_substr), після чого ця змінна може використовуватися для виклику конкретної підпрограми: оператор v(s1, s2) викличе або процедуру del_symb, або процедуру del_substr залежно від того, яке значення було присвоєне змінній v. У зв'язку з тим, що процедурним змінним не можна присвоювати імена процедур, описаних усередині інших підпрограм (у тому числі усередині підпрограми, у якій здійснюється присвоювання), процедури del_symb і del_substr описані поза і раніше процедури TForm1.Button1Click.

Зазначимо, що даний приклад є чисто ілюстративним, оскільки в процедурі TForm1.Button1Click значно простіше було організувати безпосередній виклик процедур del_symb і del_substr без застосування допоміжної змінної процедурного типу.

Процедурний тип дозволяє використовувати процедури та функції як параметри інших процедур і функцій. При цьому як фактичні параметри придатні і процедурні змінні, і безпосередньо імена підпрограм. У цьому випадку обов'язково повинен бути оголошений процедурний тип у розділі **type**, котрий і буде застосовуватися при описі відповідного формального параметра.

```
//Приклад 8.12
//Реалізувати процедуру відшукання кореня рівняння f(x)=0
//на відрізку [a1, a2] у припущенні, що на цьому відрізку
//існує єдиний корінь. Використати метод
//поділу відрізка навпіл. Корінь рівняння f(x)=0
//відшукувати з точністю r>0: x0 – корінь, якщо довжина
//розглянутого відрізка не перевищує r.
//Обчислити корені таких рівнянь:
// f(x)=Sin(x);
// f(x)=(Sin x)(Sin x)-3(Cos x)(Cos x)/4.
```

Розмістимо на формі з прикладу 2.2 поруч із міткою виведення ще дві мітки, під якими (поруч із однорядковим редактором введення) вмістимо ще два редактори введення, і присвоїмо властивостям компонентів такі значення:

- Перша мітка:
  Caption — Ліва межа
- Друга мітка:
  Caption — Ліва межа
  Name — lbOutput2

- Третя мітка:
  Caption — Точність
  Name — lbOutput2
- Другий редактор введення:
  Name — edInput2
- Третій редактор введення:
  Name — edInput3
- Кнопка Button1:
  Caption — Обчислити

У секцію **implementation** модуля внесемо такі описи:

```
type
  f_type = function (x: Real): Real;
procedure root(var x: Real; a1, a2: Real;
               eps: Real; f: f_type);
                    //Формальний параметр f має тип "функція"
begin
  while Abs(a2 - a1) > eps do begin
    x := (a1 + a2) / 2;
    if f(a1) * f(x) <= 0 then a2 := x
    else a1 := x;
  end;
end;
function f1(x: Real): Real;         //Процедурна змінна не може
            //мати стандартне ім'я (тут Sin). Здійснено заміну
begin
  Result := Sin(x);
end;
function f2(x: Real): Real;
begin
  Result := (Sin(x)) * (Sin(x)) - 1.75 * (Cos(x)) * (Cos(x));
end;
```

Крім того, створимо для форми опрацьовувач події OnCreate:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DecimalSeparator := '.';
  edInput1.TabOrder := 0;
end;
```

Створимо також для компонента Button1 опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a1, a2, r, x: Real;       //Межі інтервалу, точність, корінь
  ff: f_type;
```

```
begin
    a1 := StrToFloat(edInput1.Text);
    a2 := StrToFloat(edInput2.Text);
    r := StrToFloat(edInput3.Text);
  //Нижче два варіанти фактичних параметрів процедурного типу:
    //при i = 1 - ім'я функції,
    //при i = 2 - змінна процедурного типу
    if Tag = 0 then
      root(x, a1, a2, r, f1)
    else begin
      ff := f2;
      root(x, a1, a2, r, ff);
      Button1.Enabled := False;
    end;
    mmOutput1.Lines.Add('x = ' + FloatToStr(x));
    Tag := Tag + 1;
    edInput1.SetFocus;
end;
```

У даному прикладі демонструється можливість використання процедурного типу для опису параметрів підпрограм. Так, у процедурі `root` формальний параметр `f` має тип «функція». Для опису цього параметра визначено користувальницький тип `f_type`, який застосовано при описі змінної `ff` в опрацьовувачі `TForm1.Button1Click`. Необхідність визначення користувальницького типу пояснюється непридатністю типу **`function`** `(x:Real):Real` для безпосереднього опису формального параметра `f` процедури `root`. В опрацьовувачі `TForm1.Button1Click` ілюструється можливість підстановки як фактичного параметру безпосередньо імені функції (при `Tag=0`) і змінної `ff` типу «функція» (при `Tag=1`), яка попередньо дістала значення `f2`, що забезпечує доступ до функції `f2`. Властивість `Tag`, яка за умовчанням дорівнює `0`, використана для організації циклу з метою перебору двох варіантів функцій. Даний приклад є чисто ілюстративним, оскільки задача може бути розв'язана значно раціональніше.

8.8. Функції, що повертають вказівник

Як це зазначалося в підрозділі 8.1, функції можуть повертати вказівник. Програма, що наводиться нижче, містить функцію, яка, повертаючи значення у вигляді вказівника, фактично забезпечує повернення одновимірного масиву, що є рядком двовимірного масиву.

```
//Приклад 8.13
//Дано двовимірний масив цілих чисел, розміру
```

```
//не більш 10x10. Переписати в одновимірний масив рядок,
//який містить найбільшу кількість непарних елементів.
//Оформити функцію, що повертає вказівник на такий
//рядок двовимірного масиву або nil, якщо у двовимірному
//масиві відсутні непарні елементи.
```

Скористаємося формою з прикладу 2.2, записавши у властивість Caption мітки текст Уведіть кількість рядків, а у властивість Caption однорядкового редактора – текст Введення. У секції **implementation** модуля опишемо наступні типи, змінні та функцію:

```
type
  T10 = array[1..10] of Integer;//Тип для одновимірного масиву
  T10x10 = array[1..10] of T10; //Тип для двовимірного масиву
  TPt10 = ^T10;                 //Посилальний тип – посилання
                                        //на одновимірний масив
var
  m, n: Integer;
  q: T10x10;                              //Двовимірний масив
  d: T10;                                //Одновимірний масив
function f(q: T10x10; m, n: Integer): TPt10;
var
  i, j: Integer;
  max, num: Integer;
begin
  max := 0;
  for i := 1 to m do begin
    num := 0;
    for j := 1 to n do
      if Odd(q[i, j]) then Inc(num);
    if num > max then begin
      max := num;
      Result := @q[i, 1];   //Запам'ятовуємо адресу і-го рядка
    end;
  end;
  if max = 0               //За відсутності непарних елементів
    then Result := nil;              //повертаємо значення nil
end;
```

Для компонента Button1 напишемо такий опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
  i, j: Integer;
begin
  if Tag = 0 then
  begin
    Randomize;
```

```
    m := StrToInt(edInput1.Text);
    lbOutput1.Caption := 'Уведіть кількість стовпців';
    mmOutput1.Lines.Add('m = ' + IntToStr(m));
    edInput1.SetFocus;
  end
  else
  begin
    n := StrToInt(edInput1.Text);
    mmOutput1.Lines.Add('n = ' + IntToStr(n));
    mmOutput1.Lines.Add('Оброблюваний масив');
    for i := 1 to m do
    begin
      s := '';
      for j := 1 to n do
      begin
        q[i, j] := Random(200) - 100;
        s := s + IntToStr(q[i, j]) + #9;          //#9 – символ
      end;                                          //табуляції
      mmOutput1.Lines.Add(s);
    end;
            //Записуємо у масив d вміст одновимірного масиву,
    d := f(q, m, n)^;        //адресу якого повертає функція f

    mmOutput1.Lines.Add('Результуючий рядок:');
    s := '';
    for j := 1 to n do
      s := s + IntToStr(d[j]) + #9;
    mmOutput1.Lines.Add(s);
    edInput1.Hide;
    lbOutput1.Hide;
    Button1.Enabled := False;
  end;
  Tag := Tag + 1;
end;
```

У даному прикладі використана особливість розподілу пам'яті, яка полягає в тому, що двовимірні масиви розміщаються в пам'яті безперервною ділянкою за рядками, починаючи з першого рядка. Конструкція `@q[i, 1]` позначає адресу першого елемента `i`-го рядка двовимірного масиву. Оскільки ім'я функції має посилальний тип `TPt10` з базовим типом `T10`, еквівалентним типу **`array`**`[1..10]` **`of`** `Integer`, то після виконання оператора присвоювання `Result := @q[i, 1]` функція поверне вказівник на `i`-й рядок масиву `q`. Оператор `d := f(q, m, n)^` забезпечує переписування у масив `d` одновимірного масиву, на який указує повернутий функцією `f` вказівник. Він також демонструє методику доступу до значення, що зберігається за адресою, яку

повертає функція у вигляді вказівника (символ ^ розташовується після дужок з переліком фактичних параметрів функції).

Зазначимо також, що якщо в масиві немає непарних елементів, функція поверне порожній вказівник **nil**, результатом чого буде помилковість оператора `d := f(q, m, n)^`. У зв'язку з цим правильнішим буде вдатись до такого опрацьовувача події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s, ss: string;
  i, j: Integer;
  Pt: TPt10;
begin
  if Tag = 0 then
  begin
    Randomize;
    m := StrToInt(edInput1.Text);
    lbOutput1.Caption := 'Уведіть кількість стовпців';
    mmOutput1.Lines.Add('m = ' + IntToStr(m));
    edInput1.SetFocus;
  end
  else begin
    n := StrToInt(edInput1.Text);
    mmOutput1.Lines.Add('n = ' + IntToStr(n));
    mmOutput1.Lines.Add('Оброблюваний масив');
   for i := 1 to m do begin
      s := '';
      for j := 1 to n do begin
        q[i, j] := Random(200) - 100;
        s := s + IntToStr(q[i, j]) + #9;
      end;                              //#9 - символ табуляції
      mmOutput1.Lines.Add(s);
    end;
    Pt := f(q, m, n);          //У Pt переписуємо посилання на
         //одновимірний масив, адресу якого повертає функція f
    if Pt = nil then
      mmOutput1.Lines.Add('Немає непарних елементів')
    else begin
      d := Pt^;
      mmOutput1.Lines.Add('Результуючий рядок:');
      s := '';
      for j := 1 to n do
        s := s + IntToStr(d[j]) + #9;
      mmOutput1.Lines.Add(s);
    end;
    edInput1.Hide;
    lbOutput1.Hide;
```

```
    Button1.Enabled := False;
  end;
  Tag := Tag + 1;
end;
```

У цьому випадку у вказівник `Pt` записується або посилання на одновимірний масив, що є рядком двовимірного масиву, або значення **`nil`**, у результаті чого в масив `d` переписуються дані тільки при поверненні функцією `f` значення, відмінного від **`nil`**.

Можливість повернення функцією значення у вигляді вказівника дозволяє організувати в програмі меню, яке базується на поверненні функцією вказівників на процедурний (функціональний) тип, що демонструє наступний приклад.

```
//Приклад 8.14
//Здійснити табулювання однієї з двох функцій з трьома
//коефіцієнтами (a, b, c) на заданому інтервалі [d1, d2]
//для n рівновіддалених точок. Побудувати підпрограму
//табулювання, що залежить від функції із чотирма параметрами
//(три коефіцієнти і значення аргументу). Для установлення
//функції, що табулюватиметься, використовувати функцію
//menu, яка повертає вказівник на функцію, яка табулюється.
```

Скористаємося формою з попереднього прикладу, записавши у властивість Caption мітки текст Уведіть ліву межу інтервалу. У секції **`implementation`** модуля опишемо наступні змінні, типи, функції та процедуру:

```
var
  a, b, c, d1, d2: Real;
  n: Integer;

type
  TFile = file of Real;
  TFunc = function(a, b, c, x: Real): Real;
  TPtTFunc = ^TFunc;            //Визначення типу TPtTFunc для
               // оголошення вказівників на функції з чотирма
                        //параметрами і результатом типу Real

function SinABC(a, b, c, x: Real): Real;
begin
  Result := a * Sin(b * x + c);
end;

function ParabolaABC(a, b, c, x: Real): Real;
begin
  Result := a * (x + b) * (x + b) + c;
end;
```

```
procedure tabul(d1, d2: Real; n: Integer; a, b, c: Real;
                var fr: TFile; f: TFunc);
var
  dx, v: Real;
  i: Integer;
begin
  dx := (d2 - d1) / (n - 1);
  for i := 1 to n do
  begin
    v := f(a, b, c, d1 + (i - 1) * dx);
    Write(fr, v);
  end;
end;
```

Опрацьовувач події OnClick для компонента Button1 має такий вигляд:

```
procedure TForm1.Button1Click(Sender: TObject);
                //Визначення локальної функції без параметрів,
                                  //що повертає вказівник на функцію
function menu: TPtTFunc;

                        //Визначаємо масив вказівників на функції
          //з його ініціалізацією адресами завантаження функцій
const
  f_items: array[1..2] of TPtTFunc = (@SinABC, @ParabolaABC);

var
  key: Integer;
begin
  key := StrToInt(edInput1.Text);
  case key of
   1,2: Result := f_items[key];//Повернення вказівника на одну
 //з функцій або значення nil при неправильно введеному номері
   else Result := nil;
  end;
end;                                          //Кінець функції menu

var             //Локальні змінні процедури TForm1.Button1Click
  FileOfReal: TFile;
  item: TPtTFunc;       //Змінна item визначається за допомогою
                  //раніше введеного типу TPtTFunc як вказівник
    //на функцію з чотирма параметрами і результатом типу Real
begin
  case Tag of
    0: begin
        DecimalSeparator := '.';
        lbOutput1.Caption := 'Права межа інтервалу';
        edInput1.SetFocus;
       end;
```

```
    1: begin
        d2 := StrToFloat(edInput1.Text);
        lbOutput1.Caption := 'Уведіть кількість вузлів >1';
        edInput1.SetFocus;
       end;
    2: begin
        n := StrToInt(edInput1.Text);
        lbOutput1.Caption := 'Вибери функцію: ' +
                     '1 (синусоїда) або 2 (парабола)';
        edInput1.SetFocus;
       end;
    3: begin
                      //Заносимо в item адресу функції або nil
        item := menu;
        if item = nil
        then mmOutput1.Lines.Add('Неправильний номер')
        else begin
          AssignFile(FileOfReal, 'result.dat');
          Rewrite(FileOfReal);       //Нижче використання item
          tabul(d1, d2, n, a, b, c,
                FileOfReal, TFunc(item));
          mmOutput1.Lines.Add('Табулювання завершене');
          CloseFile(FileOfReal);
        end;
        edInput1.Hide;
        lbOutput1.Hide;
        Button1.Enabled := False;
       end;
  end;
  Tag := Tag + 1;
end;
```

Визначений посилальний тип `TPtTFunc=^TFunc` з базовим функціональним типом `TFunc=`**`function`**`(a,b,c,x:Real):Real` використовується як тип значення, що повертає функцією `menu`, яка є локальною функцією в опрацьовувачі `TForm1.Button1Click`. Функція `menu` повертає вказівник на одну з функцій `SinABC` й `ParabolaABC`.

Якщо функція, що табулюється, вибирається неправильно, функція `menu` повертає порожній вказівник **`nil`**. Вказівник, що повертається функцією `menu`, дозволяє забезпечити виклик функції, на яку він вказує. Повернуте функцією `menu` значення записується в змінну `item`. Якщо в цій змінній не записано значення **`nil`**, то відповідний вказівник, перетворений до типу `TFunc`, підставляється як фактичний параметр у підпрограму `tabul`, забезпечуючи тим самим передачу у функцію `tabul` адреси, за якою розташовується код функції, яка табулюється.

Вибір функції, що повертається, виконується за допомогою оператора **case**, який забезпечує звертання до одного з елементів масиву `f_items`, у якому зберігаються вказівники на функції (адреси їхнього завантаження). Функція `tabul` не орієнтована на табулювання конкретної функції, – вона орієнтована тільки на перелік і тип параметрів, а також тип значення, що повертається. Результати табулювання у вигляді послідовності значень типу `Real` записуються у файл `result.dat`. Правильність введення даних і коректність звертання до файлу не контролюються.

Даний приклад, як і решта розглянутих у цьому розділі, є чисто ілюстративним. Для побудови меню в Delphi є спеціальні засоби.

8.9. Підпрограми з параметрами за умовчанням

Як і в C++, у Delphi можливе використання так званих параметрів процедур і функцій за умовчанням.

Параметри за умовчанням – це параметри, які можуть опускатися при виклику підпрограми.

Щоб оголосити параметр за умовчанням необхідно при описі підпрограми в її заголовку слідом за типом формального параметра після знака рівності вказати значення цього параметра за умовчанням.

При виклику підпрограми параметри за умовчанням **можуть бути** опущені. У цьому випадку замість опущених фактичних параметрів підставляються їхні значення за умовчанням, зазначені при описі підпрограми. Таким чином, список фактичних параметрів у цьому випадку буде коротшим від списку формальних параметрів.

Якщо при звертанні до підпрограми який-небудь з параметрів за умовчанням указаний явно, то решта параметрів за умовчанням, що стоять перед ним, також повинні бути указані явно.

Слід зазначити, що параметри за умовчанням можуть бути тільки останніми в списку параметрів підпрограми. Якщо для якого-небудь параметра наведене його значення за умовчанням, то значення за умовчанням повинні мати й всі наступні параметри.

Нехай, наприклад, деяка змінна `num` має значення `21`, а підпрограма має заголовок

```
procedure Out(f:string='IF';gr:Byte=57;ind:Char='v');
```

Тоді допустимі такі звертання:

```
Out;                    //Те ж, що й Out('IF', 57, 'v');
Out('FT', 21);          //Те ж, що й Out('FT', 21, 'v');
Out('IF', num, 'a');
Out('IF', 56, 'a');
```

У то же час такі звертання помилкові:

```
Out('FT', ,'a');          //Потрібно Out('FT', 57, 'a');
Out('IF','a');//Очевидно, потрібно Out('IF', 57, 'a');
```

За умовчанням можуть бути й параметри при визначенні процедурного типу. У цьому випадку виникає невизначеність, коли підпрограма, що ініціалізує змінну процедурного типу, також має параметри за умовчанням, причому значення таких її параметрів відрізняється від значень, наведених при визначенні процедурного типу. Ця невизначеність вирішується на користь визначення процедурного типу, тобто при виклику підпрограми як значення за умовчанням, береться те, яке вказане у визначенні процедурного типу.

8.10. Перевантажувані підпрограми

У Delphi існує так званий механізм перевантаження підпрограм, фактично запозичений з мови C++.

Перевантаження підпрограм означає те, що в одній програмній одиниці одночасно можуть існувати кілька підпрограм з одним і тим самим ім'ям. При цьому компілятор за контекстом автоматично визначає, яка з них повинна бути підключена.

Мета перевантаження підпрограм – забезпечити різне виконання підпрограм з одним і тим самим ім'ям при звертанні до неї з різними за типом та кількістю параметрами.

Перевантаження підпрограм також називають поліморфізмом підпрограм. Термін поліморфізм походить від двох грецьких слів – *полі* (багато) і *морфі* (форма). Таким чином, поліморфна підпрограма – це підпрограма, що характеризується різноманіттям форм.

Для забезпечення перевантаження підпрограм потрібно для кожного імені підпрограми визначити кілька різних варіантів заголовка, які дають різні припустимі способи звертання.

Щоб підпрограми були ***перевантажуваними***, тобто щоб компілятор міг розпізнати однойменні підпрограми, необхідне виконання двох умов:

1) однойменні підпрограми повинні мати різне число параметрів або відрізнятися в типі хоча б одного з параметрів;
2) у заголовку перевантажуваної підпрограм після крапки з комою має бути зазначене службове слово **`overload`** (перевантажити)

Зазначимо спеціально, що компілятор не реагує на те, чим є перевантажувані підпрограми, – процедурами або функціями. Наявність однойменних функцій, які відрізняються тільки типом значення, що поверта-

ється, неможлива, як неможлива і наявність однойменних процедури та функції з однаковими кількістю й типами параметрів.

Наприклад, у модулі Math визначено п’ять перевантажених функцій для знаходження найбільшого з двох числових значень:

```
function Max(A, B: Integer): Integer; overload;
function Max(A, B: Int64): Int64; overload;
function Max(A, B: Single): Single; overload;
function Max(A, B: Double): Double; overload;
function Max(A, B: Extended): Extended; overload;
```

Як це видно з наведених заголовків функцій, вони відрізняються одна від одної **типами параметрів**, кількість же параметрів у всіх цих функцій однакова – 2.

```
//Приклад 8.15
//Оформити дві перевантажені функції з ім'ям Max. Перша
//функція визначає найбільший з двох рядків, а друга –
//найбільший елемент масиву з N значень типу Real.
```

Оскільки ця задача є чисто ілюстративною, наведемо тільки можливі тексти функцій:

```
function Max(s1, s2: string): string; overload;
begin
  if s1 > s2 then Result := s1
  else Result := s2;
end;

function Max(a: array of Real; Count: Integer): Real;overload;
var
  i: Integer;
begin
  Result := a[0];
  for i := 0 to Count - 1 do
    if a[i] > Result then Result := a[i];
end;
```

Тексти функцій не вимагають особливих коментарів через їхню простоту та очевидність. Відзначимо лише два моменти. По-перше, звернемо увагу на те, що формальний параметр a другої функції, будучи безрозмірним масивом, передбачає нумерацію його елементів усередині функції від нуля. По-друге, оскільки обидві функції описані зі службовим словом **overload**, якщо підключено модуль Math, зберігається звичайний доступ до перевантажених функцій Max цього модуля. У другій функції параметр Count введено у припущенні, що масив a може бути заповнений не до кінця.

Щоб запобігти можливим помилкам, потрібно уникати неоднозначності використання параметрів за умовчанням у перевантажених підпрограмах. Так, неоднозначними є такі заголовки підпрограм:

```
procedure Proc(I: Integer); overload;
procedure Proc(I: Integer; J: Integer = 0);overload;
```

Компілятор не може визначити, яка процедура повинна бути викликана таким оператором:

```
Proc(X);
```

У результаті буде генеруватися помилка.

У той же час при звертанні до функції із задаванням двох параметрів невизначеність відсутня, і помилки не буде.

8.11. Використання стандартних директив, асемблерних вставок, інструкцій машинного коду. Вставка фрагментів, написаних мовою Delphi

При описі підпрограм мовою Delphi можливе використання стандартних директив, які вказуються відразу ж за заголовком підпрограми і відокремлюються від нього крапкою з комою. Ці директиви задаються іменами **assembler**, **external**, **far**, **forward**, **near** і службовим словом **inline**. Дія директив поширюється тільки на ту підпрограму, слідом за заголовком якої вони зазначені.

Про директиву **forward** ішла мова в підрозд. 8.4 при розгляді випереджального опису. Директиви ж **far** (дальня модель виклику) і **near** (ближня модель виклику) зараз практично не застосовуються, оскільки вони передбачають використання сегментної моделі пам'яті, яка не в сучасних операційних системах практично не використовується.

Якщо вся процедура (крім заголовка) написана на вбудованому асемблері, то слідом за її заголовком (через крапку з комою) зазначається стандартна директива **assembler**, а далі після крапки з комою між службовими словами **asm** і **end** вміщується асемблерний текст процедури:

```
procedure ім'я_процедури(опис_форм_параметрів); assembler;
asm
    Асемблерний текст
end;
```

Аналогічно оформляється функція, написана на вбудованому асемблері. При цьому значення, що повертається функцією, повинне бути

занесене в змінну `Result` із зазначенням перед нею символу `@` (тобто `@Result`).

Відзначимо, що на асемблері можна не тільки оформляти підпрограми. У програмі, написаній мовою Delphi, можливе використання асемблерних вставок. Для цього необхідно розмістити написаний на асемблері фрагмент програми між службовими словами **asm** і **end**:

```
asm
   Фрагмент програми на асемблері
end;
```

У програмах, написаних мовою Delphi, можливе використання підпрограм, повністю написаних на асемблері та записаних після асемблювання в деякий об'єктний файл, що має стандартне розширення .OBJ. Підпрограма може бути написана й іншою мовою програмування з наступною компіляцією з одержанням об'єктного файлу. Однак не треба при цьому забувати про те, що підпрограми, які підключаються, повинні компілюватися з орієнтацією на прийняту в Delphi методику передачі параметрів. Для використання такої підпрограми необхідно в описовій частині програми за допомогою директиви компілятору `$L` підключити файл з об'єктним кодом підпрограми, після чого звичайним способом оголошується заголовок підпрограми з зазначенням (через крапку з комою) службового слова **external**:

```
{$L lib.obj}                    //Підключення файлу lib.obj
      //Оголошення заголовка зовнішньої підпрограми
function Hypoten(a, b: Real): Real; external;
```

Крім асемблерних вставок і зовнішніх підпрограм, у Delphi передбачена можливість використання інструкцій машинного коду, для чого служить **inline**-оператор, що складається зі службового слова **inline**, за яким у круглих дужках ідуть елементи машинного коду, розділені косою рисою.

Можливе також застосування стандартних директив **register**, **pascal**, **cdecl**, **safecall** і **stdcall**, які регламентують спосіб передачі параметрів у підпрограму, видалення параметрів зі стека, використання регістрів при передачі параметрів та обробку винятків і помилок.

За умовчанням передбачається, що підпрограма компілюється з директивою **register**. При зазначенні директиви **register** для передачі параметрів, крім стека, використовується до трьох регістрів центрального процесора; інші ж директиви регламентують передачу параметрів тільки через стек.

Директиви **register** і **pascal** визначають передачу параметрів зліва направо, тобто крайній зліва параметр передається і розміщається в стеку першим, а крайній справа – останнім. Директиви ж **cdecl**, **safecall** й **stdcall** визначають передачу параметрів у зворотному порядку.

При всіх зазначених директивах, крім **cdecl**, видалення параметрів зі стека виконується підпрограмою перед виходом з неї. Директива **cdecl** покладає відповідальність за видалення параметрів зі стека на підпрограму, що здійснювала виклик даної підпрограми, після виходу з останньої.

У програму (як у текст проекту, так й у текст кожного з модулів) дозволено вставляти тексти, що написані мовою Delphi і зберігаються в текстових файлах. Для цього у відповідному місці програми вказують директиву компілятору `$I`, а як її параметр вживається ім'я текстового файлу (наприклад, `{$I f.pas}`). Компіляція програми в цьому випадку проводиться так, ніби вміст файлу, що включається, розміщується в програмі безпосередньо в місці розташування директиви `$I`. Подібним чином можуть бути включені в текст програми підготовлені у вигляді окремих файлів константні масиви, процедури, функції тощо. Природно, розширення файлу, який вставляється, може бути довільним.

Запитання для контролю і самоконтролю

1. Як визначити функцію?
2. Як визначити процедуру?
3. У чому відмінність процедур і функцій і для чого вони використовуються?
4. Що таке формальні і фактичні параметри?
5. Що таке локальні змінні й у чому їхня особливість?
6. Що таке глобальні змінні та чим зумовлені рекомендації щодо обмеження їх використання?
7. Які види параметрів застосовуються при описі процедур і функцій та у чому їхні особливості?
8. У чому особливість передачі параметрів за значенням?
9. Що таке «передача параметра за адресою», яким способом вона забезпечується?
10. Для чого служать **var**-параметри?
11. У чому особливість **const**-параметрів?
12. Яке призначення **out**-параметрів?
13. Що таке випереджальний опис, для чого він використовується і як реалізується?

14. Що таке рекурсія?
15. Що таке відкритий масив?
16. Що таке константний масив?
17. Що таке конструктор масиву?
18. Як оголошуються і використовуються процедурні змінні?
19. Що таке параметри за умовчанням і як вони використовуються?
20. Що таке перевантаження підпрограм і які обмеження існують для їх визначення?
21. Яким способом визначається підпрограма, написана на вбудованому асемблері?
22. Чи може бути виконаний оператор `Pr([1,200,2])` для заголовка процедури `Pr`, що наведений нижче?

```
procedure Pr(a: array of Byte);
```

23. Нехай є такі описи:

```
type
  TSetOfByte = set of Byte;
procedure Pr(a: TSetOfByte); overload; forward;
procedure Pr(a: array of Byte); overload; forward;
```

а) Чи може бути виконаний в цьому випадку оператор `Pr([1, 200, 2])`, і якщо так, то яка з двох процедур буде викликатися?

б) Чи буде в цьому випадку коректним програмний код, що наводиться нижче?

```
var
  s: TSetOfByte;
begin
  s := [1, 200, 2];
  Pr(s);
```

Завдання для практичного відпрацьовування матеріалу

1. Дано цілі числа $p, n_0, d_0, n_1, d_1, \cdots, n_p, d_p, a, b$ $(d_0 \cdot d_1 \cdot \ldots \cdot d_p \cdot b \neq 0,$ $p \geq 0)$. Обчислити $\frac{n_p}{d_p}\left(\frac{a}{b}\right)^p + \frac{n_{p-1}}{d_{p-1}}\left(\frac{a}{b}\right)^{p-1} + \ldots + \frac{n_0}{d_0}$, одержавши результат у вигляді простого дробу. Визначити процедури повного скорочення числа, заданого чисельником і знаменником, а також процедури множення та додавання простих дробів.

2. Дано невід'ємні цілі числа n та m. Написати рекурсивну підпрограму для обчислення функції Акермана:

$$A(n,m)=\begin{cases} m+1, & \text{якщо } n=0, \\ A(n-1,1), & \text{якщо } n\neq 0,\ m=0, \\ A(n-1,A(n,m-1)), & \text{якщо } n>0,\ m>0. \end{cases}$$

3. Скласти рекурсивну підпрограму, що обчислює суму двох цілих невід'ємних чисел шляхом багаторазового додавання числа 1. Наприклад, 6 + 10 = (6 + 1) + 9 = (7 + 1) + 8 = ...
4. Дано квадратну матрицю ***A*** розміру $n\times n$, де n – натуральне число. Обчислити її визначник. Використати рекурсивну підпрограму обчислення визначника розкладанням за першим рядком.
5. Оформити три процедури або функції для обчислення площі трикутника за довжиною основи і висотою, площі прямокутника за довжиною його сторін, площі кругу за значенням його радіуса. У файлі записана інформація про тип фігур і їх геометричні розміри. Користуючись процедурним типом, обчислити площі фігур, характеристики яких наведені у файлі, і записати результат в інший файл.
6. Виконати завдання 5 з розд. 17, замінивши графічне відображення виведенням пари чисел (абсциса, ордината) в Memo-компонент. Використати процедурний тип для вибору виду кривої.
7. Натуральне число задане послідовністю цифр, записаною в числовому масиві або рядку символів. Визначити функцію або процедуру, що реалізує аналог операції **`shl`** зі збільшенням розрядності у випадку, якщо зсув приводить до втрати старших розрядів. За допомогою розробленої підпрограми для заданого зазначеним способом натурального числа n одержати 2^n.

9. МОДУЛІ

9.1. Призначення модулів та їх структура

Коли говорять про файли, що містить програми, написані в середовищі Delphi, йдеться про декілька типів файлів. Зокрема, це:

1) текстові файли з текстами програм, написаних мовою Delphi (ці файли називаються проектами і мають розширення .DPR);
2) текстові файли, що містять самостійні програмні одиниці, що звуться модулями (вони мають розширення .PAS);
3) файли, отримані після компіляції файлів з текстами модулів (вони мають розширення .DCU);
4) готові до виконання файли (виконувані файли), отримані у результаті компіляції програм (вони мають розширення .EXE).

Що ж таке DCU-файли?

У міру розвитку обчислювальної техніки з'явилася можливість у масовому порядку розв'язувати складні задачі, що вимагають написання великих програм із залученням для цього колективів розроблювачів. Виникло поняття модульного програмування, під яким розуміють і розбиття програми на окремі фрагменти, і використання файлів, що містять текстові фрагменти, з яких можна сформувати програму (в Delphi вставки в текст програми здійснюються за допомогою директиви компілятору `{$I` ім'я_файлу_що_включається`}`), і написання підпрограм, у тому числі зовнішніх (ця можливість також є в Delphi і реалізується застосуванням стандартної директиви **`external`**).

Можна також створювати програмні одиниці, що зберігаються і компілюються незалежно одна від одної і мають певний інтерфейс, який дозволяє використовувати різні ресурси цих програмних одиниць у розроблюваних програмах. Такі програмні одиниці в Delphi називають модулями; у результаті їхньої компіляції створюються файли з розширенням .DCU (DCU-файли). Застосування модулів дозволяє формувати великі програми (EXE-файли).

У Delphi модуль (**unit**) вважається окремою програмною одиницею і створюється спочатку як звичайний PAS-файл, що оформлюється за певними правилами:

```
unit ім'я_модуля;                    //Заголовок модуля
interface       //Заголовок інтерфейсної частини
uses список використовуваних модулів
type ...
const ...
var ...
procedure ...
function ...
implementation  //Заголовок розділу реалізації
  Опис локальних модулів, типів, констант, змінних,
  процедур і функцій, а також процедур і функцій,
  заголовки яких оголошені в розділі interface
initialization
  Розділ ініціалізації
finalization
  Розділ завершення
end.
```

Як і проект, текст модуля починається з заголовка і закінчується службовим словом **end** з крапкою. Однак заголовок модуля починається не зі слова **program**, а зі службового слова **unit**. Після слова **unit** вказується ім'я модуля, що обов'язково повинне збігатися з ім'ям того файлу, який містить модуль. Наприклад, для заголовка

```
unit module;
```

текст модуля повинен зберігатися у файлі module.pas, у результаті компіляції якого створюється файл module.dcu). Завершується заголовок крапкою з комою. Наведене в заголовку ім'я модуля вказується при його підключенні до програми або іншого модуля. Щоб підключити модуль до програми необхідно відразу ж слідом за заголовком проекту після службового слова **uses** зазначити ім'я модуля, що підключається:

```
uses module;
```

Службове слово **uses** може зустрітися в проекті тільки один раз, тому для підключення декількох модулів їхні імена подають через кому:

```
uses module1, module2;
```

До проекту можна підключити не тільки модуль, що уже відкомпільований (DCU-файл), але й модуль, розміщений у деякому PAS-файлі. Для

цього після імені модуля слідом за службовим словом **in** наводять укладене в апострофи ім'я файлу, що містить модуль; наприклад,

```
uses Unit1 in 'Unit1.pas';
```

За заголовком модуля йде його інтерфейсна частина, що починається зі службового слова **interface**. У ній зазначаються програмні ресурси (константи, типи, змінні, заголовки процедур і функцій), призначені для використання іншими модулями і програмами. Описані тут елементи є видимими поза модулем (зовнішніми). Вимоги до опису тут ті самі, що й взагалі в Delphi, але для процедур і функцій вказують тільки заголовки, причому описувачі **external**, **forward** і **assembler** не ставляться.

Відзначимо також, що в інтерфейсній частини модуля можна підключати модулі, ресурси яких використовуються в даному модулі. Для цього відразу ж після заголовка інтерфейсної частини відкривають розділ **uses**, у якому подають імена використовуваних модулів. Однак модулям, що використовують даний модуль, вони все-таки недоступні. Тому, щоб отримати в модулі доступ до ресурсів іншого модуля, останній потрібно оголосити явно, а не опосередковано (через інший модуль). При корекції інтерфейсної частини модуля можна застосувати браузер коду, про що говорилося в підрозд. 2.6.

Слідом за інтерфейсною частиною модуля йде розділ реалізації, який починається зі службового слова **implementation**. Тут оголошуються невідомі поза модулем його внутрішні елементи (їх ще називають невидимими, схованими), а саме, локальні змінні, константи, типи, процедури і функції, а також модулі, ресурси яких використовуються нижче по тексту даного модуля. Крім того, тут же реалізуються ті процедури і функції, заголовки яких описані в інтерфейсній частини і які відомі поза модулем. Такий метод опису процедур і функцій обрано через те що користувачеві для звертання до підпрограми достатньо знати її ім'я та список параметрів. Знання техніки її реалізації зовсім не потрібне. Якщо здійснюється модифікація підпрограми, поміщеної в модуль, то достатньо змінити тільки її тіло в розділі реалізації, не змінюючи заголовка в інтерфейсній частини. У зв'язку з цим програми, що використовують дану підпрограму, не змінюються. Якщо підпрограма оголошена в інтерфейсній частині, то в розділі реалізації її заголовок можна давати тільки у вигляді імені без вказівки переліку й типів параметрів, а також типу значення, що повертається, для функції (аналогічно випереджальному опису). Однак з метою поліпшення читаності тексту модуля рекомендується й у розділі реалізації вказувати повні заголовки підпрограм. При цьому повинна бути забезпечена повна відповідність заголовків в інтерфейсній частині модуля та у розділі реалізації.

Розділи **interface** і **implementation** обов'язкові; навіть якщо вони порожні, їхні заголовки зазначити потрібно.

Після розділу реалізації в модулі може розташовуватися розділ ініціалізації, що починається зі слова **initialization**, за яким ідуть оператори, які будуть виконуватися ***до*** операторів з тіла програми (наприклад, установка початкових значень для деяких змінних, оголошених у модулі). Розділ ініціалізації не обов'язковий, тому, якщо при підключенні модуля не потрібно робити ніяких початкових установок, він може бути опущений (разом зі службовим словом **initialization**). Замість службового слова **initialization** може використовуватися службове слово **begin** (але це більш притаманне Pascal-програмам).

Якщо в декількох модулях присутні розділи ініціалізації, то вони виконуються послідовно в порядку подання модулів у проекті й інших модулях (з урахуванням вкладеності при підключенні), причому розділ ініціалізації виконується тільки один раз незалежно від того, скільки разів здійснюється підключення того модуля, що містить його.

Розділ ініціалізації може бути порожнім, тобто не містити жодного оператора (у цьому випадку його краще опускати).

Якщо в модулі присутній розділ ініціалізації і він відкривається службовим словом **initialization**, то після нього може розміщуватися розділ завершення, що починається зі службового слова **finalization** і містить оператори, які повинні виконуватися ***по закінченні*** роботи програми (наприклад, звільнення пам'яті, запис на диск деяких файлів тощо). Якщо кілька модулів містять розділи завершення, то ці розділи виконуються у зворотному порядку щодо порядку, у якому ці модулі підключалися. Як і розділ ініціалізації, розділ завершення модуля може бути порожнім, і в цьому випадку його звичайно опускають.

9.2. Компіляція програм, що використовують модулі

До програми можна підключити декілька модулів. У свою чергу, модуль може залучати ресурси інших модулів. У цьому випадку в його тексті у специфікації **uses** наводять тільки імена тих модулів, які безпосередньо використовуються. Характерною рисою модулів є те, що вони не повністю включаються в EXE-файл: до програми додається з модуля тільки те, що буде використане. Явне або опосередковане звертання модулю до самого себе є забороненим, хоча, якщо модуль оголошується в розділі реалізації, це обмеження поширюється тільки на пряме звертання.

Можливий випадок, коли в даному модулі та в інтерфейсних частинах декількох інших модулів, підключених до нього, оголошені за допомогою одного й того ж імені різні об'єкти. Наприклад, якщо у даному модулі є оголошення

```
uses module1, module2;
```

і як у ньому, так і в інтерфейсних частинах підключених до нього модулів є описи з ім'ям `name`, то даний модуль оперує, насамперед, елементами, оголошеними в ньому самому (тобто тим елементом `name`, що оголошений у ньому). Для звертання до об'єкта, ім'я якого є «перекритим», необхідно ліворуч до імені об'єкта дописати ім'я модуля, у якому він оголошений, з'єднавши їх крапкою (наприклад, `module1.name` або `System.Real`).

У Delphi немає зв'язку між ім'ям програми (проекту) та ім'ям файлу, в якому вона (він) зберігається. У той же час для підключення модуля до програми він повинен бути знайдений на диску. Тому ім'я модуля та ім'я файлу, що містить модуль, повинні збігатися. Як це відзначалося вище, початковий текст модуля зберігається у файлі з розширенням .PAS, а отриманий у результаті компіляції код модуля – у файлі з розширенням .DCU (від Delphi Compile Unit). Так, якщо ім'я модуля `module`, то відповідними файлами будуть module.pas та module.dcu.

При компіляції програми за командою Project ► Compile Ім'я_проекту (Проект ► Компілювати Ім'я_проекту) компілятор послідовно відшукує DCU-файли, що містять коди використовуваних модулів, для їхнього підключення. Пошук проводиться в такий спосіб:

1. Перегляд системної папки Lib, призначеної для зберігання .DCU-файлів стандартних модулів (ця папка міститься в папці, у яку виконувалась інсталяція Delphi).

2. Якщо модуля немає в папці Lib, то пошук продовжується в поточному каталозі.

3. Якщо й тут модуль не знайдений, то йде перегляд каталогів, наведених в альтернативі Search Path (Шлях пошуку) на вкладці Directories/Conditionals (Каталоги/Умови) пункту меню Project ► Options (Проект ► Опції) з видачею діагностичного повідомлення і припиненням компіляції, якщо модуль не знайдений (каталоги в альтернативі Search Path зазначаються через крапку з комою).

При цьому всі модулі поточного проекту, що змінилися до моменту створення нового виконуваного файлу, динамічно підключуваної бібліотеки (.DLL), файлу ресурсу (.RES) і так далі, компілюються заново.

При компіляції програми за командою Project ► Build Ім'я_проекту (Проект ► Побудувати Ім'я_проекту) в обов'язковому порядку йде компіляція всіх PAS-файлів, що містять тексти використовуваних модулів. Якщо який-

небудь PAS-файлу відсутній, але існує відповідний DCU-файл, здійснюється підключення останнього.

Компілятор будує виконувані файли відповідно до таких правил:

- модулі завжди компілюються перед компіляцією програми;
- файл проекту (.DPR) завжди компілюється повторно;
- якщо змінився вихідний код модуля після того, як він компілювався востаннє, модуль компілюється повторно;
- якщо Delphi не може визначити місцезнаходження вихідного коду модуля, та модуль повторно не компілюється;
- якщо змінилася інтерфейсна частина модуля, всі інші модулі, які залежать від зміненого модуля, компілюються повторно;
- якщо модуль пов'язаний з OBJ-файлом (із зовнішніми підпрограмами), і цей файл змінився, здійснюється повторна компіляція модуля;
- якщо модуль містить файл, що включається за допомогою директиви компілятору `$I`, і цей файл змінився, модуль компілюється повторно.

9.3. Приклад оформлення модуля і його підключення

Розглянемо особливості реалізації та підключення модулів на конкретному прикладі.

```
//Приклад 9.1
//Модуль для роботи з комплексними числами: визначення
//типу комплексного числа, констант, процедур завдання
//(InitComplexValue) і виведення в багаторядковий редактор
//(WriteLnComplexValue) комплексного числа, додавання
//(AddComplexValues), множення (MultComplexValues) і
//ділення (DivComplexValues) комплексних чисел.
unit Compl_Un;

interface                          //Початок інтерфейсної частини

uses
  StdCtrls;

type
  Complex = record                                  //Тип комплексний
    Re, Im: Real;
  end;
```

```
const
  ComplexZero: Complex = (Re: 0; Im: 0);    //Комплексний нуль
  ImagineOne: Complex = (Re: 0; Im: 1);       //Уявна одиниця

procedure InitComplexValue(r, i: Real; var c: Complex);
procedure WriteLnComplexValue(c: Complex; mm: TMemo);
procedure SumComplexValues(c1, c2: Complex;
                           var Res: Complex);
procedure MultComplexValues(c1, c2: Complex;
                            var Res: Complex);
procedure DivComplexValues(Numerator, Denominator: Complex;
                           var Res: Complex);

implementation                                  //Розділ реалізації

uses
  SysUtils;

procedure WriteLnComplexValue;
var
  s: string;
begin
  s := '';
  with c do begin
    s := FloatToStr(Re);
    if Im > 0 then s := s + ' + ';
    if Im <> 0 then s := s + FloatToStr(Im) + 'i';
  end;
  mm.Lines.Add(s);
end;

procedure InitComplexValue;
begin
  with c do begin
    Re := r;
    Im := i;
  end;
end;

procedure SumComplexValues;
begin
  with Res do begin
    Re := c1.Re + c2.Re;
    Im := c1.Im + c2.Im;
  end;
end;

procedure MultComplexValues;
begin
```

```
  with Res do begin
    Re := c1.Re * c2.Re - c1.Im * c2.Im;
    Im := c1.Re * c2.Im + c1.Im * c2.Re;
  end;
end;

procedure DivComplexValues;
var
  z: Real;
begin
  with Denominator do
    z := Re * Re + Im * Im;
  with Res do begin
    Re := (Numerator.Re * Denominator.Re +
           Numerator.Im * Denominator.Im) / z;
    Im := (Denominator.Re * Numerator.Im -
           Numerator.Re * Denominator.Im) / z;
  end;
end;              //Розділи ініціалізації і завершення відсутні

end.
```

Розглянемо особливості наведеного тексту модуля.

Ім'я модуля `Compl_Un`. Текст модуля повинен зберігатися у файлі Compl_Un.pas.

В інтерфейсній частині оголошено тип `Complex`; цей тип використовується усередині модуля і придатний для застосування в будь-якій програмі або будь-якому модулі, до якого підключений модуль `Compl_Un`. У цьому ж розділі описано дві константи `ComplexZero` і `ImagineOne`; вони можуть бути використані як усередині модуля, так і в будь-якому модулі, до якого підключений модуль `Compl_Un`.

Оголошені в інтерфейсній частини п'ять процедур також можуть бути застосовані як усередині модуля, так і поза ним.

У розділі реалізації подано скорочені заголовки процедур, хоч їх можна повторити в тому ж вигляді, що й в інтерфейсній частині.

Розділ ініціалізації модуля відсутній, про що свідчить пропуск службових слів **`begin`** і **`initialization`**.

Аналогічно відсутній розділ завершення.

У цьому випадку крапка з комою перед **`end`** із крапкою обов'язкова (якби в модулі був присутній розділ ініціалізації, її довелося б поставити перед ним).

Як другий параметр процедури `WriteLnComplexValue` використано параметр типу `TMemo` з метою забезпечення можливості звертання до Memo-компонента. Оскільки тип `TMemo` оголошений у стандартному

модулі StdCtrls, для забезпечення доступу до цього типу в розділі **interface** підключено модуль StdCtrls.

У секції **implementation** підключено стандартний модуль SysUtils для забезпечення можливості звертання до функції FloatToStr.

Для використання згаданих вище типу, констант і процедур, що працюють із комплексними числами, достатньо у модулі форми в розділі **uses** оголосити модуль Compl_Un. Після цього з ресурсами модуля можна працювати так, ніби вони були оголошені в самому модулі форми.

Зазначимо також, що підпрограми InitComplexValue, SumComplexValues, MultComplexValues та DivComplexValues можна оформити і як функції, що повертають значення типу Complex, який оголошений у модулі.

Скористаємося ресурсами розробленого вище модуля для розв'язання наступної задачі.

```
//Приклад 9.2
//Уводяться два комплексних числа. Вивести їх
//добуток. Скористатися ресурсами модуля Compl_Un.
```

Наведемо повний текст модуля форми для цієї задачі.

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, ExtCtrls,
  Compl_Un;                        //Підключення модуля Compl_Un

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    lbOutput1: TLabel;
    edInput1: TEdit;
    Button1: TButton;
    bbClose: TBitBtn;
    mmOutput1: TMemo;
    edInput2: TEdit;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private

  public
    c, c1, c2: Complex;
  end;
```

```
var
  Form1: TForm1;

implementation
{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  DecimalSeparator := '.';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  r, i: Real;
begin
  case Tag of
    0: begin
        r := StrToFloat(edInput1.Text);
        i := StrToFloat(edInput2.Text);
        InitComplexValue(r, i, c1);
        mmOutput1.Lines.Add('Перше число');
        WriteLnComplexValue(c1, Form1.mmOutput1);
        lbOutput1.Caption := 'Дійсна та уявна частини' +
                             ' другого числа';
        edInput1.SetFocus;
       end;
    1: begin
        r := StrToFloat(edInput1.Text);
        i := StrToFloat(edInput2.Text);
        InitComplexValue(r, i, c2);
        mmOutput1.Lines.Add('Друге число');
        WriteLnComplexValue(c2, Form1.mmOutput1);
        Button1.Caption := 'Обчислити';
       end;
    2: begin
        MultComplexValues(c1, c2, c);
        mmOutput1.Lines.Add('Добуток дорівнює ');
        WriteLnComplexValue(c, Form1.mmOutput1);
        edInput1.Hide;
        edInput2.Hide;
        lbOutput1.Hide;
        Button1.Enabled := False;
       end;
  end;
  Tag := Tag + 1;
end;

end.
```

У даному модулі відомий тип `Complex` і процедури `InitComplexValue`, `MultComplexValues` та `WriteLnComplexValue` за рахунок оголошення модуля `Compl_Un` в **`uses`**-рядку секції **`interface`**. Тільки ці елементи модуля збільшують EXE-файл, всі інші елементи (константи `ComplexZero`, `ImagineOne`, процедури `AddComplexValues` та `DivComplexValues`) в EXE-файл не потрапляють. Підключення модуля `Compl_Un` у секції **`implementation`** буде помилковим, оскільки поля `c`, `c1` та `c2` класу `TForm1` описані за допомогою визначеного в модулі `Compl_Un` типу `Complex`.

Текст проекту матиме стандартний вигляд:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Підкреслимо, що підключення модуля `Compl_Un` до проекту не обов'язкове (що видно з тексту проекту).

Запитання для контролю і самоконтролю

1. Для чого служать модулі в Delphi?
2. Як оголошуються та підключаються модулі?
3. Яка загальна структура модуля?
4. Як здійснюється компіляція програм за наявності декількох модулів?

Завдання для практичного відпрацьовування матеріалу

1. У модулі оформити власні підпрограми для перетворення в рядку великих (малих) українських літер у малі (великі), повороту вмісту рядка на 180°, перевірки того, що рядок є паліндромом і т. д. Використати ресурси модуля в програмі.

2. Оформити модуль для роботи з файлами цілих даних: підпрограми для пошуку позиції, у якій розміщується елемент даних, видалення у файлі елемента, що стоїть у заданій позиції, видалення у файлі всіх елементів із заданим значенням і т. д. Використати ресурси модуля в програмі.
3. Оформити модуль для обчислення значення визначеного інтеграла різними методами – методами прямокутників і методом трапецій. Використати ресурси модуля в програмі. Підінтегральну функцію задати як функцію із трьома коефіцієнтами.

   Визначений інтеграл деякої функції, заданої на інтервалі [a, b] – це площа криволінійної трапеції, обмеженої кривою, віссю абсцис, і вертикальними лініями, що проходять через межі інтервалу. Для його наближеного обчислення *методом прямокутників* інтервал [a, b] розбивають на підінтервали, на яких будують прямокутники, висоти яких дорівнюють значенню функції на лівій (правій) межі відповідного підінтервалу або в його середині. Як значення визначеного інтеграла беруть суму площ таких прямокутників.

   У *методі трапецій* замість прямокутників використовуються трапеції, побудовані на отриманих підінтервалах.
4. Виконати завдання 1 з розд. 8, визначивши процедури (функції) в окремому модулі. Визначити в тому ж модулі константи 0 та 1, що задаються у вигляді простих дробів.
5. Два натуральних числа a і b задані послідовністю цифр, записаною в числовому масиві або рядку символів. Обчислити $a + b$, $a - b$, $a \times b$. Визначити процедури або функції для обчислення суми, різниці та добутку двох чисел, заданих зазначеним способом, та оформити їх в окремому модулі.

10. ВАРІАНТИ

10.1. Поняття варіантного типу і його подання в пам'яті

У деяких випадках у програмі виникає необхідність у використанні даних, тип яких не може бути визначений під час компіляції або який може змінюватися по ходу виконання програми. Для таких випадків у Delphi розроблено спеціальний тип `Variant`, що служить для подання значень, які можуть змінити свій тип під час виконання програми. Варіанти забезпечують більшу гнучкість, але потребують більше пам'яті, ніж звичайні змінні, і дії над ними здійснюються значно повільніше, ніж над статично зв'язаними типами. Крім того, некоректні дії над варіантами часто завершуються помилками часу виконання програми, у той час як подібні помилки у випадку звичайних змінних були б помічені компілятором.

За умовчанням змінні типу `Variant` можуть містити значення будь-якого типу, крім записів, множин, статичних масивів, файлів, класів, посилань на класи (див. підрозд. 11.8) і вказівників. Вони можуть містити цілі, дійсні та булівські значення, значення типу `TDateTime`, рядки, динамічні масиви, а також спеціальний вид масиву, що називається ***варіантним масивом***. Варіантні змінні, крім того, сумісні з OLE-об'єктами. У виразах варіанти можуть змішуватися з іншими варіантами та даними перелічених вище типів. Перетворення типів, якщо воно взагалі можливе, у цьому випадку здійснюється автоматично. Автоматичне перетворення типу відбувається і при присвоюванні варіантних значень змінним іншого типу (знову ж, якщо це можливо).

У пам'яті змінна типу `Variant` займає 16 байт, причому перші 2 байти (поле `VType`) служать для кодування типу значення, що зберігається в даний момент.

Структура варіантної змінної визначається записом типу `TVarData`, що описаний у модулі `System` у такий спосіб:

```
type
  TVarArrayBound = packed record
    ElementCount: Integer;
    LowBound: Integer;
  end;
  TVarArrayBoundArray = array[0..0] of TVarArrayBound;
  PVarArray = ^TVarArray;
  TVarArray = packed record
    DimCount: Word;
    Flags: Word;
    ElementSize: Integer;
    LockCount: Integer;
    Data: Pointer;
    Bounds: TVarArrayBoundArray;
  end;
  TVarType = Word;
  TVarData = packed record
    VType: TVarType;
    case Integer of
      0: (Reserved1: Word;
          case Integer of
            0: (Reserved2, Reserved3: Word;
                case Integer of
                  varSmallInt:(VSmallInt: SmallInt);
                  varInteger: (VInteger: Integer);
                  varSingle:  (VSingle: Single);
                  varDouble:  (VDouble: Double);
                  varCurrency:(VCurrency: Currency);
                  varDate:    (VDate: Double);
                  varOleStr:  (VOleStr: PWideChar);
                  varDispatch:(VDispatch: Pointer);
                  varError:   (VError: LongWord);
                  varBoolean: (VBoolean: WordBool);
                  varUnknown: (VUnknown: Pointer);
                  varShortInt:(VShortInt: ShortInt);
                  varByte:    (VByte: Byte);
                  varWord:    (VWord: Word);
                  varLongWord:(VLongWord: LongWord);
                  varInt64:   (VInt64: Int64);
                  varString:  (VString: Pointer);
                  varAny:     (VAny: Pointer);
                  varArray:   (VArray: PVarArray);
                  varByRef:   (VPointer: Pointer);
              );
```

```
            1: (VLongs: array[0..2] of LongInt);
        );
      2: (VWords: array [0..6] of Word);
      3: (VBytes: array [0..13] of Byte);
  end;
```

Крім коду типу, у варіантну змінну записується або значення, або вказівник на значення, тип якого відповідає коду.

Щоб визначити тип значення, що зберігається у даний момент у варіантній змінній, використовують функцію

`VarType(`варіант`)`,

яка повертає зчитаний з поля `VType` код збереженого у варіанті значення. Безпосередній доступ до поля `VType` заборонений.

Код типу є цілочисловим значенням, причому в модулі `System` визначені такі константи, що відповідають різним типам значень, які можуть зберігатися у варіантних змінних:

```
varEmpty    = $0000;    // Значення Unassigned
varNull     = $0001;    // Невідомі дані - Null
varSmallInt = $0002;    // Значення типу SmallInt
varInteger  = $0003;    // Значення типу Integer
varSingle   = $0004;    // Значення типу Single
varDouble   = $0005;    // Значення типу Double
varCurrency = $0006;    // Значення типу Currency
varDate     = $0007;    // Значення типа TDateTime
varOleStr   = $0008;    // OLE-UNICODE-рядок
varDispatch = $0009;    // Посилання на OLE-об'єкт
varError    = $000A;    // Код помилки операц. системи
varBoolean  = $000B;    // Значення типу Boolean
varVariant  = $000C;    // Значення типу Variant
varUnknown  = $000D;    // Невідомий OLE-об'єкт
varShortInt = $0010;    // Значення типу ShortInt
varByte     = $0011;    // Значення типу Byte
varWord     = $0012;    // Значення типу Word
varLongWord = $0013;    // Значення типу LongWord
varInt64    = $0014;    // Значення типу Int64
varStrArg   = $0048;    // COM-сумісний рядок
varString   = $0100;    // Значення типу string
varAny      = $0101;    // Будь-яке CORBA-значення
varTypeMask = $0FFF;    // Бітова маска від VarType
```

```
varArray    = $2000;    // Варіантний масив
varByRef    = $4000;    // Вказівник на дані
```

Функція `VarType` повертає не обов'язково одне із наведених вище значень, а, можливо, їхню комбінацію. Так, якщо варіантна змінна є варіантним масивом (див. нижче) з елементами типу `Byte`, то функція `VarType`, застосована до такої змінної, поверне значення `8209` (`$2011`), тобто `varArray+varByte`.

При створенні варіантів усі вони ініціюються особливим значенням `Unassigned` типу `Variant`. Ще одне особливе значення `Null` вказує на невідомі дані (насправді обидва ці значення є результатом звертання до однойменних функцій, визначених у модулі `Variants`).

Звісна річ, можливе створення динамічних масивів, елементами яких є варіанти. Однак ні статичний, ні динамічний масив не можуть бути значеннями змінної варіантного типу. У цьому випадку використовують спеціальний вид варіантних значень – варіантні масиви (див. підрозд. 10.3).

10.2. Особливості виразів, що використовують варіанти

Усі цілі, дійсні (у тому числі типу `TDateTime`), булівські та символьні значення, а також дані типу **`string`** сумісні за присвоюванням з варіантами. При записі значення у варіантну змінну приведення типів здійснюється автоматично (за умови, що це взагалі можливо). У складних виразах, що містять варіантні дані, результат залежить від порядку виконання дій і може бути досить несподіваним.

При записі у варіантну змінну нового значення відбувається автоматичне оновлення поля `VType` відповідно до типу значення, записуваного у варіантну змінну.

Автоматичне перетворення варіантних значень відбувається і при їх записі в змінні інших типів. Особливості подібного роду перетворень представлені в таблиці 10.1.

Відзначимо те, що варіанти, які містять рядкові значення, не можуть бути індексовані. При цьому виникає помилка часу виконання, не контрольована компілятором. Крім того, до варіантних даних не можна застосовувати операції `^`, **`is`** та **`in`**.

Якщо варіант містить дійсне значення, то при його записі в рядок (незалежно від того, змінювалося чи ні значення системної змінної `DecimalSeparator`) як роздільник цілої та дробової частин завжди використовується регіональний роздільник (у нас – кома).

При присвоюванні дійсній змінній записаного у варіант рядкового значення правильне перетворення рядка до дійсного числа здійснюється і при задаванні як роздільник цілої та дробової частин регіонального роздільника, і при задаванні роздільника, записаного в системну змінну `DecimalSeparator`.

Таблиця 10.1 – Перетворення варіантних значень

Тип перетворюваного варіантного значення	Тип приймаючої змінної			
	Цілий	Дійсний	Рядок	Булівський
Цілий	До цілого	До дійсного	До рядка	`0`→`False`, інакше →`True`
Дійсний	Округлення до найближчого цілого	До дійсного	До рядка з урахуванням регіонального подання	`0` → `False`, інакше → `True`
Рядок	До цілого з відсіканням дробової частини і збудженням винятку при нечисловому рядку	До дійсного з урахуванням регіонального подання й збудженням винятку при нечисловому рядку	До рядка	`'False'`→`False`, числове значення рядка `0`→`False`, `'True'`→`True`, числове значення рядка не `0`→`True`; інакше – виняток
Булівський	`False`→`0`, `True`→`-1` (для `Byte`→`255`)	`False`→`0`, `True`→`-1`	`False`→`'0'`, `True`→`'-1'`	`False`→`False`, `True`→`True`
Unassigned	`0`	`0`	`''`	`False`
Null	Виняток	Виняток	Виняток	Виняток

```
//Приклад 10.1
//Приклади перетворення варіантів.
```

Скористаємося формою з прикладу 2.2, видаливши з неї мітку виведення та однорядкове редаговане текстове поле:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  V1, V2, V3: Variant;
  S: string;
  I: Integer;
  B: Boolean;
```

```
  R: Real;
begin
  DecimalSeparator := '.';
  V2 := '2.6';
  R := V2;        //Тут ураховується значення DecimalSeparator
  mmOutput1.Lines.Add(FloatToStr(R));         //Виведеться 2.6
  V1 := 15;
  V2 := '2,6';
  V3 := -3.8;
  R := V2;          //Тут ураховується регіональний роздільник
  mmOutput1.Lines.Add(FloatToStr(R));         //Виведеться 2.6
            //Нижче значення DecimalSeparator не враховується
  mmOutput1.Lines.Add(V1 + V2 + V3);         //Виведеться 13,8
  I := -V2;                                          //Тут i=-3
  mmOutput1.Lines.Add(V1 + V2 + I);          //Виведеться 14,6
  S := V3;                                          //S='-3,8'
  mmOutput1.Lines.Add(V1 + V2 + S);          //Виведеться 13,8
  B := V3;                                            //B=True
  mmOutput1.Lines.Add(BoolToStr(B));         //Виведеться '-1'
end;
```

У змінні `V1`, `V2` і `V3` записуються відповідно ціле число, рядок і дійсне значення. При цьому записуваний у змінну `V2` рядок у перший раз являє собою запис дійсного числа у форматі, прийнятому в оригінальній версії Windows, а в другий – у русифікованій версії. В обох випадках при присвоюванні значення `V2` дійсній змінній `R` перетворення проходить правильно.

У випадку русифікованої версії Windows у виразі `V1 + V2 + V3` значення `V2` перетвориться до числа, у результаті чого сума буде дорівнювати `13.8`. Отримане значення має тип `Variant`, який автоматично може бути перетворений до рядкового подання, що й буде здійснене при виведенні в Memo-поле. При цьому дійсне значення виводиться з урахуванням регіонального подання його в рядку.

При записі значення `-V2` у цілочислову змінну `I` відбувається автоматичне округлення значення до найближчого цілого. Запис значення `V3` у рядкову змінну `S` приводить до формування рядка з регіональним поданням роздільника цілої та дробової частин. Надалі демонструється трактування виразу `V1 + V2 + S`, що містить два варіантних (`V1` та `V2`) та одне рядкове (`S`) значення. Наостанок розглядається перетворення до булівського типу: оскільки `V3` є відмінним від `0`, у булівську змінну `B` записується значення `True`, що функцією `BoolToStr` перетвориться в значення `'-1'`.

Окрім згаданої вище функції `VarType`, є ще кілька підпрограм для роботи з варіантними даними:

- функція `VarIsEmpty(`варіант`)` – повертає `True`, якщо варіант не містить даних (`Unassigned`);
- функція `VarIsNull(`варіант`)` – повертає `True`, якщо варіант містить значення невідомого типу (`Null`);
- процедура `VarClear(`варіант`)` – звільняє динамічну пам'ять, якщо вона була пов'язана з варіантом, і записує в поле `VType` параметра варіант значення `varEmpty`;
- функція `VarToStr(`варіант`)` – перетворює варіант до рядка, повертаючи значення типу **`string`**;
- функція `VarToDateTime(`варіант`)` – перетворює варіант до значення типу `TDateTime`.

Існують й інші підпрограми для роботи з варіантами.

10.3. Варіантні масиви

Варіантний масив – це одне з можливих значень варіанта. Особливістю таких масивів є те, що в них можуть бути записані будь-які значення, допустимі для варіантів (за винятком даних типів `ShortString` та `AnsiString`).

Індексами варіантного масиву можуть бути тільки цілі числа.

Варіантний масив створюється звертанням до однієї з функцій `VarArrayCreate` або `VarArrayOf`, що повертають значення типу `Variant`.

Функція

```
VarArrayCreate(межі, тип_значень)
```

має два параметри, першим з яких є відкритий цілочисловий масив меж вимірів (пари значень, що задають нижню та верхню межі вимірів – першого, другого і т. д.), а другий визначає тип елементів варіантного масиву. Другий параметр задається цілим числом – кодом типу варіанта (див. підрозд. 10.1).

Так, якщо `V` – змінна типу `Variant`, то оператор

```
V := VarArrayCreate([1, 10, 2001, 2008], varDouble)
```

з першим параметром, який є масивом констант, створює двовимірний масив з дійсних чисел з 10 рядків та 8 стовпців. При цьому перший індекс

має мінімальне значення 1, а максимальне значення 10. Діапазон зміни другого індексу – відрізок 2001..2008.

Функція

VarArrayOf(перелік_значень)

з параметром, що є відкритим масивом варіантів, створює одновимірний варіантний масив, із записом у нього значень, перелічених у параметрі. Елементи створеного масиву нумеруються від нуля.

Кількість вимірів варіантного масиву повертає функція

VarArrayDimCount(варіант).

Якщо її параметр не є варіантним масивом, то повертається значення 0.

Наступні дві функції повертають відповідно верхню та нижню межі зміни індексів варіантного масиву за виміром, заданим значенням другого параметра:

VarArrayHighBound(варіант, номер_виміру),

VarArrayLowBound(варіант, номер_виміру).

Так, якщо розглянути створений вище варіантний масив V, то результатом звернення VarArrayLowBound(V, 2) буде значення 2001.

Оскільки значеннями елементів варіантного масиву можуть бути варіанти (створюються з кодом varVariant), то у варіантний масив можна записувати дані різних типів, у тому числі й рядки.

```
//Приклад 10.2
//Розробити функцію, що здійснює множення на 2 цілого
//невід'ємного значення (десяткове подання результату
//може мати не більш 20 цифр).
```

У припущенні, що формальним параметром є або значення типу Int64, або варіантний масив типу Byte, у який записані цифри, що утворюють запис числа, функція може мати такий вигляд:

```
function Mult2(V: Variant): Variant;
const
  MaxInt64: Int64 = High(Int64);
var
  m: Int64;
  i, c, VAH, k: Byte;
  Flag: Boolean;
begin
  Flag := True;
  while Flag do begin            //Цикл повториться, якщо число
    //потрапляє в діапазон для Int64, а подвоєне значення – ні
```

```
    Flag := False;
    case VarType(V) of          //Довідуємося про тип варіанта
      varInt64:                                   //Тип Int64
       if V > MaxInt64 div 2 then    //Не можна помножити на 2
       begin
         m := V;                        //Запам'ятовуємо число
         i := Length(V);           //Кількість цифр у записі V
                        //Створюємо варіантний масив типу Byte
         V := VarArrayCreate([1, Length(V)], varByte);
         repeat                      //і заповнюємо його з кінця
           V[i] := m mod 10;  //цифрами, починаючи з останньої
           m := m div 10;                     //в записі числа
           Dec(i);
         until m = 0;
         Flag := True;    //Для забезпечення повторної обробки
       end
       else Result := V * 2;               //Звичайне множення
      varArray + varByte: begin   //Варіантний масив типу Byte
        VAH := VarArrayHighBound(V, 1);   //Верхня межа масиву
        if V[1] > 4 then k := 1//У результату на розряд більше
        else k := 0;           //або стільки ж, що і в множника
                     //Створюємо результуючий варіантний масив
        Result := VarArrayCreate([1, VAH + k], varByte);
        c := 0;       //Значення розряду переносу при множенні
        for i := VAH downto 1 do begin          //Цикл множення
          Result[i + k] := (c + 2 * V[i]) mod 10;
          c := (c + 2*V[i]) div 10;//Корекція розряду переносу
        end;
        if c > 0 then Result[1] := c;
        VarClear(V);   //Знищуємо масив (якщо він створювався)
      end;
    end; {case}
  end; {while}
end;
```

У функції визначена константа `MaxInt64` – максимальне можливе значення для типу `Int64`. Якщо параметр функції перевищує половину значення `MaxInt64`, але не перевищує `MaxInt64`, то множення в типі `Int64` буде неможливим, і створюється варіантний масив `V`, у який записуються цифри переданого числа, після чого проводиться обробка цього масиву. Якщо число представлене послідовністю цифр, записаною у варіантному масиві, то при множенні здійснюється послідовна обробка цифр, починаючи з наймолодшої. Далі масив `V` знищується. У звичайному ж випадку здійснюється звичайне множення на `2` значення типу `Int64`. Значенням, що повертається, є варіант, у який записаний або варіантний масив типу `Byte`, або значення типу `Int64`.

Крім стандартних варіантів в Delphi можливе створення так званих ***користувальницьких варіантів***, які нами розглядатися не будуть.

Запитання для контролю і самоконтролю

1. Що таке варіантні змінні?
2. Опишіть особливості перетворення типу при використанні варіантних даних?
3. Назвіть підпрограми для роботи з варіантними даними.
4. Що таке варіантний масив?
5. Як встановити тип значення, що зберігається у варіантній змінній?
6. Чи є помилка в процедурі, що наведена нижче?

```
procedure TForm1.Button1Click(Sender: TObject);
var
  V: Variant;
begin
  V := VarArrayCreate([0, 9], varInteger);
  V := -27;
  //...
end;
```

   За наявності помилки дайте пояснення. Якщо помилка відсутня, що буде записано в змінній `V`?
7. У чому особливості наступного фрагмента програмного коду?

```
procedure TForm1.Button1Click(Sender: TObject);
var
  V: Variant;
begin
  V := VarArrayCreate([0, 9], varInteger);
  //...
end;
```

Завдання для практичного відпрацьовування матеріалу

1. Дано масив варіантів. Скільки в цьому масиві дійсних чисел, що мають типи `Double` та `Currency`? Визначити процедуру або функцію для розв'язання цієї задачі.
2. Нехай у програмі визначена та ініціалізована змінна варіантного типу. Якщо її значенням є варіантний масив, елементами якого є варіанти, то записати в текстовий файл усі текстові рядки, наявні в цьому масиві.

11. ЕЛЕМЕНТИ СУЧАСНОГО ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

11.1. Визначення класів та основні поняття об'єктно-орієнтованого програмування

Основою об'єктно-орієнтованого програмування (ООП) є ідея об'єднання в єдиній структурі даних і дій, які виконуються над цими даними (у термінах ООП – це методи).

Поняття об'єктів як спеціального типу було властиве ще попередниці Delphi – мові Turbo Pascal. У класичному розумінні *об'єкт* – це структура, що поєднує в собі дані різних типів (поля) і процедури та функції (методи), що використовують ці дані.

ООП базується на трьох поняттях – інкапсуляції, спадкуванні та поліморфізмі.

Інкапсуляція – це комбінування даних з методами, які обробляють ці дані, результатом чого є новий тип даних – об'єкт (**`object`**), що є деяким чином «самодостатнім» з погляду опису та обробки характеристик, які властиві деякому реальному об'єкту (реальній задачі).

Спадкування – це можливість використання раніше визначених об'єктів для формування нових об'єктів («спадкоємців»), що успадковують описи даних «прабатька» та доступ до його методів.

Поліморфізм – це можливість визначення єдиної за ім'ям дії (процедури або функції), застосовної одночасно до всіх рівнів ієрархії спадкування. При цьому кожен об'єкт ієрархії може вказувати особливості дії над самим собою.

Об'єктова модель Delphi, порівняно з об'єктовою моделлю мови Turbo Pascal, значно вдосконалилася, результатом чого стала поява так званих класів (при цьому сам термін «клас» запозичений з мови C++). Класична об'єктова модель у Delphi практично не використовується,

будучи повністю поглиненою новою моделлю, що базується на концепції класів.

Класами в Delphi називаються спеціальні **типи**, що містять поля, методи і властивості.

Структура опису даного типу є аналогічною структурі запису, але при цьому замість службового слова **`record`** вживається службове слово **`class`**:

```
type
  ім'я_класу = class
                 поля;
                 заголовки_методів;
                 властивості;
  end;
```

Методами називають процедури і функції, призначені для обробки полів і властивостей класів. В описі типу класу зазначають тільки заголовки методів, а реалізуються вони пізніше (як при випереджальному описі звичайних процедур і функцій). При цьому перед ім'ям методу необхідно подати ім'я класу (ім'я_класу) і крапку. Перелік параметрів при реалізації методів наводити не обов'язково.

Поля класів аналогічні полям записів і служать для зберігання даних, що характеризують клас. Фундаментальним принципом об'єктно-орієнтованого програмування, що стосується і класів у Delphi, є вимога звертання до полів тільки за допомогою методів (а в класах – і за допомогою властивостей), хоча доступ до полів класів, аналогічний доступу до полів записів, у принципі можливий.

Методи класу оперують, насамперед, його полями і властивостями, але можуть маніпулювати й іншими змінними, описаними в програмі. Однак оскільки концепція класів передбачає, що вся інформація про деякий фізичний об'єкт міститься у відповідному класі, оголошеному в програмі, подібного роду дії є небажаними. При використанні класів бажано писати програму так, щоб методи, оголошені в класі, зверталися тільки один до одного, а також до полів даних і властивостей цього ж класу. Рекомендується також писати програму так, аби звертання до полів класу здійснювалося тільки за допомогою методів і властивостей цього класу.

Властивості – це спеціальний механізм, що регламентує доступ до полів класів. Поняття властивості було відсутнє в старій об'єктовій моделі.

Таким чином, у контексті класів ***інкапсуляція*** – це об'єднання в єдине ціле полів, методів і властивостей, що досить часто називаються ***сутностями*** класів. У принципі окремі сутності класів можуть бути відсутні.

Наприклад, якщо задача пов'язана з обробкою інформації про точки в тривимірному просторі, то в програмі можливе визначення класу «точка», характеристиками якого є координати точки в просторі. Обумовлений клас може також містити два методи – задання координат точки і безпосередньої обробки (у даному випадку конкретне формулювання задачі обробки інформації непринципове):

```
type
  TPoint3 = class                              //Опис класу
    X, Y, Z: Real;                             //Опис полів
    procedure Init;          //Опис методу ініціалізації
    procedure Run;                    //Опис методу обробки
  end;                                 //Кінець опису класу

procedure TPoint3.Init;                  //Реалізація методу
begin
  X := 0.5;                 //Надання значення координаті X
  Y := 0.5;                 //Надання значення координаті Y
  Z := 0.5;                 //Надання значення координаті Z
end;

procedure TPoint3.Run;                   //Реалізація методу
begin
       //Оператори, що реалізують обробку
end;
```

У наведеному вище прикладі передбачається ініціалізація всіх трьох координат значенням `0.5`. У реальній задачі повинна бути описана конкретна процедура або функція ініціалізації (наприклад, введення значень), яка, природно, може мати формальні параметри.

Доступ методів до полів класу здійснюється за рахунок того, що при виклику методу в нього, крім фактичних параметрів, передається невидимий параметр `Self` («свій»), що вказує, якому класу належить метод. Параметр `Self` неявно приєднується до всіх імен, що збігаються з іменами полів, методів і властивостей відповідного класу. Наприклад, у наведеному вище фрагменті програми в методі `Init` оператор `X:=0.5` сприймається так, ніби він був записаний у вигляді `Self.X:=0.5`. Параметр `Self` звичайно не зазначають, але його можна записувати й у явному вигляді.

Методи класу мають доступ не тільки до полів класу, але й до його властивостей та інших методів. Доступ методу до властивостей та інших методів класу здійснюється аналогічно тому, як це робиться щодо полів, тобто для забезпечення доступу достатньо навести ім'я властивості або методу (природно, при цьому можна зазначати й параметр `Self`).

Змінні, що мають тип **class**, називають ***екземплярами класу***. У Delphi замість терміну «екземпляр класу» найчастіше вживають термін «***об'єкт***», що у мові Turbo Pascal використовувався в іншому значенні. Методи класу викликаються із вказівкою імені екземпляра класу. Особливістю класів Delphi, порівняно з об'єктами мови Turbo Pascal, є те, що екземпляри класу завжди автоматично створюються в динамічній пам'яті, тобто змінна типу «клас» є вказівником на ділянку динамічної пам'яті. Звертання до такої змінної приводить до автоматичного її розіменування, у зв'язку з чим при звертанні до екземплярів класів символ ^ після імені екземпляра класу не подають (більш того, зазначення цього символу в такому контексті **заборонене**).

Таким чином, опис екземпляра класу `TPoint3` може бути, наприклад, таким:

```
var
  Point: TPoint3;
```

Опис екземпляра класу без попереднього визначення в розділі типів самого класу неможливий – завжди необхідно спочатку встановити тип класу, визначивши для нього ім'я, а потім раніше описаний тип використовується при описі екземплярів класу.

Безпосереднє звертання до полів, методів і властивостей екземпляра класу здійснюється аналогічно тому, як це робиться для полів записів, тобто можливі, наприклад, такі оператори:

```
Point.Init;                  //Звертання до TPoint3.Init
Point.X := 0.5;             //Присвоювання значення полю
```

Як це відзначалося вище, концепція класів (принцип інкапсуляції) вимагає звертання до полів класу за допомогою його методів і властивостей. Це означає, що для занесення інформації про координати в описаний вище екземпляр класу `Point` потрібно скористатися методом `TPoint3.Init`, а не звертатися до його полів безпосередньо.

Варто також пам'ятати про те, що класи створюються в динамічній пам'яті, а опис екземпляра класу фактично відводить пам'ять тільки під вказівник, а не безпосередньо під об'єкт. На відміну від змінних простих типів, будь-який екземпляр класу повинен бути створений **явно**. Розподіл об'єкта в динамічній пам'яті здійснюється спеціальним методом класу, який називається ***конструктором*** (див. підрозд. 11.3). Конструктори є у всіх класів, і звертання до полів і методів екземпляра класу без попереднього звертання до його конструктора приводить до помилки часу виконання програми. Таким чином, або конструктор має бути викликаний перед звертанням до методу ініціалізації чи перед безпосередньою

ініціалізацією полів об'єкта, або конструктор має сам виконувати ініціалізацію створюваного екземпляра класу. Другий варіант ініціалізації є загальноприйнятим. Наявність конструкторів у всіх класів пояснюється тим, що всі класи є нащадками (див. підрозд. 11.2) визначеного в Delphi спеціального класу `TObject`, який містить конструктор `Create`, що не має параметрів.

При завершенні роботи з екземпляром класу треба його видалити з динамічної пам'яті, для чого викликається спеціальний метод, так званий ***деструктор***. Цей метод руйнує екземпляр класу зі звільненням динамічної пам'яті, відведеної під нього. Як і конструктор, деструктор є у всіх класів. Якщо він не визначений явно, то використовується деструктор `Destroy` класу `TObject`.

При описі класів слід пам'ятати про існуючі обмеження:

– поля даних повинні стояти перед властивостями і заголовками методів;

– описи типу **`class`** не можуть зустрічатися локально усередині підпрограм.

Як і записи, екземпляри класів можуть брати участь в операторі **`with`**:

```
with Point do begin
  Init;
  Run;
end;
```

Розглянемо приклад, що демонструє методику опису найпростіших класів та обробку даних, що мають тип «клас».

```
//Приклад 11.1
//Дано чотири дійсних числа A, B, C, D, що визначають
//площину Ax+By+Cz+D=0 у тривимірному просторі.
//Дано також три дійсних числа u, v, w, що задають
//координати точки (u, v, w) у тривимірному просторі.
//Перевірити приналежність точки площині.
```

Скористаємося формою з прикладу 2.2, розмістивши додатково на панелі ще дві кнопки Button2 і Button3 з накладенням їх на кнопку Button1.

У програмі компонент Button1 буде відповідати за введення коефіцієнтів рівняння площини, компонент Button2 – за створення та ініціалізацію (введення координат) об'єкта «точка», а компонент Button3 – за обробку координат точки (власне, за розв'язання задачі).

Порівняно з прикладом 2.2, змінимо властивості деяких з компонентів й установимо значення властивостей нових компонентів відповідно до такого списку:

- Мітка:
  Caption — Коефіцієнти рівняння площини
- Кнопка Button1:
  Caption — A
- Кнопка Button2:
  Caption — Точка
  Visible — False
- Кнопка Button3:
  Caption — Перевірити
  Visible — False

У секції **interface** модуля після опису форми додамо опис двох типів:

```
type
  TPlane = array [0..3] of Real;//Коеф-ти рівняння площини
  TPoint3 = class                          //Опис класу "точка"
    X, Y, Z: Real;          //Опис полів (координати точки)
        //Опис заголовків методів ініціалізації та обробки
    procedure Init(ed: TEdit);
    procedure Run(Plane: TPlane; mm: TMemo);
  end;
```

У секції **implementation** опишемо масив коефіцієнтів рівняння площини та екземпляр класу:

```
var
  ABCD: TPlane;      //Масив коефіцієнтів рівняння площини
  Point: TPoint3;                          //Опис класу "точка"
```

Тут же для класу `TPoint3` проведемо реалізацію методів ініціалізації й обробки:

```
procedure TPoint3.Init(ed: TEdit);          //Визначення методу
var                    //ініціалізації екземпляра класу "точка"
  i: Integer;
  s: string;
begin
  i := 1;
  X := 0; Y := 0; Z := 0;                 //Значення за умовчанням
  s := Trim(ed.Text);             //Зчитування координат у рядок s
                                  //Нижче - запис координат у поля
  while (s <> '') and (i < 4) do begin
    s := Trim(s) + ' ';
    case i of
     1: X := StrToFloat(Copy(s, 1, Pos(' ', s) - 1));
     2: Y := StrToFloat(Copy(s, 1, Pos(' ', s) - 1));
     3: Z := StrToFloat(Copy(s, 1, Pos(' ', s) - 1));
```

```
    end;
    Delete(s, 1, Pos(' ', s));
    Inc(i);
  end;
end;

procedure TPoint3.Run(Plane: TPlane; mm: TMemo);       //Метод
begin                          //обробки екземпляра класу "точка"
  if Plane[0] * X + Plane[1] * Y + Plane[2] * Z +
     Plane[3] = 0 then
    mm.Lines.Add('Yes')
  else
    mm.Lines.Add('No');
end;
```

Крім того, створимо наступні опрацьовувачі події OnCreate для форми і події OnClick для компонентів Button1, Button2 та Button3:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DecimalSeparator:='.';
end;

procedure TForm1.Button1Click(Sender: TObject);
const
  Coef = 'BCD';
begin
  ABCD[Tag] := StrToFloat(edInput1.Text);
  Tag := Tag + 1;
  if (Tag < 4) then
    Button1.Caption := Coef[Tag]
  else begin
    Button1.Hide;
    Button2.Show;
    lbOutput1.Caption := '3 координати точки через пробіл';
    Tag := 0;
  end;
  edInput1.SetFocus;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Button2.Hide;
  Point := TPoint3.Create;        //Створення екземпляра класу
  Point.Init(edInput1);       //Ініціалізація екземпляра класу
  edInput1.Hide;
  lbOutput1.Hide;
  Button3.Show;
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  Point.Run(ABCD, mmOutput1);        //Обробка екземпляра класу
  Button3.Hide;
  Point.Destroy;                   //Руйнування екземпляра класу
end;
```

У даному випадку визначено клас TPoint3 (тип «точка в тривимірному просторі»), що має три поля X, Y, Z (координати точки) і методи-процедури Init та Run. Перший з методів служить для надання значень полям (ініціалізації) у припущенні, що дані будуть надходити через формальний параметр, що має тип TEdit. Другий метод служить для безпосереднього розв'язання задачі і передбачає обробку переданих у нього за допомогою формального параметра Plane коефіцієнтів рівняння площини з наступним виведенням результату розв'язання задачі у формальний параметр, що має тип TMemo. Програмна реалізація цих методів виконана в розділі **implementation** модуля. Показано, що при написанні програмного коду методу, що входить у клас, у заголовку необхідно зазначати ім'я класу (у цьому випадку TPoint3), зв'язане з ім'ям методу крапкою. Аналізуючи коди процедур TForm1.Button2Click і TForm1.Button3Click, можна зробити висновок, що для звертання до методів Init і Run класу TPoint3 потрібно знати тільки заголовки цих методів без специфіки їхньої реалізації.

Розглянемо стисло процес виконання програми.

Після запуску програми і появи на екрані форми вводяться коефіцієнти рівняння площини в описаний у секції **implementation** модуля масив ABCD за командами, що надходять від кнопки Button1. Далі після кліку над кнопкою Button2 у процедурі TForm1.Button2Click за допомогою виклику конструктора Create (див. підрозд. 11.3) створюється екземпляр класу TPoint3 у вигляді змінної Point, описаної в секції **implementation** модуля. Для створення екземпляра класу застосовується метод (конструктор) Create, наявний у всіх класів, оскільки він успадковується від класу TObject. Після створення екземпляра класу змінна Point ініціалізується за допомогою виклику методу Point.Init, як фактичний параметр якого використано однорядкове редаговане текстове поле edInput1 форми. Особливістю реалізації методу TPoint3.Init є те, що він призначений для одноразового читання рядка зі свого фактичного параметра, причому рядок повинен містити три розділених одним або декількома пробілами числа, що визначають координати точки. Оскільки на початку програмного коду цього методу для трьох координат задаються нульові значення, у випадку

введення менше трьох чисел відсутні значення залишаться нульовими; при введенні ж більше трьох чисел зайві дані ігноруються (контроль правильності введення не передбачено).

В опрацьовувачі події OnClick кнопки Button3 викликається метод `Point.Run` з масивом `ABCD` коефіцієнтів рівняння площини та багаторядковим редагованим текстовим полем `mmOutput1` форми як фактичними параметрами. Цей метод призначений для перевірки приналежності точки площині, а також виведення результату перевірки. Останній оператор процедури `TForm1.Button3Click` за допомогою деструктора `Destroy` знищує раніше створений екземпляр класу.

Відзначимо особливо таке: методи `Point.Init` та `Point.Run` «знають» поля `X`, `Y`, `Z` свого класу, і турбуватися про передачу імен `X`, `Y`, `Z` усередину цих методів не потрібно.

Для класів існує можливість реалізації випереджального опису. Для цього в описі класу після його імені й знака рівності наводять тільки службове слово **`class`**:

```
type
  ім'я_класу = class;
```

Випереджальний опис має бути розшифрований визначенням класу з тим же ім'ям у секції опису типів, у якій воно розміщене, тобто між випереджальним описом класу і його визначенням не повинне знаходитися нічого, крім інших описів типу:

```
type
  TClass1 = class; //Випереджальний опис класу TClass1

  TClass2 = class; //Випереджальний опис класу TClass2

  TClass1 = class    //Визначальний опис класу TClass1
    //.......
    FieldClass: TClass2;   //Використання типу TClass2
    //.......
  end;

  TClass2 = class    //Визначальний опис класу TClass2
    //.......
    FieldClass: TClass1;   //Використання типу TClass1
    //.......
  end;
```

Як це видно з наведеного вище прикладу, випереджальний опис дозволяє визначити взаємозалежні класи.

11.2. Спадкування та перевизначення. Класи і модулі

Будь-який клас може породити ***похідний класовий тип***, що успадковує всі поля, методи та властивості прабатьківського типу і може мати свої власні нові поля, методи та властивості. Нові поля, методи та властивості просто додаються. Якщо ім'я методу, оголошеного в ***класі-нащадку***, збігається з ім'ям якого-небудь методу прабатьківського типу, то відбувається ***перевизначення*** цього методу (***перекриття*** методу класу-прабатька). У цьому випадку класу-нащадку відомий метод, оголошений усередині нього самого, а звичайний доступ до однойменного методу прабатька виявляється неможливим.

Доступ до перекритих методів прабатьків здійснюється зазначенням імені методу з попереднім ім'ям класу-прабатька, що містить викликуваний метод (ці імена зв'язуються символом «крапка»). Для доступу до перекритого методу безпосереднього прабатька існує інша методика: перед ім'ям такого методу необхідно подати службове слово **`inherited`**. Якщо перекритий метод прабатька викликається з однойменного перекриваючого методу нащадка, що має ті ж самі параметри, то при подібному виклику ім'я методу може бути опущене (оператор у такому разі має вигляд **`inherited`**;). Це означає, що метод предка викликається з тими ж самими значеннями параметрів, з якими було виконане звертання до методу нащадка, що перекрив його.

При перевизначенні методів до всіх нащадків переходить перевизначений метод, поки він не буде ще раз перевизначений на якому-небудь рівні ієрархії спадкування.

Для визначення типу класу, як похідного від уже існуючого, слід після слова **`class`** у дужках навести ім'я прабатьківського типу:

```
type
  ім'я_класу-спадкоємця = class(ім'я_класу-предка)
    нові_поля_класу-спадкоємця;
    нові_методи_класу-спадкоємця;
    нові_властивості_класу-спадкоємця;
  end;
```

Рекомендується завжди починати з визначення елементарного рівня даних і методів, переходячи до більш складних понять за допомогою переміщення від рівня до рівня в ієрархії спадкування.

Наприклад, якщо точка може переміщатися за деяким правилом у нове положення, то для визначеного в попередньому прикладі класу

TPoint3 можна визначити породжений клас, який є його нащадком (***дочірнім класом***) і має таке визначення:

```
TPoint3Move = class(TPoint3)        //Опис класу-нащадка
  XFin, YFin, ZFin: Real;             //Опис нових полів
  procedure Move;                   //Опис нового методу
end;
```

У цьому випадку клас TPoint3Move разом з полями X, Y, Z, що задають початкове положення точки, містить нові поля XFin, YFin, ZFin (кінцеве положення точки). Крім того, цей клас, маючи успадковані від класу TPoint3 методи Init і Run, має свій власний новий метод Move, що описує спосіб переміщення точки зі стартового положення в кінцеве положення. Природно, необхідно здійснити програмну реалізацію коду цього методу (його повне ім'я TPoint3Move.Move).

Особливістю Delphi є те, що в ньому визначений спеціальний клас TObject, який не має своїх полів і методів, але містить ряд методів загального призначення, які повинні мати будь-які класи. Цей метод є предком усіх створюваних класів, навіть якщо це не зазначено явно (як це було в розглянутому прикладі). Таким чином, створити клас, що не є нащадком класу TObject, неможливо, тобто ідентичними є такі два рядки:

```
ім'я_класу = class
```

і

```
ім'я_класу = class(TObject)
```

Наприклад, у попередньому прикладі перший рядок у визначенні класу TPoint3 міг бути записаний у такий спосіб:

```
TPoint3 = class(TObject)
```

Правилом «гарного тону» в об'єктно-орієнтованому програмуванні є така вимога: якщо в похідному типі описана власна процедура ініціалізації (тобто процедура виконання початкових дій з екземпляром класу, наприклад, надання початкових значень полям), то в ній на самому початку повинна викликатися процедура ініціалізації безпосереднього прабатька. Це гарантує заповнення всіх полів, які мають одержати свої значення при ініціалізації, оскільки прабатьківська процедура ініціалізації забезпечить надання значень полям нащадка, які успадковані від прабатька.

Як говорилося вище, в Delphi будь-який породжений клас може звернутися до перекритого методу свого безпосереднього прабатька, для чого введено службове слово **inherited**, що наводиться перед ім'ям методу через пробіл.

Наприклад, нехай описаний вище ланцюжок спадкування продовжений класом, що має опис

```
TPoint3MoveModif = class(TPoint3Move)
  procedure Move;
  procedure Run;
end;
```

Якщо метод `Run` класу `TPoint3MoveModif` викликає метод `Move` свого класу, то достатньо виконати такий оператор:

```
Move;
```

У той же час метод `Run` класу `TPoint3MoveModif` може звернутися й до методу `Move` класу `TPoint3Move`. Для цього оператор виклику методу повинен мати такий вигляд:

```
inherited Move;
```

Розглянемо приклад, у якому використовується властивість спадкування при роботі з класами.

```
//Приклад 11.2
//Створити таку ієрархію спадкування класів:
//рівень 0 – клас "точка на площині", що має
//наступні методи: введення координат, виведення координат
//з попереднім повідомленням; рівень 1 – клас
//"відрізок на площині" з методами введення координат
//кінців відрізка, обчислення довжини відрізка, виведення
//результату обчислення; рівень 2 – клас "опуклий
//багатокутник на площині" з методами введення кількості
//вершин та їх координат й обчислення площі багатокутника
//як суми площ трикутників, що його утворюють,
//з обчисленням площі трикутника за формулою Герона.
```

Створимо форму, що містить кнопку введення Button1, багаторядковий редактор, а також бітову кнопку закриття форми.

- Форма:
  Name — Form1
- Кнопка Button1:
  Caption — Виконати
- Багаторядковий редактор виведення:
  Name — mmOutput1
  Lines — очистити
- Бітова кнопка закриття форми:
  Kind — bkClose

Name — BitBtn1

Крім того, розмістимо на формі розкривний список вибору (компонент ComboBox з вкладки Standard), змінивши значення деяким з його властивостей:

Name — cbChoise1
Style — csDropDown
Text — Точка

Вибравши в Інспекторі Об'єктів у компонента ComboBox властивість Items і клацнувши мишкою праворуч від цієї властивості над кнопкою, що містить три крапки, потрапимо у вікно редактора елементів списку, у якому наберемо три рядки:

Точка
Відрізок
Багатокутник

Над розкривним списком вибору помістимо мітку виведення, установивши її властивостям Caption і Name відповідно значення lbOutput1 і Вибери тип об'єкта.

Оголосимо у секції **interface** модуля перед описом форми такі типи:

```
type
  TPoint = record                          //Записний тип "точка"
    X, Y: Real;
  end;

  TPointClass = class                    //Клас "точка на площині"
    Point: TPoint;                             //Точка (стартова)
    Mess: string;                          //Поле для повідомлення
    constructor Create(P: TPoint);
    procedure Write(M: string; Memo: TMemo);       //Виведення
  end;

  TLineSegment = class(TPointClass)           //Клас "відрізок"
    PointF: TPoint;          //Кінець відрізка (фінішна точка)
    constructor Create(P1, P2: TPoint);
    //Для правильної роботи потрібно прибрати фігурні дужки
    function Calculate: Real; {virtual;}  //Обчислення довжини
    procedure Write(M: string; Memo: TMemo);       //Виведення
    function LengthSegment(P1, P2: TPoint): Real;    //Довжина
  end;

  TPolygon = class(TLineSegment)          //Клас "багатокутник"
    QuantityOfVertexes: Integer; //Кільк. вершин багатокутника
    Polygon: array of TPoint;           //Вершини багатокутника
    constructor Create;
```

```
    //Для правильної роботи потрібно прибрати фігурні дужки
    function Calculate: Real; {override;}   //Обчислення площі
  end;
```

У секцію **private** опису форми включимо опис трьох полів:

```
    Obj0: TPointClass;                 //Екземпляр класу "точка"
    Obj1: TLineSegment;              //Екземпляр класу "відрізок"
    Obj2: TPolygon;             //Екземпляр класу "багатокутник"
```

У секції **implementation** модуля перед описом опрацьовувача події OnClick кнопки Button1 опишемо такі методи класів TPointClass, TLineSegment і TPolygon:

```
constructor TPointClass.Create;
begin                                  //Ініціалізація поля Point
  Point := P;     //Те ж саме, що й Point.X:=P.X;Point.Y:=P.Y;
end;

procedure TPointClass.Write;        //Виведення координат точки
begin
  Memo.Lines.Add(M);                    //Виведення повідомлення
  with Point do begin       //Безпосередньо виведення координат
    Memo.Lines.Add('X=' + FloatToStr(X));
    Memo.Lines.Add('Y=' + FloatToStr(Y));
  end;
end;

constructor TLineSegment.Create;
begin           //Спочатку викликаємо успадкований конструктор
  inherited Create(P1);         //Можна TPointClass.Create(P1);
  PointF := P2;   //Ініціалізація кінцевої точки (поля PointF)
end;

function TLineSegment.Calculate;    //Метод обчислення довжини
begin     //При обчисленні звертаємося до методу LengthSegment
  Result := LengthSegment(Point, PointF);
end;
                              //Метод виведення довжини відрізка
procedure TLineSegment.Write;
begin      //Звертаємося до методу обчислення довжини відрізка
  Memo.Lines.Add(M + FloatToStr(Calculate));
end;

function TLineSegment.LengthSegment;         //Довжина відрізка
begin
  with P1 do
    Result := Sqrt(Sqr(X - P2.X) + Sqr(Y - P2.Y));
end;
```

```
constructor TPolygon.Create;
var
  i: Integer;
begin
  QuantityOfVertexes := StrToInt(InputBox(
         'Введення кількості вершин багатокутника',
         'Уведіть кількість вершин багатокутника', '1'));
  SetLength(Polygon, QuantityOfVertexes);   //Створення масиву
  for i := 0 to QuantityOfVertexes - 1 do
    with Polygon[i] do begin
     X := StrToFloat(InputBox('Введення вершин багатокутника',
            'Координата X вершини ' + IntToStr(i + 1), '0'));
     Y := StrToFloat(InputBox('Введення вершин багатокутника',
            'Координата Y вершини ' + IntToStr(i + 1), '0'));
    end;
end;

function TPolygon.Calculate;          //Метод обчислення площі
var
  i: Integer;
  Len1, Len2, Len3, L: Real;
begin
  Result := 0;
  for i := 0 to QuantityOfVertexes - 3 do begin
    Len1 := LengthSegment(Polygon[i], Polygon[i + 1]);
    Len2 := LengthSegment(Polygon[i], Polygon[i + 2]);
    Len3 := LengthSegment(Polygon[i + 1], Polygon[i + 2]);
    L := (Len1 + Len2 + Len3) / 2;         //L – напівпериметр
                                                  //трикутника
                              //Підсумовування площ трикутників
    Result := Result + Sqrt(L * (L - Len1) *
                            (L - Len2) * (L - Len3));
  end;
end;
```

Створимо також наступний опрацьовувач події OnClick для компонента Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  P1, P2: TPoint;
begin
  Button1.Hide;
  case cbChoise1.ItemIndex of
    1: begin                                 //Працюємо з відрізком
         Form1.Caption := 'Обчислення довжини відрізка';
         with P1 do begin
           X := StrToFloat(InputBox(
                       'Введення координат початку відрізка',
```

```
                        'Координата X', '0'));
          Y := StrToFloat(InputBox(
                        'Введення координат початку відрізка',
                        'Координата Y', '0'));
        end;
        with P2 do begin
          X := StrToFloat(InputBox(
                      'Введення координат кінця відрізка',
                      'Координата X', '0'));
          Y := StrToFloat(InputBox(
                      'Введення координат кінця відрізка',
                      'Координата Y', '0'));
        end;
                     //Нижче створюємо екземпляр - "відрізок"
        Obj1 := TLineSegment.Create(P1, P2);
        with Obj1 do begin
          Mess := 'Довжина відрізка дорівнює ';//Повідомлення
          Write(Mess, mmOutput1);         //Виведення довжини
        end;
      end;
    2: begin                          //Працюємо з багатокутником
        Form1.Caption := 'Обчислення площі багатокутника';
                                       //Створюємо багатокутник
        Obj2 := TPolygon.Create;
        with Obj2 do begin
          Mess := 'Площа багатокутника дорівнює ';
          Write(Mess, mmOutput1);           //Виведення площі
        end;
      end;
    else begin                                //Працюємо з точкою
          Form1.Caption := 'Виведення координати точки';
          with P1 do begin
           X:=StrToFloat(InputBox('Введення координат точки',
                         'Координата X точки ', '0'));
           Y:=StrToFloat(InputBox('Введення координат точки',
                         'Координата Y точки ', '0'));
          end;
                        //Нижче створюємо екземпляр - "точка"
          Obj0 := TPointClass.Create(P1);
          with Obj0 do begin
            Mess := 'Координати точки:';       //Повідомлення
            Write(Mess, mmOutput1);     //Виведення координат
          end;
        end;
  end;
  cbChoise1.Hide;
  lbOutput1.Hide;
end;
```

Розглянемо деякі особливості наведеного тексту модуля. Тип `TPoint` введено для опису координат точки й використано при описі стартової точки `Point` у класі `TPointClass` (клас «точка»), кінця відрізка `Point` у класі `TLineSegment` (клас «відрізок на площині») і масиву точок `Polygon` у класі `TPolygon` (клас «опуклий багатокутник на площині»).

Крім поля `Point`, клас `TPointClass` має поле `Mess`, призначене для зберігання повідомлення. Конструктор `TPointClass.Create` служить для створення екземпляра класу `TPointClass` і запису в його поле `Point` координат точки, переданих через його параметр `P`. Його єдиний оператор `Point:=P` ілюструє можливість присвоювання значення, що зберігається в одному записі, іншому запису того ж типу (див. п. 3.15.2). Метод-процедура `TPointClass.Write` служить для виведення рядка, заданого його першим параметром, в Memo-поле, задане другим параметром, а також виведення в зазначене Memo-поле координат точки `Point`.

Клас `TLineSegment`, будучи спадкоємцем класу `TPointClass`, має успадковані поля `Point` і `Mess`, а також нове поле `PointF`. У цього класу є власний конструктор `Create`, що перекриває однойменний конструктор прабатька, а також метод `Write`, що перекриває однойменний метод класу `TPointClass`. Крім того, у цього класу є два методи-функції `Calculate` і `LengthSegment`.

Конструктор `TLineSegment.Create` призначений для створення екземпляра класу `TLineSegment` і запису в поля `Point` та `PointF` значень координат початку й кінця відрізка. Координати початку й кінця відрізка передаються через параметри `P1` та `P2` конструктора. При цьому ініціалізація полів здійснюється стандартним способом: за допомогою виклику перекритого конструктора прабатька спочатку ініціалізується успадковане поле `Point`, після чого ініціалізується власне поле `PointF` екземпляра класу `TLineSegment`.

Метод `TLineSegment.Calculate` за допомогою звертання до методу `TLineSegment.LengthSegment` реалізує обчислення довжини відрізка, що з'єднує точки `Point` і `PointF`. При цьому показано, що для звертання до методу `TLineSegment.LengthSegment` усередині методу `TLineSegment.Calculate` немає необхідності вказувати ім'я класу, оскільки кожен метод «знає» всі поля, методи й властивості свого класу.

Метод `TLineSegment.Write` виводить у Memo-поле передане через перший параметр повідомлення, а також довжину відрізка, обчислену за допомогою методу `Calculate`.

Клас TPolygon має поля Point, PointF і Mess, які він успадковує від класу TLineSegment, будучи його безпосереднім нащадком. Крім того, у нього є власні нові поля – QuantityOfVertexes (кількість вершин багатокутника) і Polygon (масив вершин багатокутника). Від класу TLineSegment він успадковує методи LengthSegment і Write, маючи власний конструктор Create і власний метод Calculate, які перекривають однойменні методи його прабатька TLineSegment.

Конструктор TPolygon.Create при створенні екземпляра свого класу не звертається до конструкторів своїх предків, оскільки клас TPolygon не орієнтується на використання полів Point і PointF. Крім того, що він створює екземпляр класу, цей конструктор уводить кількість вершин, створює динамічний масив і заповнює його елементи. При введенні координат вершин багатокутника використовується функція InputBox (див. підрозд. 3.17).

Метод TPolygon.Calculate служить для обчислення площі опуклого багатокутника як суми площ трикутників, що утворюють його. При цьому площа трикутника обчислюється за формулою Герона:

$$S = \sqrt{L \cdot (L - a) \cdot (L - b) \cdot (L - c)},$$

де L – напівпериметр трикутника; a, b і c – довжини сторін трикутника.

Для обчислення довжини сторони трикутника застосовується успадкований метод-функція LengthSegment, а для виведення результату обчислень передбачено використання методу Write, успадкованого від безпосереднього прабатька TLineSegment.

Процедура TForm1.Button1Click служить для вибору варіанта розв'язуваної задачі, створюючи екземпляр одного з класів і звертаючись до того або іншого методу Write для виведення результату.

Розглянемо особливості тексту цієї процедури.

Вибір варіанта розв'язуваної задачі проводиться за допомогою компонента ComboBox, що являє собою розкривний список вибору, який служить для організації меню вибору варіанта і буде розглянутий пізніше (див. підрозд. 20.5). Цей компонент має властивість Items, призначену для зберігання кількох текстових рядків, один з яких вибирається стандартним для Windows методом (наприклад, кліком мишкою). Оскільки властивість Items є списком (див. розд. 12), його рядки пронумеровані від 0, й індекс обраного (сфокусованого) рядка автоматично записується у властивість ItemIndex. Значення цієї властивості використовується в перемикачі оператора **case**: 0 – точка, 1 – відрізок, 2 – багатокутник. Зазначимо, що в даному випадку список був заповнений на етапі проектування форми. Установлене на етапі проектування форми значення csDropDown властивості

Style указує на те, що у початковому стані список закритий і розкривається при натисканні кнопки праворуч від його редактора.

При виборі пункту «Точка» властивість `cbChoise1.ItemIndex` здобуває значення `0`, що забезпечує перемикання процедури `TForm1.Button1Click` на роботу з класом «точка» (альтернатива **`else`** оператора **`case`**). При цьому вводяться координати точки, а також створюється та ініціалізується за допомогою конструктора `TPointClass.Create` екземпляр `Obj0` класу `TPointClass` з наступним звертанням до методу `Write` цього класу для виведення результату.

При виборі пункту «Відрізок» у властивість `cbChoise1.ItemIndex` комбінованого списку вибору буде записане значення `1`, що у подальшому забезпечить виконання процедури `TForm1.Button1Click` з класом «відрізок». При цьому здійснюється введення координат кінців відрізка (змінні `P1` і `P2`), створення та ініціалізація за допомогою конструктора `TLineSegment.Create` екземпляра `Obj1` класу `TLineSegment` і звертання до методу `Write` цього класу, що виводить довжину відрізка, попередньо звернувшись до методу `TLineSegment.Calculate` для її обчислення.

При значенні `2` властивості `cbChoise1.ItemIndex` (пункт «Багатокутник») процедура `TForm1.Button1Click` за допомогою конструктора `TPolygon.Create` створює й ініціалізує екземпляр `Obj2` класу `TPolygon` і звертається до успадкованого від класу `TLineSegment` методу `Write` для виведення результату (площі багатокутника). Оскільки в успадкованому методі `Write` проводиться звертання до методу `Calculate`, а в класу `TPolygon` є однойменний метод, **ПЕРЕДБАЧАЄТЬСЯ**, що в цьому випадку буде здійснюватися звертання до методу `TPolygon.Calculate`, а не до методу `TLineSegment.Calculate`. У запропонованому варіанті реалізації методики опису функцій `TLineSegment.Calculate` і `TPolygon.Calculate` звертання до методу `TPolygon.Calculate` не виконуватиметься, і площа багатокутника буде виведена неправильно (точніше, вона взагалі **не буде обчислюватися**).

Щоб програма працювала правильно, слідом за заголовком методу `Calculate` в описі класу `TLineSegment` необхідно зазначити службове слово **`virtual`**, а слідом за заголовком методу `Calculate` в описі класу `TPolygon` – службове слово **`override`** (див. наступний підрозділ). Ці службові слова в наведеному вище тексті зазначені як два коментаря у фігурних дужках, і **для правильної роботи програми фігурні дужки потрібно прибрати**.

Особливістю змінних типу **`class`** є те, що їм можна присвоювати як значення відповідного типу, так і значення будь-якого похідного типу, тобто спадкоємець може передати записані в ньому значення предкові (але не навпаки).

Так, для визначених у попередній програмі змінних `Obj0`, `Obj1`, `Obj2` правильними є такі оператори:

```
Obj0 := Obj1;
Obj0 := Obj2;
Obj1 := Obj2;
```

Оскільки всі три змінні є вказівниками, то при цьому фактично забезпечується передача посилання на екземпляр класу-спадкоємця.

Якщо в наведених вище операторах присвоювання поміняти місцями ліву й праву частини, то компілятор видасть помилку.

Якщо екземпляр прабатька був ініціалізований екземпляром дочірнього класу, то, незважаючи на те, що він у цьому випадку посилається на ділянку пам'яті, займану спадкоємцем, звичайний доступ до полів і методів може бути реалізований тільки для тих полів і методів, які властиві прабатьківському класу. Для доступу до поля, оголошеного в дочірньому класі, необхідно тип екземпляра прабатька, з яким проводиться робота, привести (перетворити) до типу нащадка, що його ініціалізував.

Наприклад, для описаних у попередній програмі змінних `Obj0` та `Obj1` після виконання оператора присвоювання `Obj0 := Obj1` оператор

```
Obj0.PointF.X := -1.5;
```

виявиться помилковим (повідомлення про помилку буде видане компілятором).

Для забезпечення доступу потрібно привести тип об'єкта `Obj0` до типу `TLineSegment`, тобто скористатися оператором

```
TLineSegment(Obj0).PointF.X := -1.5;
```

Поліморфні механізми, які реалізовані в класах Delphi, приводять до того, що під час виконання програми не можна гарантовано ідентифікувати тип конкретного екземпляра класу. Для безпечного ж приведення типів необхідно їх точна ідентифікацію. У візуальному програмуванні існує спеціальний механізм, призначений для визначення типу об'єкта під час виконання програми. Цей механізм позначають абревіатурою RTTI, що розшифровується як «Run Time Type Information» – «Інформація про типи часу виконання». Змінні RTTI-типів компілятор постачає додатковою інформацією про їхній фактичний тип, що дозволяє одержати інформацію про тип змінної при виконанні програми.

Зокрема, в Delphi для класів визначена спеціальна ***операція ідентифікації типу***, що задається службовим словом **is**. Ця операція є бінарною, причому першим її операндом є екземпляр класу, а другим – ім'я типу. Результатом виконання операції **is** є булівське значення `True`, якщо екземпляр класу, що перевіряється, має тип, заданий другим операндом операції, або є нащадком цього типу. У противному випадку (або якщо екземпляр класу, що перевіряється, не ініціалізований) результатом виконання операції **is** є значення `False`. Перевірка може проводитися в такий спосіб:

```
if Obj0 is TLineSegment then ...
```

Безпосередньо перетворення типу екземпляра класу в Delphi здійснюють за допомогою спеціальної ***операції перетворення типів***, що задається службовим словом **as**. Як і операція **is**, ця операція також бінарна, причому першим її операндом є екземпляр класу, а другим – ім'я типу, до якого перетворюється тип першого операнда.

Різниця між традиційним приведенням типів і застосуванням операції **as** полягає в тому, що при використанні операції **as** у випадку несумісності типу об'єкта і типу, до якого здійснюється приведення, генерується виняток `EInvalidCast` (див. розд. 13).

Отже, наведений вище оператор повинен бути записаний у такий спосіб:

```
(Obj0 as TLineSegment).PointF.X := -1.5;
```

Ще розумнішим є використання оператора

```
if Obj0 is TLineSegment
  then TLineSegment(Obj0).PointF.X := -1.5;
```

За аналогією з доступом до полів може бути забезпечений доступ предка до нових методів нащадка:

```
if Obj0 is TLineSegment
  then TLineSegment(Obj0).Calculate;
```

Зазначимо, що спільне використання операцій **is** та **as** недоцільне, оскільки в цьому випадку буде виконуватися дворазова перевірка типу. Таким чином, останній наведений оператор не слід заміняти оператором

```
if Obj0 is TLineSegment
  then (Obj0 as TLineSegment).Calculate;
```

Традиційне приведення типів – значно швидша процедура, ніж операція **is**, оскільки остання переглядає всю ієрархію класів об'єкта, у зв'язку з чим можна рекомендувати для використання саме цю методику.

Не рекомендується замість поліморфізму використовувати RTTI-операції, до яких варто вдаватися тільки у виняткових випадках, коли один поліморфізм не може впоратись зі специфікою задачі.

Змінні типу **`class`** можуть уживатися як параметри процедур і функцій. При цьому діє те ж саме правило: фактичний параметр може мати не тільки той самий тип, що і формальний параметр, але й будь-який похідний від нього тип. За рахунок такої властивості сумісності класів забезпечується реалізація поліморфізму, результатом чого є така перевага: не потрібно піклуватися про те, щоб підпрограма викликала різні однойменні методи, властиві класам різних рівнів спадкування; це забезпечується автоматично за рахунок передачі відповідного фактичного параметра (правда, викликувані при цьому методи повинні бути віртуальними або динамічними).

Властивість спадкування, характерна для класів, визначає їхнє застосування в модулях. При цьому в інтерфейсній частині модуля дається опис класового типу, а методи реалізуються в розділі реалізації. Для використання описаних таким способом класів достатньо знати тільки інтерфейсну частину модуля. Якщо властивості оголошеного в модулі класу не задовольняють програміста, необхідно у своїй програмі оголосити клас-спадкоємець, додавши нові методи і перевизначивши ті з методів прабатьківського типу, які не потрібні нащадкові. Саме на цьому принципі побудована система спадкування компонентів Delphi

11.3. Віртуальні методи. Конструктори та деструктори

Як уже зазначалося, наведена в попередньому підрозділі програма містить помилку, що не виявляється компілятором і зумовлена некоректним викликом методу `Write` екземпляром класу `TPolygon`. Причина появи цієї помилки полягає в наступному. При компіляції коду методу `TLineSegment.Write` здійснюється зв'язування цього методу з викликуваним у ньому методом `Calculate`. У результаті, оскільки в цьому випадку мова йде про клас `TLineSegment`, метод `TLineSegment.Write` уже на етапі компіляції виявляється зв'язаним з визначеним у класі `TLineSegment` методом `TLineSegment.Calculate` (таке зв'язування називають ***статичним***, або ***раннім***). Якби не було альтернативи статичному зв'язуванню методів, неможлива була б реалізація такої фундаментальної властивості класів, як поліморфізм, що дозволяє виконувати настроювання властивостей екземплярів класів залежно від їхньої реалізації.

Отже, якщо в прабатьківському класі деякий метод Метод1 викликає статичний метод Метод2, то обидва ці методи жорстко зв'язуються при компіляції (оскільки вони компілюються в одному контексті). При цьому якщо спадкоємець викликає успадкований від предка метод Метод1, то завжди буде відбуватися звертання до методу предка Метод2, навіть якщо в спадкоємця є однойменний статичний метод Метод2, який фактично необхідно викликати.

Альтернативою статичному зв'язуванню методів, є ***динамічне*** (або ***пізнє***) зв'язування, що припускає його здійснення на етапі виконання програми залежно від типу екземпляра класу.

Для цього в батьківському класі метод, що потенційно може бути перекритий у якому-небудь дочірньому класі, повинен оголошуватися як ***віртуальний*** (з директивою **`virtual`**) або ***динамічний*** (з директивою **`dynamic`**). У дочірньому класі відповідний метод, що заміщає даний, має бути оголошений з директивою **`override`** (перекрити).

Використання віртуальних і динамічних методів забезпечує реалізацію ідеї поліморфізму: метод може викликатися щодо змінної верхнього рівня ієрархії спадкування, але те, який метод буде при цьому фактично викликаний, визначається не ім'ям об'єкта, а типом того об'єкта, на який посилається даний об'єкт.

Методи, що не є віртуальними або динамічними, називаються ***статичними***.

В описі класу, який має віртуальні методи, обов'язково повинен бути спеціальний метод, в оголошенні й реалізації якого службове слово **`procedure`** замінене словом **`constructor`**. Цей метод є особливим видом процедури (***конструктором***), призначеної для виконання настановних дій із забезпечення механізму віртуальних методів. Як уже зазначалося раніше, конструктор розміщає екземпляр класу в динамічній пам'яті. При цьому адреса початку області пам'яті, відведеної під екземпляр класу, записується в змінну `Self`, що є в кожному класі. Конструктор може успадковуватися і бути віртуальним або динамічним. Конструктор завжди викликається до першого виклику віртуального методу для даного екземпляра класу. Один клас може містити декілька конструкторів. Якщо в класі конструктор не оголошений явно, то викликається конструктор найближчого в списку спадкування прабатька, зокрема, як про це вже говорилося вище, використовується конструктор `Create` спільного для всіх класів прабатька – класу `TObject`. Конструктор (якщо в класу їх декілька, то один з них) повинен бути викликаний явно перед першим використанням екземпляра класу, оскільки саме конструктор створює екземпляр класу.

Конструктор записує нульові значення у всі звичайні поля і всі вказівники на класи, а також робить порожніми всі рядкові поля, після чого виконує інші дії, явно передбачені операторами, що містяться в ньому. Хоч конструктор у своєму визначенні не передбачає повернення ніякого значення, на завершення своєї роботи він завжди повертає посилання на створений об'єкт.

При створенні екземпляра класу виклик конструктора здійснюється в правій частині оператора присвоювання, у лівій частині якого наводять ім'я екземпляра класу, наприклад:

```
Obj0 := TPointClass.Create(P1);
```

Досить часто конструктор надає початкові значення полям класу (виконує ініціалізацію полів). Щоб забезпечити гарантоване заповнення всіх полів, рекомендується **в першому операторі** явно оголошуваного конструктора **класу-нащадка викликати конструктор його безпосереднього** прабатька (як це зроблено в конструкторі `TLineSegment.Create` у прикладі 11.2).

Для виконання завершальних дій над екземплярами класів введено спеціальний метод – ***деструктор***, що оформляється так само, як і метод-процедура, із заміною службового слова **`procedure`** на службове слово **`destructor`**. Цей метод служить для правильного звільнення пам'яті, яку займають екземпляри класів (руйнування класів), а також для виконання різних завершальних дій (наприклад, звільнення пам'яті, відведеної під динамічні поля).

Деструкторів в одного класу може бути декілька, вони можуть успадковуватися та бути віртуальними. Більш того, для гарантування виклик деструктора саме того класу, що руйнується, рекомендується описувати деструктори віртуальними. Якщо деструктор класу не описано явно, то застосовується деструктор `Destroy` класу `TObject`, що є віртуальним.

Рекомендується **наприкінці коду** явно описаного деструктора (якщо він є) **викликати деструктор безпосереднього прабатька**.

Викликається деструктор звичайним для методів-процедур способом (наприклад, `Obj.Destroy`).

Для іменування конструкторів і деструкторів можливе вживання будь-яких імен, але при цьому краще дотримуватися стандартних імен – `Create` та `Destroy`. Це тим більш важливо, оскільки руйнування об'єктів звичайно здійснюють не явним викликом деструктора класу, а викликом методу `Free`, який, у свою чергу, викликає віртуальний деструктор `Destroy`.

Якщо викликати деструктор для руйнування того екземпляра класу, який з якихось причин не був створений за допомогою конструктора, то

генеруватиметься помилка часу виконання. Тому перед викликом деструктора потрібно переконатися в тому, що екземпляр класу раніше був створений. Щоб спеціально не перевіряти наявність у пам'яті об'єкта, що руйнується, рекомендується не безпосередньо звертатися до деструктора, а викликати успадкований від класу `TObject` метод `Free`:

```
procedure Free;
```

Цей метод спочатку перевіряє те, що об'єкт, який руйнується, створювався і є в пам'яті, після чого викликає деструктор `Destroy` тільки у випадку позитивного результату перевірки.

Якщо екземпляр класу не створювався конструктором, то в ньому повинне зберігатися значення **`nil`**. Тому після руйнування об'єкта рекомендується заносити це значення у вказівник, який є екземпляром класу, чого не робить ні метод `Destroy`, ні метод `Free`. Це можна зробити за допомогою звичайного оператора присвоювання. У той же час у модулі `SysUtils` визначена процедура `FreeAndNil`, що має формат

```
procedure FreeAndNil(екземпляр_класу).
```

Дана процедура знищує об'єкт, зазначений як її параметр, і заносить у вказівник цього об'єкта значення **`nil`**.

Використання віртуальних і динамічних методів забезпечує можливість написання гнучких програм, що настроюються на конкретні екземпляри класів різних рівнів спадкування. Так, якщо екземпляру прабатьківського класу «присвоїти» раніше створений екземпляр дочірнього класу (фактично при цьому здійснюється присвоювання вказівника), то прабатько зможе оперувати методами нащадка, щоправда за умови, що ці методи є віртуальними або динамічними.

З урахуванням сказаного, можна дещо модифікувати програму, наведену як приклад 11.2, з усуненням описаної вище помилки й демонструванням можливості продовження ланцюжка спадкування, початого в іншому модулі.

```
//Приклад 11.3
//Створити таку ієрархію спадкування класів:
//рівень 0 – клас "точка на площині", що має
//наступні методи: введення координат, виведення координат
//з попереднім повідомленням; рівень 1 – клас
//"відрізок на площині" з методами введення координат
//кінців відрізка, обчислення довжини відрізка, виведення
//результату обчислення; рівень 2 – клас "опуклий
//багатокутник на площині" з методами введення кількості
//вершин та їх координат й обчислення площі багатокутника
//як суми площ трикутників, що його утворюють,
```

```
//з обчисленням площі трикутника за формулою Герона.
//Описати клас верхнього рівня в окремому модулі.
```

Насамперед створимо модуль Base, призначений для оперування розглянутим у прикладі 11.2 класом «точка на площині», дописавши в оголошенні класу `TPointClass` після заголовка методу `Write` службове слово **`virtual`**. Збережемо цей модуль у файлі BASE.PAS, що містить такий текст:

```
unit Base;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Forms,
  Graphics, Controls, Dialogs, StdCtrls, Buttons, ExtCtrls;

type
  TPoint = record
    X, Y: Real;
  end;

  TPointClass = class
    Point: TPoint;
    Mess: string;
    constructor Create(P: TPoint);
    procedure Write(M: string; Memo: TMemo); virtual;
  end;

implementation

constructor TPointClass.Create;
begin
  inherited Create;
  Point := P;     //Те ж саме, що і Point.X:=P.X;Point.Y:=P.Y;
end;

procedure TPointClass.Write;
begin
  Memo.Lines.Add(M);
  with Point do begin
    Memo.Lines.Add('X = ' + FloatToStr(X));
    Memo.Lines.Add('Y = ' + FloatToStr(Y));
  end;
end;

end.
```

Тепер дещо модифікуємо форму з прикладу 11.2, помістивши на неї ще одну кнопку введення, наклавши її на вже існуючу кнопку. Присвоїмо новій кнопці ім'я Button2, а в її властивість Caption запишемо значення Виконати. У властивість же Caption кнопки Button1 запишемо значення Введення.

Внесемо зміни в автоматично створений текст модуля форми. Насамперед у кінець списку модулів, що підключаються оператором **uses** секції **interface** модуля форми, допишемо ім'я додатково підключуваного модуля Base. Крім того, перед описом форми опишемо нові класи, які є спадкоємцями класу TPointClass, описаного в модулі Base:

```
type
  TLineSegment = class(TPointClass)
    PointF: TPoint;
    constructor Create(P1, P2: TPoint);
    function Calculate: Real; virtual;
    procedure Write(M: string; Memo: TMemo); override;
    function LengthSegment(P1, P2: TPoint): Real;
  end;

  TPolygon = class(TLineSegment)
    QuantityOfVertexes: Integer;
    Polygon: array of TPoint;
    constructor Create;
    function Calculate: Real; override;
    destructor Destroy; override;
  end;
```

У секції **private** опису форми, замість трьох полів Obj0, Obj1 та Obj2, як це було в прикладі 11.2, опишемо додаткове поле, яке має класовий тип TPointClass, що визначений у модулі Base:

```
Obj: TPointClass;
```

У секції **implementation** реалізуємо методи описаних вище класів TLineSegment і TPolygon. Оскільки тексти методів збігаються з аналогічними текстами з прикладу 11.2, обмежимося наведенням тільки тексту деструктора TPolygon.Destroy (тексти інших методів можна подивитися в прикладі 11.2):

```
destructor TPolygon.Destroy;
begin
                        //Завершення роботи з динамічним масивом
  Finalize(Polygon);
                              //Виклик успадкованого деструктора
  inherited Destroy;
end;
```

У порівнянні з прикладом 11.2, опрацьовувач події OnClick для компонента Button1 дещо зміниться, що зумовлене використанням тільки однієї змінної базового для даної задачі типу TPointClass:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  P1, P2: TPoint;
begin
  Button1.Hide;
  case cbChoise1.ItemIndex of
    1: begin
         Form1.Caption := 'Обчислення довжини відрізка';
         with P1 do begin
           X := StrToFloat(InputBox(
                       'Введення координат початку відрізка',
                       'Координата X', '0'));
           Y := StrToFloat(InputBox(
                       'Введення координат початку відрізка',
                       'Координата Y', '0'));
         end;
         with P2 do begin
           X := StrToFloat(InputBox(
                        'Введення координат кінця відрізка',
                        'Координата X', '0'));
           Y := StrToFloat(InputBox(
                        'Введення координат кінця відрізка',
                        'Координата Y', '0'));
         end;
         Obj := TLineSegment.Create(P1, P2);
         Obj.Mess := 'Довжина відрізка дорівнює ';
       end;
    2: begin
         Form1.Caption := 'Обчислення площі багатокутника';
         Obj := TPolygon.Create;
         Obj.Mess := 'Площа багатокутника дорівнює ';
       end;
    else begin
           Form1.Caption:='Виведення координати точки';
           with P1 do begin
            X:=StrToFloat(InputBox('Введення координат точки',
                        'Координата X точки ', '0'));
            Y:=StrToFloat(InputBox('Введення координат точки',
                        'Координата Y точки ', '0'));
           end;
           Obj := TPointClass.Create(P1);
           Obj.Mess := 'Координати точки:';
    end;
  end;
```

```
  Button2.Show;
  cbChoise1.Hide;
  lbOutput1.Hide;
end;
```

Для компонента Button2 створимо такий опрацьовувач події OnClick:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  with Obj do begin
    Write(Mess, Memo1);
    Destroy;
    Obj := nil;
  end;
  Button2.Hide;
end;
```

Розглянемо особливості наведеного варіанта розв'язання задачі.

У цьому випадку використовується тільки екземпляр Obj класового типу TPointClass, оголошеного в окремому модулі Base. Об'єкт Obj створюється залежно від обраного варіанта одним з конструкторів TPointClass.Create, TLineSegment.Create або TPolygon.Create. У зв'язку з тим, що віртуальними є як методи TPointClass.Write та TLineSegment.Write, так і методи TLineSegment.Calculate та TPolygon.Calculate, забезпечується настроювання об'єкта Obj і на методи свого породжуючого класу TPointClass, і на методи дочірніх класів TLineSegment й TPolygon. Якщо в процедурі TForm1.Button1Click об'єкт Obj створювався конструктором TPointClass.Create, то в процедурі TForm1.Button2Click викликається метод Write класу TPointClass. У випадку створення об'єкта Obj за допомогою конструктора TLineSegment.Create у процедурі TForm1.Button2Click викликається метод TLineSegment.Write, що звертається до методу Calculate класу TLineSegment. Якщо ж об'єкт Obj створюється конструктором Create класу TPolygon, то в процедурі TForm1.Button2Click здійснюється виклик методу Write, успадкованого класом TPolygon від класу TLineSegment, а цей метод звертається до методу Write класу TPolygon. Усе сказане можна відстежити, запустивши програму у налаштовувальному режимі (натискаючи клавіші F7 та F4), що рекомендується зробити, якщо необхідно розібратися в ході виконання програми.

Якщо в прикладі 11.2 ілюструвалася методика вкладеного виклику конструктора прабатьківського класу з конструктора дочірнього класу, то в

прикладі 11.3 проілюстрована методика виклику деструктора породжуючого класу з деструктора класу-нащадка.

У процедурі `TForm1.Button2Click` передбачено виклик деструктора для руйнування об'єкта `Obj`. Якщо об'єкт створювався одним з конструкторів `TPointClass.Create` або `TLineSegment.Create`, то в цьому випадку виконується виклик деструктора `Destroy`, успадкованого від спільного прабатька всіх класових типів `TObject`. У разі ж створення об'єкта `Obj` конструктором `TPolygon.Create` для руйнування об'єкта викликається деструктор `TPolygon.Destroy`, оскільки цей деструктор є віртуальним і перекриває стандартний деструктор (він описаний зі службовим словом **`override`**). Деструктор спочатку за допомогою процедури `Finalize` знищує динамічний масив `Polygon`, після чого звертається до успадкованого стандартного деструктора прабатька (**`inherited`** `Destroy`).

Наголосимо, що у наведеній процедурі `TForm1.Button2Click` з метою наближення до загальноприйнятої методики руйнування об'єктів виклик методу `Destroy` **слід** замінити викликом методу `Free`.

Оскільки, як це зазначалося вище, ні метод `Destroy`, ні метод `Free` не заносять значення **`nil`** в екземпляр класу, у прикладі 11.3 замість звертання до методу `Destroy` можна звернутися до методу `FreeAndNil`, виклик якого повинен виглядати так:

```
FreeAndNil(Obj);
```

11.4. Таблиця віртуальних методів

Якщо класовий тип містить або успадковує віртуальні методи, то для нього автоматично створюється так звана ***таблиця віртуальних методів*** – ТВМ (Virtual Method Table – VMT). У цій таблиці міститься розмір екземплярів класів, які будуть створюватися або знищуватися з використанням конструкторів і деструкторів. Крім того, у ній зберігаються посилання на віртуальні методи, доступні для даного класового типу (у тому числі й на успадковані). При цьому для кожного класового типу (але не для кожного екземпляра) створюється своя ТВМ, у яку потрапляють посилання на всі доступні віртуальні методи, навіть якщо відповідні посилання на ті ж самі методи існують у ТВМ предка (у випадку спадкування віртуальних методів).

Конструктор установлює зв'язок між створюваним екземпляром класу і ТВМ відповідного типу, забезпечуючи тим самим доступ екземпляра класу до його віртуальних методів. Тому при звертанні до віртуаль-

ного методу за його ім'ям завжди викликається той метод, який є характерним для класового типу, що відповідає екземпляру класу. Деструктор при його роботі розриває зв'язок між екземпляром класу та ТВМ.

У класичній об'єктовій моделі (при описі змінних за допомогою службового слова **`object`**), якщо об'єкт не ініціалізований конструктором, то для нього відсутній зв'язок із ТВМ, у результаті чого робота з таким об'єктом виявляється неможливою при першому ж виклику віртуального методу. Оскільки зв'язок із ТВМ установлюється для конкретного об'єкта, робота з неініціалізованим за допомогою конструктора об'єктом неможлива навіть у випадку, коли в нього переписана інформація з об'єкта, що пройшов ініціалізацію. В об'єктовій моделі, що базується на концепції класів, це тим більше неможливо, оскільки саме конструктор створює екземпляр класу, розміщаючи його в динамічній пам'яті.

11.5. Динамічні методи

У Delphi, крім віртуальних, введено ще так звані динамічні методи, які є підкласом віртуальних методів. Відмінність динамічних методів від віртуальних проявляється у способі їх виклику.

Щоб який-небудь метод оголосити динамічним, необхідно замість службового слова **`virtual`** вжити службове слово **`dynamic`**. Правила перевизначення динамічних методів збігаються з відповідними правилами для віртуальних методів.

У чому полягає перевага динамічних методів перед віртуальними? Як уже зазначалося, таблиці віртуальних методів створюються для всіх класових типів, які містять віртуальні методи. При цьому навіть якщо віртуальний метод не перевизначений, а успадкований, він усе одне потрапляє у ТВМ. Якщо в програмі використовується довгий ланцюжок класів з великою кількістю віртуальних методів, то буде витрачатися великий обсяг пам'яті для зберігання таблиць віртуальних методів. При застосуванні динамічних методів разом із ТВМ створюється ***таблиця динамічних методів*** – ТДМ (Dynamic Method Table – DMT). При цьому у ТВМ розміщаються віртуальні методи, а в ТДМ – динамічні, причому успадковані динамічні методи в ТДМ не потрапляють, а для їх пошуку використовується ТДМ класу-прабатька. Таким чином, застосовуючи динамічні методи, можна домогтися значної економії пам'яті (щоправда, за рахунок збільшення часу звертання до методів).

Як же взаємодіють таблиці віртуальних та динамічних методів, і в який спосіб здійснюється виклик динамічних методів у разі їх наявності у класів з декількома рівнями спадкування?

Як і раніше, у ТВМ потрапляють усі віртуальні методи. Але якщо класовий тип містить хоча б один динамічний метод, то додатково створюється ТДМ. При цьому у ТВМ заноситься посилання на ТДМ, що дозволяє відшукувати ТДМ класу. У ТДМ же потрапляють динамічні методи, визначені або перевизначені в даному класі. Якщо в класу є предок, для якого створена ТДМ, то в ТДМ класу-спадкоємця вміщується посилання на ТДМ клас-предка. Оскільки ТДМ предка досяжна із ТДМ спадкоємця, спадкоємцем може бути викликаний успадкований динамічний метод за допомогою ТДМ предка. При виклику динамічного методу через ТВМ класу відбувається звертання до ТДМ цього класу. Далі проводиться висхідний пошук динамічного методу у всіх ТДМ ланцюжка спадкування з викликом того з методів, який при такому висхідному пошуку буде знайдений першим.

11.6. Перевантажувані методи

У Delphi існує так званий механізм перевантаження методів аналогічний механізму перевантаження підпрограм, розглянутому в підрозд. 8.10. Цей механізм дозволяє оголошувати в рамках одного класу кілька однойменних методів. На відміну від використання методів нащадків, що перекривають однойменні методи прабатька й припускають забезпечення доступу до перекритих методів за допомогою службового слова **`inherited`**, у цьому випадку немає необхідності в спеціальних діях з організації доступу, за винятком методики опису ***перевантажуваних методів***.

Щоб компілятор міг розпізнати однойменні методи, потрібне виконання тих самих двох умов, що й у разі перевантаження звичайних підпрограм (див. підрозд. 8.10).

За наявності однойменних методів у класу компілятор видає попередження про цю ситуацію. Таке повідомлення може бути відключене, для чого метод, що перевантажує вже оголошений однойменний метод класу, описують із зарезервованим словом **`reintroduce`** (ввести повторно), яке розміщають перед службовим словом **`overload`** (з розділенням цих слів крапкою з комою).

11.7. Абстрактні методи та методи класу

Класи можуть містити так звані ***абстрактні методи*** – методи, які нічого не виконують. Ці методи застосовуються для опису стилю взаємодії з об’єктами і служать заготівкою для забезпечення поліморфізму. Їх в обов’язковому порядку потрібно перекривати в нащадках. Оскільки вони

не виконують жодних дій, можна заборонити їх виклик, для чого вони оголошуються з директивою **abstract**. Будучи перекриваними в обов'язковому порядку, вони, природно, повинні бути оголошені або як віртуальні, або як динамічні. При цьому компілятор не контролює звертання до неперекритого абстрактного методу.

Абстрактний метод має тільки оголошення, тобто інтерфейсну частину (заголовок, оголошений при описі класу). Оскільки ж реалізація абстрактного методу відсутня, звертання до нього спричиняє помилку часу виконання:

```
type
  TFirst = class
    procedure Proc;virtual; abstract;
  end;
  TSecond = class(TFirst)
    procedure Proc; virtual; override;
  end;
procedure TSecond.Proc;
begin
  //Виконувана частина процедури
end;
//.......................
var
  First : TFirst;
  Second: TSecond;
//.......................
begin
//.......................
  First.Proc;         //Цей метод абстрактний – помилка
  Second.Proc;  //Цей метод перекрив абстрактний метод
end;
```

У Delphi є так звані ***методи класу***, до яких можна звертатися без створення екземпляра класу. Метод класу описується звичайним способом, але перед словом **procedure** або **function** необхідно вказати службове слово **class**. Заголовок у визначенні методу класу також повинен починатися зі службового слова **class**. При звертанні до методу класу без попереднього створення екземпляра класу замість імені останнього вказують ім'я класу:

```
type
  Tclass = class
    class procedure Proc;
  end;
//.......................
```

```
class procedure TClass.Proc;
begin
  //Виконувана частина процедури
end;
  //.......................
begin
  TClass.Proc;    //Звернення без виклику конструктора
  //.......................
end;
```

Оскільки метод класу може бути викликаний без створення екземпляра класу, він не повинен звертатися до полів свого класу (при виклику такого методу в загальному випадку поля просто не існують).

11.8. Посилання на класи та вказівники методів

Поняття посилань на класи, можна було б розглянути в підрозд. 11.1, але оскільки це поняття багато в чому пов'язане з ідеями поліморфізму, ми розглянемо його саме тут.

Посилання на клас реалізують концепцію маніпулювання класами за допомогою програмного коду. Змінна, оголошена як посилання на клас, – це не сам вказівник, яким є об'єкт класового типу, а тільки посилання на тип об'єкта. Найчастіше посилання на клас використовуються для створення об'єктів, тип яких стає відомим тільки під час виконання програми.

Для опису типу «посилання на клас» застосовується конструкція **class of**:

```
ім'я_типу_посилання = class of ім'я_класу;
```

При цьому, природно, раніше має бути визначене ім'я_класу.

Змінна типу «посилання на клас» оголошується звичайним способом:

```
var
  посилання_на_клас : ім'я_типу_посилання;
```

Значенням такого роду змінної є тип, що визначається на етапі виконання, тобто або ім'я_класу, що вживалося в описі типу посилання, або ім'я будь-якого похідного від нього класового типу. Посилання на клас можна застосовувати в будь-якому виразі, де використання типу, на який посилається дане посилання, є законним. При цьому сумісність типів для посилань на класи ідентична аналогічній сумісності типів для класів.

Наприклад, для визначених у підрозд. 11.7 класів `TFirst` і `TSecond` можна описати такі типи посилань на класи:

```
type
  TRefFirst = class of TFirst;//Посилання на прабатька
  TRefSecond = class of TSecond;//Посилання на нащадка
```

Можна також описати такі змінні:

```
var
  RefFirst: TRefFirst;
  RefSecond: TRefSecond;
```

Допустимими будуть оператори

```
RefFirst := TRefFirst;
RefSecond := TRefSecond;
RefFirst := TRefSecond;
RefFirst := RefSecond;
```

У той же час наступні оператори є недопустимими, оскільки посилання на нащадка не може посилатися на прабатьківський клас:

```
RefSecond := TRefFirst;
RefSecond := RefFirst;
```

Приклад, що наводиться нижче, ілюструє використання посилань на класи, що містять віртуальні методи (у тому числі віртуальні конструктори).

```
//Приклад 11.4
//Приклад 11.2 з використанням посилань на класи.
```

Скористаємося формою з прикладу 11.2, замінивши код модуля таким кодом:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, ExtCtrls;

type
  TPoint = record
    X, Y: Real;
  end;

  TPointClass = class
    Point: TPoint;
    Mess: string;
    constructor Create; virtual;
    procedure Init; virtual;
    procedure Write(M: string; Memo: TMemo); virtual;
  end;
```

```
              //Оголошення типу "посилання на клас TPointClass"
  TPointClassRef = class of TPointClass;

  TLineSegment = class(TPointClass)
    PointF: TPoint;
    procedure Init; override;
    function Calculate: Real; virtual;
    procedure Write(M: string; Memo: TMemo); override;
    function LengthSegment(P1, P2: TPoint): Real;
  end;

  TPolygon = class(TLineSegment)
    QuantityOfVertexes: Integer;
    Polygon: array of TPoint;
    constructor Create; override;
    function Calculate: Real; override;
    destructor Destroy; override;
  end;

  TForm1 = class(TForm)
    Button1: TButton;
    Panel1: TPanel;
    BitBtn1: TBitBtn;
    Memo1: TMemo;
    Button2: TButton;
    cbChoise1: TComboBox;
    lbOutput1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
    Obj: TPointClass;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

constructor TPointClass.Create;
begin
                            //Виклик успадкованого конструктора
  inherited Create;
  Init;
end;
```

```
procedure TPointClass.Init;
begin
  with Point do begin
    X := StrToFloat(InputBox(
                            'Введення координат точки',
                            'Координата X', '0'));
    Y := StrToFloat(InputBox(
                            'Введення координат точки',
                            'Координата Y', '0'));
  end;
  Mess := 'Координати точки:';
end;

procedure TPointClass.Write;
begin
  Memo.Lines.Add(M);
  with Point do
  begin
    Memo.Lines.Add(' x= ' + FloatToStr(x));
    Memo.Lines.Add(' y= ' + FloatToStr(y));
  end;
end;

procedure TLineSegment.Init;
begin
  with Point do begin
    X := StrToFloat(InputBox(
                   'Введення координат початку відрізка',
                   'Координата X', '0'));
    Y := StrToFloat(InputBox(
                   'Введення координат початку відрізка',
                   'Координата Y', '0'));
  end;
  with PointF do begin
    X := StrToFloat(InputBox(
                   'Введення координат кінця відрізка',
                   'Координата X', '0'));
    Y := StrToFloat(InputBox(
                   'Введення координат кінця відрізка',
                   'Координата Y', '0'));
  end;
  Mess := 'Довжина відрізка дорівнює ';
end;

function TLineSegment.Calculate;
begin
  Result := LengthSegment(Point, PointF);
end;
```

```
procedure TLineSegment.Write;
begin
  Memo.Lines.Add(M + FloatToStr(Calculate));
end;

function TLineSegment.LengthSegment;
begin
  with P1 do
    Result := Sqrt(Sqr(x - P2.x) + Sqr(y - P2.y));
end;

constructor TPolygon.Create;
var
  i: Integer;
begin
  QuantityOfVertexes := StrToInt(InputBox(
         'Введення кількості вершин багатокутника',
         'Уведіть кількість вершин багатокутника', '1'));
  SetLength(Polygon, QuantityOfVertexes);
  for i := 0 to QuantityOfVertexes - 1 do
    with Polygon[i] do
    begin
     x := StrToFloat(InputBox('Введення вершин багатокутника',
            'Координата X вершини ' + IntToStr(i + 1), '0'));
     y := StrToFloat(InputBox('Введення вершин багатокутника',
            'Координата Y вершини ' + IntToStr(i + 1), '0'));
    end;
  Mess := 'Площа багатокутника дорівнює ';
end;

function TPolygon.Calculate;
var
  i: Integer;
  Len1, Len2, Len3, L: Real;
begin
  Result := 0;
  for i := 0 to QuantityOfVertexes - 3 do
  begin
    Len1 := LengthSegment(Polygon[i], Polygon[i + 1]);
    Len2 := LengthSegment(Polygon[i], Polygon[i + 2]);
    Len3 := LengthSegment(Polygon[i + 1], Polygon[i + 2]);
                        //Обчислюємо напівпериметр трикутника L
      L := (Len1 + Len2 + Len3) / 2;
                              //Підсумовування площ трикутників
    Result := Result + Sqrt(L * (L - Len1) *
                            (L - Len2) * (L - Len3));
 end;
end;
```

```
destructor TPolygon.Destroy;
begin
  Finalize(Polygon);  //Завершення роботи з динамічним масивом
  inherited Destroy;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Ref: TPointClassRef;            //Змінна – посилання на клас
begin
  Button1.Hide;
  case cbChoise1.ItemIndex of
    1: begin
         Form1.Caption := 'Обчислення довжини відрізка';
         Ref := TLineSegment;          //Присвоювання значення
                                           //посиланню на клас
       end;
    2: begin
         Form1.Caption := 'Обчислення площі багатокутника';
         Ref := TPolygon;
       end;
    else begin
           Form1.Caption := 'Виведення координати точки';
           Ref := TPointClass;
    end;
  end;
      //Реалізація поліморфізму за допомогою посилання на клас
  Obj := Ref.Create;
  Button2.Show;
  cbChoise1.Hide;
  lbOutput1.Hide;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  with Obj do
    Write(Mess, Memo1);
  FreeAndNil(Obj);
  Button2.Hide;
end;

end.
```

Зупинимося на особливостях наведеного вище модуля.

У класі `TPointClass` описано віртуальний конструктор `Create`, що перекрив конструктор класу `TObject`. При його виклику конструктор `TPointClass.Create`, крім створення об’єкта, звертається до віртуального методу `Init`, що забезпечує введення координат точки. Оскільки

описаний у класі `TPointClass` віртуальний метод `Init` перекривається однойменним віртуальним методом у класі `TLineSegment`, а метод `Create` успадковується класом `TLineSegment`, то при створенні екземпляра класу `TLineSegment` метод `Create` викликає не `TPointClass.Init`, а `TLineSegment.Init`, забезпечуючи введення координат кінців відрізка. Конструктор `Create` класу `TPolygon` також описано як віртуальний конструктор, що перекриває конструктор `Create` свого прабатька. При цьому ініціалізація екземпляра класу здійснюється самим конструктором без використання додаткового методу.

За рахунок того, що конструктори всіх класів у ланцюжку спадкування є віртуальними, обґрунтованим є бажання створювати об'єкт так, щоб у процесі свого створення він деяким чином сам би визначав, який тип він матиме. Саме з цією метою визначено тип `TPointClassRef`, що є посиланням на клас `TPointClass`. «Значенням» змінної цього типу може бути один з типів – `TPointClass`, `TLineSegment`, `TPolygon`.

Відповідно до властивостей посилань на клас виконано корекцію процедури `TForm1.Button1Click`. У ній оголошена змінна `Ref`, що є посиланням на клас й одержує своє значення на етапі виконання програми. Залежно від того, на який тип посилається змінна `Ref`, оператор `Obj := Ref.Create` викликає потрібний конструктор і створює відповідний клас.

В іншому робота даної програми нічим не відрізняється від варіантів розв'язання задачі в прикладах 11.2 і 11.3. Можна відзначити хіба що удавану відсутність використання деструкторів, у тому числі спеціально описаного в програмі деструктора `TPolygon.Destroy`. Замість явного виклику деструктора в програмі здійснюється звертання до методу `FreeAndNil`, що викликає потрібний деструктор залежно від того, з яким об'єктом працювала програма.

Раніше в підрозд. 8.7 було розглянуто такий елемент Delphi, як процедурний тип. Технічно змінна процедурного типу являє собою вказівник, що зберігає адресу підпрограми. Аналогічним за призначенням є тип що, називається вказівником методу (вказівником на метод). ***Вказівники методів*** можуть указувати не на автономні підпрограми, а **тільки** на методи об'єкта, причому не обов'язково того ж самого класу.

Формат оголошення типу «вказівник методу» аналогічний формату оголошення процедурного типу, за винятком того, що наприкінці визначення типу дописується пара слів **`of object`**:

```
type
  ім'я_типу = procedure (специфікація_параметрів) of object;
```

Наприклад:

```
type
  IntMethod = procedure (N: Integer) of object;
```

У дійсності вказівник методу – це пара вказівників, перший з яких є адресою методу, а другий зберігає посилання на об'єкт, якому належить метод.

11.9. Властивості

Властивості, подібно полям, визначають характеристики екземплярів класів. Але в той час як поле – це просто ділянка пам'яті, вміст якої може змінюватися, властивість зв'язує певні дії із читанням даних, записаних у ній, або їх зміною. При цьому властивості забезпечують керування доступом до полів.

Оголошення властивості виконується службовим словом **property**, після якого ідуть ім'я та тип властивості, а також специфікатори:

```
property ім'я_властивості: тип специфікатори;
```

Тут

- ім'я_властивості – будь-який припустимий ідентифікатор;
- тип – визначений у мові або попередньо оголошений тип даних; наприклад, неправильним є опис властивості вигляду

```
property Num: 0..9;
```

- специфікатори – у найпростішому випадку – це так звані *специфікатори доступу*, визначені службовими словами **read** і **write**, слідом за кожним з яких подається ім'я поля або методу.

Зазвичай (але не завжди) властивість зв'язується з деяким полем, а специфікатори доступу вказують, які методи використовуються при читанні із зазначеного поля або при записі в нього значення. Методи доступу, крім безпосередньо читання або запису значень, забезпечують виконання деяких додаткових специфічних дій.

У контексті програми властивість може сприйматися як звичайне поле, але при звертанні до неї автоматично викликається метод доступу.

На відміну від полів, властивості не можна передавати як **var**-параметри. Крім того, до властивості не може бути застосована операція `@`. Це пояснюється тим, що властивість не обов'язково існує в пам'яті.

Якщо на властивість посилаються у виразі, його значення читається з використанням поля або методу, внесеного в список специфікатора **read**. Якщо ж на властивість посилаються в лівій частині оператора присвою-

вання (або в іншому операторі, що змінює значення змінної), то в нього записується нове значення із застосуванням поля або методу, внесеного в список у специфікаторі **write**.

Якщо ніяких додаткових дій при звертанні до властивості не передбачається, то замість імені методу після специфікатора доступу наводиться ім'я поля, з яким зв'язана властивість.

Таким чином, кожна властивість має або специфікатор **read**, або специфікатор **write**, або обидва ці специфікатори. Вони мають такий формат:

```
read поле_або_метод
```

або

```
write поле_або_метод
```

де поле_або_метод – ім'я поля чи методу, оголошеного в тому ж класі, у якому оголошена властивість, або в класі предка.

При цьому існують деякі обмеження:

- якщо елемент поле_або_метод оголошений у тому ж самому класі, це має відбутися перед оголошенням властивості;
- якщо елемент поле_або_метод оголошений у класі предка, він повинен бути видимим для нащадка, тобто цей елемент не може бути полем або методом секції **private** класу предка, оголошеного в іншому модулі (див. наступний підрозділ);
- якщо поле_або_метод – поле, то воно повинне мати той самий тип, що й властивість;
- у специфікаторі **read**, якщо поле_або_метод – метод, то він повинен бути функцією без параметрів, тип результату якої збігається з типом властивості;
- у специфікаторі **write**, якщо поле_або_метод – метод, то він повинен бути процедурою, що має скалярний параметр-значення або **const**-параметр того ж типу, що й тип властивості (**за допомогою цього параметра усередину процедури передається записуване значення**);
- якщо поле_або_метод – метод, то він не може бути перевантажений.

Властивість, оголошення якої включає тільки специфікатор **read**, є властивістю «тільки для читання», а властивість, оголошення якої містить тільки специфікатор **write**, є властивістю «тільки для запису». Спроба запису значення у властивість «тільки для читання» або читання значення з властивості «тільки для запису» є помилковими.

Існують *угоди* відносно іменування властивостей, методів доступу і полів. Ці угоди зводяться до такого:

- властивість повинна мати цілком зрозуміле ім'я;
- при використанні **private**-поля для зберігання значення властивості потрібно давати цьому полю ім'я, яке починається з великої літери `F` (від *field*) і продовжується ім'ям властивості (наприклад, якщо властивість має ім'я `Radius`, то відповідному полю треба дати ім'я `FRadius`);
- процедурі, що змінює значення властивості, треба давати ім'я, що починається зі слова `Set`, за яким подається ім'я властивості (наприклад, `SetRadius`);
- ім'я функції для читання властивості повинне починатися зі слова `Get` (наприклад, `GetRadius`).

Зазначені угоди не є обов'язковими до їхнього виконання, але в багатьох випадках відхилення від них може призвести до негативних наслідків, тим більш, що сприйняття тексту програми, написаної з використанням угод відносно іменування, значно полегшується.

Розглянемо, наприклад, такий клас:

```
TCirc = class
  private
    FRadius: Real;
    FX, FY: Real;
    function GetRadius: Real;
    procedure SetRadius(R: Real);
  public
    Square: Real;
  published
    property Radius: Real read GetRadius
                          write SetRadius;
    property X: Real read FX write FX;
    property Y: Real read FY write FY;
    function Test: Boolean;
end;
```

Клас `TCirc` містить три дійсні поля: `FRadius`, `FX`, `FY`, які, будучи оголошеними в секції **private**, є недоступними спадкоємцям при описі останніх в іншому модулі.

Для доступу до поля `FRadius` у секції **published** оголошена властивість `Radius`, яка має методами доступу `GetRadius` й `SetRadius`. Функція `TCirc.GetRadius` призначається для читання значення, що зберігається в полі `FRadius` із забезпеченням доступу до нього за допо-

могою властивості Radius. Процедура ж TCirc.SetRadius призначається для запису в поле FRadius значення за допомогою властивості Radius. При цьому метод TCirc.SetRadius може бути оголошений або так, як це зроблено вище, або в такий спосіб:

```
procedure SetRadius(const R: Real);
```

Властивості X та Y за допомогою специфікаторів доступу **read** і **write** забезпечують опосередкований доступ до полів FX і FY без ужи-вання спеціальних методів. При цьому дія специфікаторів **read** властивостей X та Y полягає в сприйнятті значень, занесених у відповідні цим властивостям поля FX і FY, а специфікатори **write** забезпечують запис у зазначені поля значень властивостей X та Y.

З огляду на сказане, якщо в програмі є змінна Circ типу TCirc, то оператор

```
if (Circ.X < 0) and (Circ.Y < 0) and
   (Circ.Radius > 100) then Circ.Radius := 100;
```

заміняє оператор

```
if (Circ.FX < 0) and (Circ.FY > 0) and
   (Circ.GetRadius > 100) then Circ.SetRadius(100);
```

```
//Приклад 11.5
//Ввести радіус R кола й обчислити його площу.
//Чи належить точка, обидві координати якої
//є випадковими числами з інтервалу від -50 до 50,
//колу заданого радіуса із центром на початку координат?
```

Скористаємося формою з прикладу 2.2, записавши у властивість Caption мітки текст Уведіть радіус і натисніть "Виконати", а у властивість Caption кнопки Button1 текст Виконати.

У секцію **interface** модуля внесемо опис наведеного вище класу TCirc, а в секції **implementation** опишемо змінну Circ типу TCirc, а також такі методи:

```
function TCirc.GetRadius;                    //Функція описана як
begin          //приклад – можна було просто зв'язатися з полем
  Result := FRadius;
end;

procedure TCirc.SetRadius;                  //Запис у поле FRadius
begin                                        //з обчисленням площі
  FRadius := R;     //У FRadius записується параметр процедури
         //Буде дворазове звернення до функції TCirc.GetRadius
  Square := Pi * Radius * Radius;
  Form1.mmOutput1.Lines.Add('Площа круга радіуса ' +
```

```
                                        //Нижче ще одне звертання
     FloatToStr(Radius) + ' дорівнює ' + FloatToStr(Square));
end;

function TCirc.Test;
begin
  if Sqrt(X * X + Y * Y) <= Radius then Result := True
  else Result := False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  DecimalSeparator := '.';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Randomize;
  Circ := TCirc.Create;
  with Circ do begin
    Radius := StrToFloat(Trim(edInput1.Text));
    X := (Random - 0.5) * 100;      //Фактично запис у Circ.FX
    Y := (Random - 0.5) * 100;      //Фактично запис у Circ.FY
    if Test
      then mmOutput1.Lines.Add('Точка (' + FloatToStr(X) + ','
                    + FloatToStr(Y) + ')' + ' усередині кола')
      else mmOutput1.Lines.Add('Точка (' + FloatToStr(X) + ','
                    + FloatToStr(Y) + ')' + ' поза колом');
  end;
  edInput1.SetFocus;
end;
```

При кліку мишкою над кнопкою Button1 спрацьовує опрацьовувач події OnClick цієї кнопки, усередині якого відбувається звертання до властивостей екземпляра `Circ` класу `TCirc`. При записі введеного значення радіуса у властивість `Radius` здійснюється звертання до методу `TCirc.SetRadius`, що записує в поле `FRadius` введене значення, обчислює площу круга й виводить її у вікно багаторядкового редактора. При цьому тричі зчитується значення властивості `Radius` зі зверненням до методу `TCirc.GetRadius`, що у цьому випадку виконує чисто ілюстративну функцію, звертаючись до поля `FRadius` і не виконуючи жодних додаткових дій.

Як про це говорилося вище, метод `TCirc.SetRadius` є процедурою з одним параметром, за допомогою якого передається значення, записуване у властивість `Radius`. У той же час метод

TCirc.GetRadius є функцією без параметрів, тип якої збігається з типом властивості Radius.

Звертання до властивостей X та Y для читання і запису забезпечує доступ до **private**-полів FX і FY.

У програмі властивість поводиться як звичайне поле. У наведених вище текстах підпрограм прикладами такого звертання є оператори

```
Square := Pi * Radius * Radius;
```

та

```
X := (Random - 0.5) * 100;
```

Однак особливістю властивостей є те, що їх не можна підставляти замість **var**-параметрів підпрограм.

Деякі із властивостей можуть бути ***властивостями-масивами***, тобто такими властивостями, які можуть індексуватися. Оголошення властивості-масиву містить список, що визначає імена та типи індексів, і має формат

property ім'я_властивості[індекси]: тип специфікатори;

Формат списку параметра індекси такий самий, як у списку параметрів процедури або функції, за винятком того, що тут параметри оголошуються у квадратних дужках замість круглих. Таким чином, індекси – це послідовність описів параметрів, розділених крапкою з комою. Кожен опис параметра має вигляд

ідентифікатор1,..., ідентифікаторN: тип

На відміну від масивів, які можуть використовувати тільки індекси порядкового типу, властивості-масиви допускають індекси будь-якого типу, наприклад:

```
property Objects[Index: Integer]: TObject read GetObj
                                            write SetObj;
property Pixels[X, Y: Integer]: TColor read GetPixel
                                          write SetPixel;
property Values[const Name: string]: string
                            read GetValue write SetValue;
```

Елементи властивості-масиву в певному розумінні є аналогами пунктів у списку, команд керування тощо

Для властивостей-масивів специфікатори доступу повинні посилатися тільки на методи, а не на поля. У цьому випадку методом у специфікаторі **read** має бути функція, кількість і типи параметрів якої відповідають кількості й типам параметрів, перелічених у специфікаторі індекси властивості. Тип результату цієї функції має збігатися з типом властивості.

У специфікаторі **write** методом може бути тільки процедура з кількістю та типами параметрів, що збігаються з кількістю та типами параметрів у списку параметра індекс властивості. Додатково наприкінці списку параметрів цього методу слід розмістити параметр (можливо, константний) того ж типу, що і властивість.

Наприклад, методи доступу для описаних вище властивостей-масивів можуть бути оголошені так:

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

До властивості-масиву звертаються, індексуючи ім'я властивості. Наприклад, оператори

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

аналогічні наступним:

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

Крім специфікаторів **read** і **write**, опис властивості відразу ж за його типом може містити специфікатор **index**, що має такий формат:

index специфікація_цілочислової_константи

Специфікатор **index** служить для забезпечення зв'язку властивостей з елементами поля, що є масивом, дозволяючи декільком властивостям розділяти метод доступу при звертанні до різних елементів поля-масиву. Специфікатор індексу складається з директиви **index**, що супроводжується цілочисловою константою від `-2147483647` до `2147483647`. Якщо властивість має специфікатор **index**, його специфікатори **read** і **write** повинні звертатися до методів, а не до полів.

Наприклад:

```
type
  TRectangle = class
  private
    FCoordinates: array[0..3] of LongInt;
    function GetCoordinate(Index: Integer): LongInt;
```

```
    procedure SetCoordinate(Index: Integer;
                            Value: LongInt);
  public
    property Left: LongInt index 0 read GetCoordinate
                                    write SetCoordinate;
    property Top: LongInt index 1 read GetCoordinate
                                    write SetCoordinate;
    property Right: LongInt index 2 read GetCoordinate
                                    write SetCoordinate;
    property Bottom:LongInt index 3 read GetCoordinate
                                    write SetCoordinate;
    property Coordinates[Index: Integer]: LongInt
               read GetCoordinate write SetCoordinate;
    //...
  end;
```

Метод доступу для властивості зі специфікатором **index** повинен мати додатковий параметр типу `Integer`. Для **read**-функції таким є останній параметр, а для **write**-процедури цей параметр має бути передостаннім (тобто він повинен передувати параметру, що визначає значення властивості). При звертання програми до такої властивості зазначена після його специфікатора **index** ціла константа автоматично передається методу доступу за допомогою цього параметра.

Властивості можуть описуватися з додатковими директивами **stored**, **default** і **nodefault**, які називаються ***специфікаторами зберігання***. Ці директиви не впливають на поведінку програми і задаються зазвичай тільки для властивостей, описаних у секції **published** класу. Зокрема, вони визначають, чи буде значення властивості зберігатися у файлі форми.

За директивою **stored** розміщується або одне зі значень `True` чи `False`, або ім'я поля типу `Boolean`, або ім'я методу, що є функцією без параметрів, яка повертає значення типу `Boolean`. Якщо властивість не має специфікатора **stored**, це відповідає значенню `True`, заданому для **stored**. Якщо атрибут дорівнює `True` і значення властивості відрізняється від значення, зазначеного в атрибуті **default** (або якщо немає жодного **default**-значення), то значення властивості зберігається у файлі форми, у противному випадку – ні.

Директива **default** супроводжується константою того ж типу, що і властивість, і дозволяє вказати для властивості значення за умовчанням:

```
property Tag: LongInt read FTag write FTag default 0;
property Tag: LongInt read FTag write FTag default 0;
```

Щоб перекрити успадковане **default**-значення, не визначаючи нове, використовують директиву **nodefault**. Директиви **default** і **nodefault** підтримуються тільки для порядкових типів і для множин з діапазоном порядкових значень базового типу від `0` до `31`. Якщо така властивість оголошена без **default** або **nodefault**, то це відповідає директиві **nodefault**. Для цілих типів, вказівників і множин **default**-значенням за умовчанням є відповідно `0`, **nil** й `[]` (порожня множина).

Значення за умовчанням, задане директивою **default**, стосується тільки значення властивості, відображуваної Інспектором Об'єктів, і не присвоюється властивості автоматично при запуску програми. Його необхідно явно присвоїти в конструкторі компонента.

Специфікатори зберігання не підтримуються для властивостей-масивів. Директива **default** має невизначене значення при її застосуванні в описі властивості-масиву або разом з директивою **implements**.

Розгляд специфікатора **implements** виходить за рамки матеріалу, що викладається.

У Delphi є особливий вид властивостей, що називаються подіями і які базуються на використанні вказівників методів. Оскільки події, як елемент Delphi, більшою мірою стосуються компонентів, то цей елемент буде розглядатися дещо пізніше.

11.10. Секції класів

При проектуванні форми ми вже стикалися з секціями, які можуть використовуватися в класах. Більш того, порожні секції **private** і **public** вставляються в клас форми автоматично.

Призначення секцій класів – визначити область видимості полів, методів і властивостей класу.

Класи в Delphi, крім секції, що не має назви і розташовується відразу ж за заголовком класу, можуть містити ще 5 видів секцій, кожна з яких відкривається своїм службовим словом:

- **public** (*загальнодоступні*) – оголошені в цій області складові класу мають необмежену область доступу, тобто до них можна звертатися в будь-якому модулі програми;
- **published** (*опубліковані*, *декларовані*) – аналогічна секції **public**, але оголошені в цій секції властивості доступні і на етапі конструювання форми у вікні Інспектора Об'єктів;
- **private** (*особисті*) – елементи, описані в цій секції, відомі тільки методам даного класу і підпрограмам, описаним у модулі, в

якому оголошено даний клас, але не відомі нащадкам класу, оголошеним в іншому модулі;

- **protected** (*захищені*) – елементи, описані в цій секції, відомі як усередині даного класу, так і у його нащадках, навіть якщо останні описані в інших модулях;
- **automated** (*автоматизовані*) – область видимості елементів даної секції необмежена, але ця секція вживається тільки для оголошення методів і властивостей при використанні інтерфейсу OLE- автоматизації.

Розташована на початку опису класу секція, що не має назви, є **published**-секцією, у якій ICP Delphi поміщає опис компонентів форми. Вручну цю секцію коректувати не рекомендується. У секції **published** не можна оголошувати властивості-масиви.

Видимість властивості може бути підвищена в нащадка базового класу. Для цього воно повинне бути оголошене повторно. При цьому існують такі обмеження:

- особиста властивість базового класу може бути перенесена в одну з секцій **protected**, **public** або **published**;
- якщо властивість базового класу є захищеною, у похідному класі її можна зробити відкритою або опублікованою;
- відкриту властивість прабатька в похідному класі можна зробити опублікованою;
- переміщення опублікованих властивостей в інші секції неможливе;
- зниження рівня видимості властивості заборонене.

Особливістю захищених елементів класу є те, що вони, будучи відомими навіть нащадкам класу, оголошеним в інших модулях, є доступними тільки програмному коду, розташованому в тому модулі, в якому описано базовий клас. Тому програмний код, розміщений в іншому модулі, не може звернутися до захищених елементів прабатьківського класу. З метою усунення цього протиріччя для забезпечення доступу до захищених елементів класу в інших модулях варто створювати в цих модулях класи-нащадки, переносячи в них захищені елементи в секцію **published** (іноді в секцію **public**), що реалізовано в компонентних класах Delphi. Інша методика забезпечення доступу до захищених полів даного класу, яку часто називають *хакерською*, описана в п. 15.3.4.

Запитання для контролю і самоконтролю

1. На яких основних поняттях базується об’єктно-орієнтоване програмування? Наведіть їх визначення.

2. Що таке клас?
3. Подайте визначення понять «поле», «метод», «властивість».
4. Що таке екземпляри класів, як вони створюються і в якій області пам'яті розміщаються?
5. Які обмеження накладаються на опис класів?
6. Що таке «похідний класовий тип» і як він визначається?
7. Що таке спадкування і перевизначення методів?
8. Як забезпечується спадкування і перевизначення?
9. Чи можна перевизначити поле?
10. Як здійснюється ідентифікації типу екземпляра класу?
11. Як можна виконати перетворення типу екземпляра класу?
12. У чому відмінність віртуальних і статичних методів?
13. Для чого служать конструктори і деструктори? Чи можуть вони бути віртуальними?
14. Для чого служить таблиця віртуальних методів?
15. Скільки таблиць віртуальних методів створюється для одного класу за наявності декількох його екземплярів?
16. Чим відрізняються віртуальні методи від динамічних?
17. Що таке абстрактний метод?
18. Що таке метод класу?
19. Що таке «посилання на клас»?
20. Як оголошується властивість?
21. Як властивість взаємодіє з полями?
22. Яке призначення специфікаторів **`read`** і **`write`** в оголошенні властивості?
23. Назвіть та охарактеризуйте секції класів.

Завдання для практичного відпрацьовування матеріалу

1. Створити об'єктно-орієнтовану програму з двома рівнями спадкування: клас верхнього рівня служить для обчислення визначника матриці третього порядку, а клас-спадкоємець – для обчислення зворотної матриці для матриці третього порядку.
2. Визначити клас `TRectangle` для обробки даних про прямокутники, задані на координатній площині координатами центра (x, y), висотою h і шириною w. Конструктор, створюючи об'єкт-прямокутник, повинен задати його параметри. У класу повинен бути метод, що реалізує обчислення периметра. Реалізувати програму, яка створює та ініціалізує масив прямокутників і виводить їхні периметри.

3. Модифікувати клас `TRectangle` у такий спосіб. Параметри прямокутника повинні визначатися **`private`**-полями, яким поставлені у відповідність властивості, що задають і зчитують значення відповідних їм полів за допомогою визначених у класі методів доступу. Реалізувати програму, що створює та ініціалізує масив прямокутників і виводить їхні периметри.
4. Модифікувати попередню програму так, щоб після введення всіх даних за командою від кнопки здійснювався зсув всіх прямокутників по вертикалі та горизонталі таким чином, аби лівий верхній кут прямокутника з найменшим периметром розташувався на початку координат.
5. Нехай натуральне число задається послідовністю цифр, записаною в числовому масиві або рядку символів. Описати клас, що визначає таке число і містить методи піднесення в степінь, обчислення факторіала і перевірки на простоту. Як спадкоємця визначити клас для роботи із двома такими числами, який містить методи, що реалізують обчислення їх суми, різниці, добутку, найбільшого спільного дільника і найменшого спільного кратного. Ще один спадкоємець повинен забезпечувати аналогічну роботу з послідовністю з n таких чисел, де n – натуральне число.
6. Розв’язати задачу, сформульовану в завданні 6 до розд. 20 (побудову рисунків можна замінити виведенням різних повідомлень).

12. СПИСКИ

У Delphi є декілька класів, визначених у модулі `Classes`, які служать для створення наборів даних, що у певному розумінні аналогічні динамічним масивам. Такі набори даних називаються списками (не плутати зі зв'язаними списками, що розглядалися в підрозд. 7.4). Мова в цьому випадку, насамперед, іде про класи `TList`, `TStrings` і `TStringList`.

12.1. Списки класу TList

Клас `TList` забезпечує можливість створення індексованої послідовності елементів, які дозволяють організувати доступ до даних довільного типу. Екземпляр класу `TList` є масивом нетипізованих вказівників (типу `Pointer`). Звичайно, коли говорять про *списки* без уточнення їхніх властивостей, мають на увазі екземпляри класу `TList`. Будучи індексованими й указуючи на розміщені в динамічній пам'яті елементи, ці вказівники дозволяють здійснювати індексований доступ до таких елементів. Оскільки ж нетипізований вказівник узгоджений за присвоюванням з вказівниками будь-якого типу, подібний набір даних дає можливість поєднувати дані різного типу, на відміну від масивів, які за визначенням служать для зберігання даних одного й того самого типу. Розмір списку обмежується доступною пам'яттю, хоча, взагалі кажучи, індекс списку може набувати значень тільки з діапазону `0..MaxListSize`, де `MaxListSize` – системна константа зі значенням `134217727` (вона визначена в модулі `Classes` так: `MaxListSize = MaxInt div 16`).

Властивості та методи класу `TList` дозволяють:

- додавати й видаляти елементи в списку;
- відшукувати елементи в списку;
- перетворювати список.

Екземпляр класу `TList` створюється звичайним способом – за допомогою конструктора (але не за допомогою процедури `SetLength`, як це має місце у випадку динамічних масивів).

Клас `TList` має такі властивості:

- **property** Capacity: Integer– поточна ємність списку;
- **property** Count: Integer – кількість елементів, що фактично містяться в списку;
- **property** Items[Index: Integer]: Pointer – вказівник на елемент списку з індексом Index;
- **property** List: PPointerList – вказівник на масив елементів списку.

При створенні списку властивості Capacity та Count ініціалізуються нулем. Надалі за необхідності поповнення вже повністю заповненого списку значення Capacity автоматично збільшується: якщо Capacity<6, – на 4, якщо 5<Capacity<9 – на 8, якщо 8<Capacity – на 16.

При додаванні (знищенні) елементів списку автоматично збільшується (зменшується) на 1 значення Count; якщо Count=Capacity і необхідно додати новий елемент, попередньо збільшується значення Capacity і список розширюється до нового обсягу.

За значенням Items[Index] забезпечується доступ до елементів списку. При цьому індекс Index може набувати значень з діапазону від 0 до Count-1. Самі елементи списку зберігаються в масиві, доступ до якого здійснюється за допомогою властивості List, що має тип PPointerList, визначений у модулі Classes у такий спосіб:

```
type
  PPointerList = ^TPointerList;
  TPointerList = array[0..MaxListSize - 1] of Pointer;
```

Екземпляр класу TList створюється за допомогою конструктора Create, наслідуваного класом TList від класу TObject – його безпосереднього прабатька. Оскільки елементами списку є вказівники типу Pointer, їх можна зв’язати з елементами даних будь-якого типу. Природно, при цьому необхідно якимсь чином забезпечити розпізнавання фактичного типу даних. Тому звичайно список зв’язують із даними, що мають один і той самий тип. Самі дані в динамічній пам’яті автоматично не розміщаються, – програміст повинен забезпечити їх розподіл і записати адресу їх розміщення як елементу списку.

Видаляють список за допомогою успадкованого від класу TObject методу Free, хоча можна використовувати і власний деструктор Destroy, що викликає метод Free для видалення списку.

Клас має досить багато власних методів, більшість із яких наведено в таблиці 12.1.

Таблиця 12.1 – Власні методи класу `TList`

Метод	Опис
`function Add(Item: Pointer): Integer; virtual;`	Додає новий елемент у кінець списку, повертаючи його індекс із відліком від `0`
`procedure Assign(ListA: TList; AOperator: TListAssignOp = laCopy; ListB: TList = nil);`	Комбінує два списки `ListA` і `ListB` з використанням операції `AOperator`
`procedure Clear; virtual;`	Очищає список, записуючи в `Count` і `Capacity` значення `0`, але не очищаючи пам'ять, пов'язану з елементами списку
`procedure Delete(Index: Integer);`	Видаляє елемент з номером `Index`, зміщаючи елементи, що стоять за вилученим, і зменшуючи на `1` значення `Count`
`destructor Destroy; override;`	Викликає метод `Free` для знищення списку
`procedure Exchange(Index1, Index2: Integer);`	Міняє місцями елементи із зазначеними індексами
`function Expand: TList;`	Збільшує `Capacity` і розширює список
`function Extract(Item: Pointer): Pointer;`	Видаляє зазначений елемент, зменшуючи `Count` і зсуваючи елементи за вилученим
`function First: Pointer;`	Установлює вказівник на перший елемент списку
`procedure Insert(Index: Integer; Item: Pointer);`	Вставляє елемент `Item` за елементом з номером `Index`, збільшуючи на `1` значення `Count` і, якщо це потрібно, розширюючи список
`function Last: Pointer;`	Установлює вказівник на останній елемент списку
`procedure Move(CurIndex, NewIndex: Integer);`	Переміщає елемент із номером `CurIndex` у позицію `NewIndex`
`procedure Pack;`	Видаляє порожні елементи списку, зменшуючи при цьому значення `Count`
`function Remove(Item: Pointer): Integer;`	Видаляє порожні елементи в списку та зменшує значення `Count`
`procedure Sort(Compare: TListSortCompare);`	Сортує список за допомогою функції `Compare` типу `TListSortCompare`

Підкреслимо, що жоден з методів видалення елементів списку не знищує автоматично дані, що адресуються елементом, який видаляється, – знищення даних має бути передбачене програмістом.

Особливістю методу `Sort` є те, що він виконує сортування елементів списку за правилом, яке визначається розробленою програмістом функцією

з двома параметрами типу Pointer, яка повертає значення типу Integer. При звертанні до методу Sort необхідно як його параметр зазначити ім'я цієї функції. Параметр методу Sort повинен мати тип TListSortCompare, визначений у такий спосіб:

```
TListSortCompare=function(Item1,Item2:Pointer):Integer;
```

Ця функція повинна порівнювати два значення, з якими зв'язані її параметри-вказівники, повертаючи одне з трьох значень:

- будь-яке від'ємне число, якщо перестановка непотрібна;
- 0 при Item1^ = Item2^;
- будь-яке додатне число, якщо значення Item1^ та Item2^ мають бути переставлені.

Процедура Sort розташовує елемент списку таким чином, щоб для будь-яких його двох елементів виконувалося співвідношення, визначене функцією, заданої як параметр звертання.

Метод Assign формує список, комбінуючи елементи списків, заданих як перший і третій параметри, з використанням логічної операції, що задається другим його параметром. Останній параметр методу (або два останніх параметри) може бути опущений.

Другий параметр методу Assign задається значенням такого переліченого типу:

```
type
  TListAssignOp = (laCopy, laAnd, laOr, laXor,
                   laSrcUnique, laDestUnique);
```

Ці значення визначають методику злиття двох списків (див. таблицю 12.2). За умовчанням другий параметр має значення laCopy.

Таблиця 12.2 – Операції копіювання списків

Значення	Результуючий список
laCopy	Копіювання елементів ListA в ListB
laAnd	Елементи списку ListA, що містяться в списку ListB
laOr	Елементи списку ListA, за якими ідуть елементи списку ListB, що не містяться в ListA
laXor	Елементи списку ListA, що не ввійшли в список ListB, за якими йдуть елементи ListB, що не містяться в ListA
laDestUnique	Елементи списку ListB, що не містяться в ListA (у зворотному порядку)
laSrcUnique	Елементи списку ListA, що не містяться в ListB

```
//Приклад 12.1
//Вводяться дійсні числа до першого недодатного значення.
```

```
//Видалити з уведеної послідовності те значення, що найбільш
//віддалене від середнього арифметичного всіх введених чисел.
//Упорядкувати значення, що залишилися, за зростанням відстані
//до останнього з цих чисел.
```

Створимо форму, розташувавши на ній у лівій частині багаторядкове редаговане текстове поле, а в правій частині – панель. Розмістимо на панелі три накладені одну на одну кнопки введення – Button1, Button2 і Button3 та змінимо деякі властивості створених компонентів:

- Панель:
  Align — alRight
  BevelOuter — bvNone
  Caption — очистити
- Багаторядкове поле:
  Align — alClient
  Lines — очистити
  Name — mmOutput1
  ScrollBars — ssBoth
- Кнопка Button1:
  Caption — Введення
- Кнопка Button2
  Caption — Видалення
  Visible — False
- Кнопка Button3:
  Caption — Сортування
  Visible — False

У розділі **implementation** визначимо такий тип:

```
type
  TPtR = ^Real;
```

Крім того, опишемо змінну `List` типу `TList` і змінну `Last` типу `Real`. Визначимо також наступні підпрограми, у тому числі опрацьовувачі події OnClick кнопок:

```
procedure Output(List: TList; Memo: TMemo; Msg: string);
var
  i: Integer;
begin
  Memo.Lines.Add(Msg);
  for i := 0 to List.Count - 1 do
    Memo.Lines.Add(FloatToStr(TPtR(List[i])^));
                          //Список вміщує вказівники типу TPtR
end;
```

```
function Compare(Item1, Item2: Pointer): Integer;
begin
  if Sqr(TPtR(Item1)^ - Last) < Sqr(TPtR(Item2)^ - Last)
  then Result := -1
  else
    if Sqr(TPtR(Item1)^ - Last) > Sqr(TPtR(Item2)^ - Last)
    then Result := 1
    else Result := 0;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  PtR: TPtR;
begin
  List := TList.Create;                    //Створення списку
  repeat
    new(PtR); //Відведення пам'яті під число, що вводитиметься
    PtR^ := StrToFloat(InputBox('Введення чисел',
                                'Уведіть число', '-1'));
    if PtR^ <= 0 then begin          //Умова закінчення введення
      Dispose (PtR);               //Видалення введеного значення
      Break;     //Примусовий вихід із циклу (кінець введення)
    end;
    List.Add(PtR);                  //Додавання елемента в список
  until False;                                //Нескінченний цикл
  if List.Count = 0 then Exit;
  Output(List, Memo1, 'Початковий список');
  Button1.Hide;
  Button2.Show;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  i, n: Integer;
  S, Max: Real;
  PtR: TPtR;
begin
  S := 0;
  for i := 0 to List.Count - 1 do
    S := S + TPtR(List[i])^;
  S := S / List.Count;
  n := 0;
  Max := Sqr(TPtR(List[0])^ - S);
  for i := 1 to List.Count - 1 do
    if Max < Sqr(TPtR(List[i])^ - S) then begin
      n := i;
      Max := Sqr(TPtR(List[i])^ - S);
    end;
```

```
  PtR := List[n];  //Беремо вказівник елемента, що видаляється
  Dispose(PtR);              //Спочатку видаляємо елемент даних
  List.Delete(n);              //Тепер видаляємо елемент списку
  Output(List, Memo1, 'Після видалення');
  Button2.Hide;
  if List.Count = 0 then begin
    Output(List, Memo1, 'список порожній');
    Exit;
  end;
  Button3.Show;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin          //Last – останнє значення, що записане в список
  Last := TPtR(List[List.Count - 1])^;    //Доступ до значення
  List.Sort(Compare);                      //Сортування списку
  Output(List, Memo1, 'Після сортування');
  Button3.Hide;
end;
```

Перший оператор процедури `TForm1.Button1Click` створює список `List`, після чого в циклі спочатку виділяється пам'ять під значення, що буду водитися, а далі після введення значення адреса його місця розташування заноситься в список за допомогою методу `Add`. При цьому, незважаючи на те, що введене значення адресується вказівником `Pt` типу `TPt`, турбуватися про узгодження типу немає необхідності, оскільки елементи списку мають тип `Pointer`. У той же час при обробці елементів даних, розміщених у списку, потрібно здійснювати приведення типу `Pointer` до типу `TPt`. Це ілюструє оператор `S:=S+TPt(List[i])^` у процедурі `TForm1.Button2Click` (у ньому доступ до значень, що розміщені у динамічній пам'яті, сполучений із приведенням типу).

У процедурі `TForm1.Button2Click` демонструється методика видалення елемента списку: елемент списку видаляється **після** того, як знищиться елемент даних, що адресується ним.

Варіант звертання до методу `Sort` сортування списку демонструється в підпрограмі `TForm1.Button3Click`. При цьому попередньо визначена власна функція порівняння елементів списку `Compare`, що передається методу сортування через його параметр.

12.2. Списки класу TStrings

Клас `TStrings` є абстрактним спадкоємцем класу `TPersistent`, який, у свою чергу, є безпосереднім спадкоємцем класу `TObject`. Цей

клас дозволяє створювати списки, елементами яких є пари типу рядок-об'єкт. При цьому елементи пари зв'язані один з одним у припущенні, що вони мають одне й те саме значення індексу, причому рядок – це рядок символів, а об'єкт – екземпляр (об'єкт) будь-якого класу. Деякі з методів даного класу орієнтовані на те, що рядок, який входить в елемент списку, може містити знак рівності, який розділяє його на дві частини – до і після нього. Саме в такому вигляді подаються різні рядки в конфігураційних файлах операційної системи. Таким чином, даний клас і його спадкоємці пристосовані для обробки службових текстових файлів операційної системи.

Екземпляри класу `TStrings` практично не використовуються, оскільки базові методи цього класу або самі є абстрактними (`Clear`, `Delete`, `Insert`), або викликають абстрактні методи (`Add`). Властивості `Count` і `Strings`, що є одними з основних властивостей цього класу, як свої специфікатори читання мають абстрактні методи `GetCount` (для `Count`) і `Get` (для `Strings`). Однак у цього класу є велика кількість спадкоємців, у яких абстрактні методи перекриті, що забезпечує їх активне застосування в програмах. Методи, що не є абстрактними, або використовуються спадкоємцями класу `TStrings` без їх перекриття, або також перекриваються.

Деякі з власних (не успадкованих) властивостей і методів класу `TStrings` наведені в таблиці 12.3.

Таблиця 12.3 – Деякі власні властивості та методи класу `TStrings`

Властивість або метод	Опис
property `Capacity: Integer;`	Поточна ємність списку
property `CommaText: string;`	Об'єднує всі рядки списку в єдиний рядок з укладанням кожного з рядків у подвійні лапки та розділенням їх комою (дві поруч розміщені коми – порожній рядок)
property `Count: Integer;`	Фактична кількість елементів списку
property `Names[Index: Integer]: string;`	Якщо в `Strings[Index]` є знак =, містить частину рядка до цього знака, інакше містить порожній рядок
property `Objects[Index: Integer]: TObject;`	Доступ до об'єкта елемента списку з номером `Index`
property `Strings[Index: Integer]: string;`	Доступ до рядка елемента списку з номером `Index`
property `Text: string;`	Всі рядки інтерпретуються як один рядок з розділенням окремих рядків стандартною ознакою кінця рядка: `#13#10`

Продовження табл. 12.3

Властивість або метод	Опис
function Add(**const** S: **string**): Integer; **virtual**;	Додає новий рядок у кінець списку, повертаючи його індекс з відліком від 0
function AddObject(**const** S: **string**; AObject: TObject): Integer; **virtual**;	Додає новий рядок і новий об'єкт у кінець списку, повертаючи їх індекс з відліком від 0
procedure AddStrings(Strings: TStrings); **virtual**;	Додає групу рядків у кінець списку
procedure Append(**const** S: **string**);	Додає новий рядок у кінець списку
procedure; Clear; **virtual**; **abstract**	Очищає список, видаляючи рядки й об'єкти
procedure Delete(Index: Integer); **virtual**;**abstract**;	Знищує елемент списку з номером Index
procedure Exchange(Index1, Index2: Integer)	Міняє місцями рядки з зазначеними індексами
function IndexOf(**const** S: **string**): Integer; **virtual**;	Якщо в списку є рядок S, повертає його індекс, у противному випадку повертає –1
function IndexOfName(**const** Name: **string**): Integer; **virtual**;	Якщо в списку є рядок типу Name=Value, де Name збігається з параметром, повертає його індекс, у противному випадку повертає –1
function IndexOfObject(AObject: TObject): Integer; **virtual**;	Якщо в списку є об'єкт AObject, повертає індекс зв'язаного з ним рядка, у противному випадку повертає –1
procedure Insert(Index: Integer; **const** S: **string**); **virtual**; **abstract**;	Вставляє рядок у зазначену позицію списку, розширюючи список. Якщо рядок зв'язаний з об'єктом, використовують InsertObject
procedure InsertObject(Index: Integer; **const** S: **string**; AObject: TObject); **virtual**;	Вставляє рядок разом з об'єктом у позицію Index списку
procedure LoadFromFile(**const** FileName: **string**); **virtual**;	Завантажує в рядки списку рядки текстового файлу з ім'ям FileName
procedure Move(CurIndex, NewIndex: Integer);**virtual**;	Переміщає рядок (разом з об'єктом, якщо він зв'язаний з рядком) з положення CurIndex у положення NewIndex
procedure SaveToFile(**const** FileName: **string**); **virtual**;	Записує рядки списку як рядки текстового файлу з ім'ям FileName

Характерною рисою об'єктів класу TStrings є те, що їх властивості Strings і Text почасти дублюють один одного. При цьому властивість Strings забезпечує доступ до окремих рядків за індексом, а властивість Text дозволяє звернутися до всього тексту, як до єдиного рядка.

12.3. Списки класу TStringList

Клас TStringList, будучи безпосереднім нащадком класу TStrings, також є класом, що дозволяє створювати списки, елементами яких є пари типу рядок-об'єкт. У цьому класі перекриваються абстрактні методи класу TStrings, що забезпечує повноцінне використання класу TStringList у програмах. Крім перекритих абстрактних батьківських методів, цей клас містить ряд власних методів.

Елементи списку TStringList зберігаються в масиві, оголошеному як поле FList, що має тип PStringItemList, визначений у модулі Classes у такий спосіб:

```
type
  PStringItem = ^TStringItem;
  TStringItem = record
    FString: string;
    FObject: TObject;
  end;
  PStringItemList = ^TStringItemList;
  TStringItemList=array[0..MaxListSize]of TStringItem;
```

Екземпляр класу TStringList створюється за допомогою конструктора Create, наслідуваного класом TStringList від класу TObject. При цьому в міру необхідності створений список розширюється відповідно до такого ж принципу, що і для класу TList.

Крім успадкованих від класу TStrings властивостей, даний клас має такі власні властивості:

- **property** CaseSensitive: Boolean – керує урахуванням регістра літер при порівнянні рядків списку;
- **property** Duplicates: TDuplicates – забезпечує керування можливістю поміщати у відсортований список кілька ідентичних рядків (якщо список не відсортований, властивість ігнорується);
- **property** Sorted: Boolean – установлює необхідність автоматичного сортування рядків за абеткою відповідно до кодування Windows.

Якщо властивість `CaseSensitive` має значення `True`, то при порівнянні рядків ураховується регістр літер, у тому числі символів національного алфавіту. У противному разі при порівнянні регістр літер не враховується.

Тип `TDuplicates` є таким переліченим типом:

```
type
  TDuplicates = (dupIgnore, dupAccept, dupError).
```

Можливі значення цього типу вказують на наступне:

- `dupIgnore` – копія рядка не вставляється в список;
- `dupAccept` – дозволяється вставка копій рядків;
- `dupError` – спроба вставки копії рядка викликає виняток `EStringListError`.

Якщо властивість `Sorted` має значення `True`, то це означає, що рядки списку будуть сортуватися автоматично. При цьому вставка рядків за допомогою методу `Insert` неможлива, – для додавання нових рядків потрібно використовувати метод `Add`. Якщо ж `Sorted=False`, рядки автоматично не сортуються, і можливі як вставка нових рядків усередину списку, так і додавання їх у кінець списку.

У класу `TStringList` є дві властивості-події (OnChange та OnChanging), які визначають реакцію на зміну набору рядків. Подія OnChange виникає після останньої зміни, а подія OnChanging – до чергової зміни. За необхідності програміст може визначити опрацьовувачі цих подій.

Оскільки більшість методів даного класу, незважаючи на їхнє перевизначення, мають те ж саме призначення, що й відповідні методи класу `TStrings`, зупинимося тільки на віртуальних методах `CustomSort` і `Sort`.

Процедура без параметрів

```
procedure Sort;
```

сортує рядки у випадку, якщо `Sorted=False` (при `Sorted=True` рядки сортуються автоматично). Якщо при цьому `CaseSensitive=True`, то рядки сортуються за допомогою функції `AnsiCompareStr`, а якщо `CaseSensitive = False`, то при сортуванні застосовується функція `AnsiCompareText` (див. п. 5.2.4).

Процедура

```
CustomSort(Compare: TStringListSortCompare)
```

сортує рядки з використанням розробленої програмістом функції порівняння, ім'я якої (аналогічно тому, як це робиться при вживанні методу

Sort класу TList), задається як параметр виклику, що має наступний тип:

```
type
 TstringListSortCompare = function(List: TStringList;
                    Index1, Index2: Integer): Integer;
```

Перший параметр функції порівняння забезпечує доступ до списку, що впорядковується, а два наступні параметри (Index1 та Index2) задають номери рядків, які будуть порівнюватися.

Функція порівняння повинна повертати значення згідно з таким правилом:

- менше 0, якщо рядки з номерами Index1 та Index2 вже розташовані в потрібному порядку;
- 0, якщо рядки однакові;
- значення більше 0, якщо рядки з номерами Index1 та Index2 повинні бути переставлені.

```
//Приклад 12.2
//Створити список з імен і розширень усіх файлів папки
//Windows. Відсортувати список за абеткою. Якщо файл є файлом,
//що містить зображення в форматі BMP, завантажити це зобра-
//ження як об'єкт списку. Вивести список імен файлів таким
//чином, щоб спочатку йшли за абеткою імена файлів з розши-
//ренням .BMP, а потім також за абеткою імена
//інших файлів. Організувати перегляд зображень.
```

Скористаємося формою з прикладу 12.1, розташувавши в її лівій частині компонент Image для показу зображень і видаливши кнопку Button3. Змінимо також деякі властивості компонентів:

- Кнопка Button1:
  Caption — Імена
- Кнопка Button2
  Caption — Слайди
- Компонент Image:
  Align — alClient
  Name — imOut1

Визначимо в розділі **implementation** змінну List типу TStringList, функцію порівняння Compare та опрацьовувачі події OnClick кнопок:

```
function Compare(List: TStringList;
                 Index1, Index2: Integer): Integer;
begin                  //Якщо два рядки списку, що стоять поруч,
```

```
  if List.Strings[Index1] = List.Strings[Index2]    //однакові
  then Result:= 0      //(такого не може бути), повертається 0
  else         //Нижче: 1-й рядок - ім'я BMP-файлу, а 2-й - ні
    if (ExtractFileExt(List.Strings[Index1]) = '.BMP') and
       (ExtractFileExt(List.Strings[Index2]) <> '.BMP')
    then Result := -1                //Упорядкування правильне
    else                  //Нижче: два рядки - імена BMP-файлів
      if (ExtractFileExt(List.Strings[Index1]) = '.BMP') and
         (ExtractFileExt(List.Strings[Index2]) = '.BMP')
      then                                //Тоді порівнюємо імена
        if AnsiCompareStr(List.Strings[Index1],
                          List.Strings[Index2]) < 0
        then Result := -1                    //Вже впорядковані
        else Result := 1         //Треба переставити два рядки
      else
        if (ExtractFileExt(List.Strings[Index1]) <> '.BMP')
           and (ExtractFileExt(List.Strings[Index2]) = '.BMP')
        then Result := 1                //Потрібно переставляти
        else         //Якщо обидва рядки - не імена BMP-файлів
          if AnsiCompareStr(List.Strings[Index1],
                            List.Strings[Index2]) < 0
          then Result := -1                  //Не переставляємо
          else Result := 1;               //Будемо переставляти
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Rec: TSearchRec;
  c, i: Integer;
begin              //Спочатку створюємо список і шукаємо запис
  List := TStringList.Create;                    //про будь-який
  c := FindFirst('C:\Windows\*.*', faAnyFile, Rec);     //файл
  while c = 0 do begin           //Поки такий запис виявляється
    with Rec do
      if Attr and faDirectory = 0 then begin       //Не папка?
        if AnsiUpperCase(ExtractFileExt(Name)) <> '.BMP'
                        //Не .BMP-файл? Тоді додаємо ім'я файлу
        then List.Add(AnsiUpperCase(Name))           //у список
        else begin           //Якщо ж це BMP-файл, то створюємо
                                        //об'єкт-картинку разом
         List.AddObject(AnsiUpperCase(Name),         //з рядком
                        TBitMap.Create);      //і завантажуємо
         (List.Objects[List.Count - 1]as    //у нього BMP-файл
          TBitMap).LoadFromFile('C:\Windows\' + Name);
        end;
      end;
    c := FindNext(Rec);          //Шукаємо запис про новий файл
  end;                 //Нижче сортування списку з використанням
```

```
  List.CustomSort(Compare);        //функції порівняння Compare
  for i := 0 to List.Count-1 do //Виведення списку імен файлів
    mmOutput1.Lines.Add(List.Strings[i]);
  Button1.Hide;                    //Приховуємо кнопку "Імена",
  Button2.Show;                    //показуємо кнопку "Слайди",
  imOut1.Show;               //і компонент для показу зображень
end;

procedure TForm1.Button2Click(Sender: TObject);
begin       //Тут Tag - лічильник розглянутих елементів списку
  mmOutput1.Hide;                          //Приховуємо Memo-поле
  if (Tag < List.Count) and //Якщо не переглянутий весь список
     (List.Objects[Tag] <> nil) //і з рядком зв'язаний об'єкт,
  then begin                              //то виводимо зображення
    imOut1.Picture.Bitmap := (List.Objects[Tag] as TBitMap);
    Tag := Tag + 1;                       //Збільшуємо лічильник
  end
  else Button2.Hide;                //Приховуємо кнопку "Слайди"
end;
```

Клік мишкою над кнопкою Імена приводить до виклику процедури `TForm1.Button1Click`, у якій переглядається папка `Windows` з вибором всіх імен, які не є іменами папок (при цьому застосована бітова операція **`and`**). Імена файлів за допомогою функції `AnsiUpperCase` записуються у верхньому регістрі, і з них за допомогою функції `ExtractFileExt` витягається розширення з метою впізнавання BMP-файлів. Далі додається новий елемент у список, причому додавання елемента, пов'язаного з BMP-файлом, проводиться методом `AddObject`. Останній одночасно вставляє в список рядок й асоційований з ним об'єкт, що створюється конструктором `TBitMap.Create` (у даному випадку об'єкт класу `TBitMap`). У створений об'єкт завантажується зображення з файлу. При цьому за допомогою операції **`as`** тип об'єкта приводиться до типу `TBitMap`. Ім'я файлу, що не містить зображення у форматі BMP, додається в список методом `Add`. Після формування списку він упорядковується за допомогою методу `CustomSort`, що для порівняння використовує функцію `Compare`, визначену в розділі **`implementation`**.

Функція `Compare` виконує порівняння двох рядків з номерами `Index1` та `Index2` у списку типу `TStringList`. Вона повертає значення `0`, якщо порівнювані рядки однакові, значення `-1`, якщо рядок з номером `Index1` – ім'я файлу з розширенням BMP, а рядок з номером `Index2` – ім'я іншого файлу, а також коли результат порівняння цих двох рядків за допомогою функції `AnsiCompareStr` дорівнює `-1`. В інших випадках ця функція повертає значення `1`, яке говорить про те, що рядки повинні бути

переставлені в списку. При цьому припускається, що вміст двох порівнюваних рядків наведено у верхньому регістрі (при порівнянні розширень іде зіставлення з підрядком `'.BMP'`).

В опрацьовувачі події OnClick кнопки Button2 для поточного значення властивості `Tag`, що служить в даному разі для вказівки порядкового номера поточного елемента списку `List`, та виводиться зображення, що зберігається в об'єкті `List.Objects[Tag]`, якщо з рядком зв'язаний об'єкт. При цьому враховано, що якщо з рядком не зв'язаний об'єкт, то `List.Objects[Tag]=` **`nil`**. Крім того, враховано й те, що в усі об'єкти, які потрапили в список, завантажені BMP-зображення, у зв'язку із чим є коректним перетворення `(List.Objects[Tag]` **`as`** `TBitMap)` і запис отриманого результату перетворення у властивість `imOut1.Picture.Bitmap`. Запис результату перетворення в цю властивість приводить до виведення зображення (про компонент Image йтиметься далі в п. 17.3.1).

Запитання для контролю і самоконтролю

1. Для чого використовується клас `TList`?
2. Назвіть базові властивості класу `TList`.
3. Як виконується видалення списку?
4. Яка особливість методу `Sort` і що потрібно зробити, щоб відсортувати з його допомогою елементи списку за заданим правилом?
5. Яке призначення класу `TStrings`?
6. Які списки створює клас `TStringList`?
7. Яким способом здійснюється завантаження текстового файлу в список класу `TStringList`?
8. Як зберегти рядки списку класу `TStringList` у файлі?
9. Як забезпечити недопущення розміщення ідентичних рядків у відсортований список класу `TStringList`?
10. Що потрібно зробити, щоб при додаванні рядків у список класу `TStringList` виконувалося їх автоматичне сортування?
11. Що визначають події OnChange та OnChanging класу `TStringList`?

Завдання для практичного відпрацьовування матеріалу

1. Дано натуральне число *n*. Увести *n* дійсних чисел, розміщаючи їх у списку класу `TList` за зростанням.

2. Дано натуральне число n. Записати в список класу `TList` послідовність з n цілих чисел. Відсортувати їх таким чином, щоб спочатку розташовувалися всі непарні значення, а потім – парні. Порядок проходження значень усередині кожної з груп зберегти.
3. Організувати введення дійсних чисел до першого від'ємного з розміщенням їх у списку класу `TList`. Видалити ті з них, які перевищують середнє арифметичне всіх записаних у список значень.
4. Уводити цілі числа доти, доки їх середнє арифметичне не стане більшим за 10. Розмістити введені дані в списку класу `TList` у порядку, зворотному порядку введення.
5. Дано натуральне число n. Записати n цілих чисел у список класу `TList`. Якщо у введеній послідовності парні та непарні числа перемежовуються, змінити послідовність елементів списку на протилежну. У противному випадку залишити послідовність без зміни.
6. Для списку, що буде утворений у результаті розв'язання попередньої задачі, вставити після кожного парного числа його подвоєне значення.
7. Дано натуральне число n. Одержати послідовність d_k, d_{k-1}, …, d_0 десяткових цифр числа 2^n, тобто таку послідовність, у якій кожен член d_i одночасно відповідає двом умовам: $0 \le d_i \le 9$ та $d_k \cdot 10^k + d_{k-1} \cdot 10^{k-1} + \ldots + d_0 = 2^n$.
8. Одержати аналогічні послідовності десяткових цифр числа:

$$а)\ 1000! + 2^{1000}; \quad б)\ 1000! - 2^{1000}.$$

9. Дано текстовий файл. Видалити в ньому всі порожні рядки. Для розв'язання задачі переписати файл у список класу `TStringList` та обробити його.
10. Дано текстовий файл. Переписати його рядки в порядку, зворотному початковому порядку проходження (скористатися для розв'язання списком класу `TStringList`).
11. Дано текстовий файл. Упорядкувати його рядки за зростанням їх довжин. Для розв'язання задачі переписати файл у список класу `TStringList` і скористатися методом `Sort`.

13. ОБРОБКА ОСОБЛИВИХ (ВИНЯТКОВИХ) СИТУАЦІЙ

При некоректному виконанні програми відбувається або аварійне її завершення, або ігнорування помилки з можливим подальшим проявом наслідків її виникнення. Для контролю деяких помилок у Delphi можуть бути застосовані директиви компілятору, які при їх активізації забезпечують тільки стандартну реакцію, що зводиться до видачі інформаційного повідомлення з наступним завершенням процесу виконання програми. Крім того, в Delphi збереглася можливість використання так званих процедур завершення, які дозволяють здійснювати контроль наявності помилок і забезпечувати реалізацію нестандартної реакції на їх появу. У той же час існують спеціальні засоби для обробки особливих ситуацій, які виникають при виконанні програми. Такі ситуації називають ***винятковими (особливими) ситуаціями*** або ***винятками*** (*exceptions*).

Механізм обробки особливих ситуацій не є прерогативою мови Delphi. Цей механізм широко застосовується в мові C++. Більш того, він існував і в інших мовах програмування ще до появи як Delphi, так і C++. Наприклад, у мові PL/1 програмісти могли працювати і з убудованими (стандартними) винятковими ситуаціями, і з ситуаціями, які створювалися за вказівкою програміста при настанні якої-небудь події.

Винятки не скасовують можливість використання директив компілятору.

13.1. Захищені блоки і їх використання як механізму обробки винятків

Для обробки винятків у Delphi у модулі `SysUtils` визначено спеціальний клас `Exception`, що є безпосереднім нащадком класу `TObject` і породжує декілька класів-нащадків, які також служать для обробки винятків.

Оголошення класу `Exception` має такий вигляд:

```
Exception = class(TObject)
  private
    FMessage: string;
    FHelpContext: Integer;
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string;
                const Args: array of const);
    constructor CreateRes(Ident: Integer); overload;
    constructor CreateRes(ResStringRec:
                PResStringRec); overload;
    constructor CreateResFmt(Ident: Integer;
                const Args: array of const); overload;
    constructor CreateResFmt(
                ResStringRec: PResStringRec;
                const Args: array of const); overload;
    constructor CreateHelp(const Msg: string;
                AHelpContext: Integer);
    constructor CreateFmtHelp(const Msg: string;
                const Args: array of const;
                AHelpContext: Integer);
    constructor CreateResHelp(Ident: Integer;
                AHelpContext: Integer); overload;
    constructor CreateResHelp(
                ResStringRec: PResStringRec;
                AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(
                ResStringRec: PResStringRec;
                const Args: array of const;
                AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(Ident: Integer;
                const Args: array of const;
                AHelpContext: Integer); overload;
    property HelpContext: Integer read FHelpContext
                                  write FHelpContext;
    property Message: string read FMessage
                             write FMessage;
  end;
```

За допомогою визначених у цьому класі конструкторів можна реалізувати 8 варіантів створення об’єктів. Найпростіший варіант – це виклик конструктора `Create`, параметр якого дозволяє передати в об’єкт (або одержати з нього) текстове повідомлення. Саме повідомлення записується в приватне поле `FMessage`, доступ до якого забезпечується за допомогою

властивості `Message`. Інші конструктори виконують більш складні функції.

Методика обробки винятків базується на понятті ***захищеного блоку***, що прийшло з мови Object Pascal.

Механізм захищеного блоку полягає в тому, що оператори, при виконанні яких можливе виникнення виняткової ситуації, включаються в оформлений спеціальним способом блок, налаштований на деякий перелік виняткових ситуацій, для кожної з яких визначається реакція програми, що задається послідовністю операторів.

Існує два види захищених блоків – **`try`**...**`except`** та **`try`**...**`finally`**.

Захищений блок першого виду має такий формат:

```
try
  оператори
except
  опрацьовувачі_винятків
else
  оператори
end;
```

При цьому ті оператори, правильність виконання яких контролюється в захищеному блоці, розташовуються між службовими словами **`try`** та **`except`**.

Робота захищеного блоку **`try`**...**`except`** полягає в наступному. Спочатку виконуються оператори, розташовані безпосередньо між службовими словами **`try`** та **`except`**, з проведенням контролю коректності їх виконання. Якщо при виконанні якого-небудь з цих операторів виникає виняткова ситуація, то решта операторів, що залишилися, не виконується і керування передається в секцію **`except`**, що містить перелік так званих ***опрацьовувачів винятків***.

Можливі винятки, на які налаштований захищений блок, визначаються іменами класів, що служать перемикачами операторів, які повинні виконатися при виникненні відповідного винятку. Власне опрацьовувач винятку має формат, визначений у такий спосіб:

```
on клас_винятку do оператор;
```

Тут клас_винятку – це ім'я класу, що кодує виняток; оператор – оператор (у тому числі й складений), що повинен бути виконаний при виявленні даного винятку (цей оператор не може передавати керування на мітку поза секцією **`except`**).

По закінченні роботи опрацьовувача винятків керування передається першому оператору, розташованому за захищеним блоком (тобто першому

оператору, що стоїть після службового слова **end**). Оператори контрольованої частини захищеного блоку, розташовані за оператором, що збудив виняток, виконані не будуть.

Якщо в секції **except**...**else** не виявляється опрацьовувач, що може обробити виняток, який був генерований, то керування передається першому оператору, розташованому за службовим словом **else**.

Секція **else** у захищеному блоці може бути відсутня. Якщо в цьому випадку в секції **except** відсутній опрацьовувач, здатний обробити генерований виняток, видається стандартне повідомлення про появу даної помилки, і програма припиняє виконання.

Підкреслимо: якщо програма компілювалася таким чином, що в ній не здійснюється перевірка виникнення даної виняткової ситуації, механізм винятків також не спрацює. Так, якщо при компіляції відключена перевірка переповнення (скинутий прапорець Overflow Checking на вкладці Project ► Options ICP Delphi), механізм обробки винятків не спрацює навіть у випадку, коли переповнення фактично відбудеться.

В опрацьовувачі винятків перед ім'ям класу винятку можна визначити ідентифікатор:

on ідентифікатор **:** клас_винятку **do** оператор;

Цей ідентифікатор є екземпляром класу, що відповідає опрацьовувачу винятку. При збудженні винятку створюється локальний об'єкт, копія якого передається в опрацьовувач винятку й ініціалізує змінну, ім'я якої подано після службового слова **on**. Екземпляр класу, ідентифікатор якого зазначений після службового слова **on**, є локальним, тобто відомий (існує) тільки усередині оператора, що стоїть після службового слова **do**, і використовується для забезпечення явного доступу до полів і методів класу_винятку. Наприклад,

```
try
      ...              //Оператори контрольованого блоку
except
  on E: Exception do ErrorDialog(E.Message,
                                 E.HelpContext);
end;
```

Захищений блок може взагалі не містити опрацьовувачі винятків:

```
try
  оператори
except
  оператори
end;
```

У цьому випадку контролюється поява будь-якої помилки при виконанні операторів, розташованих між службовими словами **try** та **except**, з переходом до першого оператора, що стоїть після **except**, якщо буде зафіксована яка-небудь виняткова ситуація.

Формат захищеного блоку **try**...**finally** такий:

```
try
  оператори
finally
  оператори
end;
```

Особливістю даного захищеного блоку є те, що оператори секції **finally**...**end** виконуються в обов'язковому порядку незалежно від того, виник чи ні виняток у контрольованій секції **try**...**finally**. Якщо виняток не генерувався, то спочатку виконаються всі оператори секції **try**...**finally**, а потім – усі оператори секції **finally**...**end**. Якщо ж виняток виник, то оператори, розташовані в секції **try**...**finally** за «винуватцем» винятку, пропускаються, і керування передається першому оператору секції **finally**...**end**.

Захищений блок **finally**...**end** використовується в тих випадках, коли необхідно гарантувати виконання деяких завершальних дій, навіть якщо процес обробки перервався появою помилки (наприклад, закрити файл). Так, у наступному фрагменті програми файл `F` буде закритий, навіть якщо при його обробці виникне помилка:

```
Reset(F);
try
  ...                                    //Обробка файлу
finally
  CloseFile(F);
end;
```

Захищені блоки можуть бути вкладеними – вони можуть зустрітися у всіх секціях (**try**...**except**, **except**...**else**, **else**...**end**, **except**...**end**, **try**...**finally**, **finally**...**end**), у тому числі в опрацьовувачах винятків.

13.2. Використання стандартних класів винятків

У модулях `SysUtils`, `Classes`, `Math`, `Graphics` та інших модулях оголошено близько 100 стандартних класів винятків, які або безпосередньо породжені класом `Exception`, або породжені деякими з його спадкоємців. Тільки виняток `OutlineError` породжений не класом

Exception, як це має місце для інших винятків, а є безпосереднім спадкоємцем класу TObject.

Нащадками класу Exception є такі класи винятків (на початку списку подано винятки, що зустрічаються в найпростіших програмах; рівні списку відповідають рівням спадкування від класу Exception):

- EAbort – обробка будь-якого винятку без видачі повідомлення;
- EAbstractError – спроба виклику абстрактного методу;
- EArrayError – помилка при роботі з масивом (неправильний індекс, додавання занадто великої кількості елементів у масив з фіксованою місткістю, вставка рядка у відсортований список);
- ECommonCalendarError – спроба введення неправильної дати в об'єкт класу TCommonCalendar або його нащадка;
  - EDateTimeError – спроба введення неправильної дати або часу при використанні компонента TDateTimePicker;
  - EMonthCalError – спроба введення неправильної дати в компонент класу TMonthCalendar або його нащадка;
- EConvertError – помилка перетворення рядка до цілого або дійсного числа, дати або часу за допомогою функцій StrTo... (або помилка зворотного перетворення);
- EExternal – виняток, що містить звіти винятків Windows;
  - EAccessViolation – спроба звертання до області пам'яті, зарезервованої для виконуваного коду або тієї, що не належить програмі, використання недійсного вказівника;
  - EControl – натиснута клавіша <Ctrl+C> при роботі в консольному режимі;
  - EExternalException – помилка, код якої не визначений в Delphi;
  - EIntError – помилка в цілочислових обчисленнях (автоматично під час виконання не збуджується, але служить базою для інших виключень цілочислової математики);
  - EDivByZero – цілочислове ділення на нуль;
    - EIntOverflow – цілочислове переповнення, тобто спроба присвоєння значення, що вимагає для свого подання більше 32 біт (для забезпечення можливості автоматичного збудження цього винятку повинна бути включена директива перевірки переповнення $Q+ або встановлений прапорець Overflow Checking на вкладці Compiler команди Project ► Options в ICP Delphi);

        - `ERangeError` – цілочисловий результат перевищує ємність цілого типу або вихід індексу масиву за допустимий діапазон (для забезпечення можливості автоматичного збудження даного винятку має бути включена директива перевірки діапазону `$R+` або встановлений прапорець Range Checking на вкладці Compiler команди Project ► Options в ICP Delphi);
    - `EMathError` – помилка при виконанні дій із плаваючою точкою (автоматично під час виконання не збуджується);
        - `EInvalidArgument` – вихід за допустимий діапазон аргументу математичної функції модуля `Math`;
        - `EInvalidOp` – помилка при виконанні операції з плаваючою точкою;
        - `EOverflow` – переповнення з плаваючою точкою (результат занадто великий для ємності регістра співпроцесора);
        - `EUnderflow` – зникнення порядку (результат занадто малий для запису в дійсну змінну);
        - `EZeroDivide` – дійсне ділення на нуль;
    - `EPrivilege` – спроба виконання привілейованої операції (такі операції може виконувати тільки ядро Windows);
    - `EStackOverflow` – переповнення виділеного стека;
- `EListError` – неправильна дія зі списком (у тому числі об'єктом класу `TStringList`) або рядком (наприклад, звертання до елемента за індексом, меншим від нуля або більшим від максимально допустимого, вставка елемента у відсортований масив);
- `EStringListError` – звертання до рядка, індекс якого виходить за діапазон можливих значень для списку рядків;
- `EInvalidGridOperation` – спроба виконання недопустимої операції над таблицею (наприклад, неправильні рядок або стовпець);
- `EHeapException` – неправильна операція над динамічними даними;
    - `EInvalidPointer` – спроба використання недопустимого вказівника (спроба повторного видалення вказівника, робота з уже віддаленим вказівником);
    - `EOutOfMemory` – запит занадто великого для даної конфігурації Windows обсягу пам'яті (користувачеві не рекомендується примусово збуджувати даний виняток – замість цього треба викликати процедуру `OutOfMemoryError`);

        - `EOutOfResources` – запит дескриптора вікна при вичерпанні Windows ліміту дескрипторів;
- `EParserError` – при читанні з потоку виникла помилка перетворення тексту у двійкові дані;
- `EStreamError` – будь-яка помилка при роботі з потоком даних;
    - `EFCreateError` – помилка при створенні файлу (створення файлу в неіснуючому каталозі або на захищеному від запису пристрої, недостатньо прав доступу для зміни файлу);
    - `EFOpenError` – помилка відкриття файлу (наприклад, відсутність файлу даних);
    - `EFilerError` – спроба повторно зареєструвати в потоці один і той самий клас;
        - `EInvalidImage` – спроба прочитати ресурс з файлу, у якому цей ресурс відсутній;
        - `EClassNotFound` – для компонента, що читається з файлу, не знайдено відповідний клас;
        - `EMethodNotFound` – з потоку даних прочитано об'єкт, для якого не знайдений пов'язаний з класом цього об'єкта метод;
        - `EReadError` – програма не може прочитати з потоку потрібну кількість байт або не може прочитати властивість при створенні форми;
        - `EWriteError` – програма не може записати в потік потрібну кількість байт або записати в потік одну з властивостей компонента;
- `EInOutError` – будь-яка помилка у файлових операціях введення/виведення (об'єкт цього класу у своєму полі `ErrorCode` містить код помилки – див. підрозд. 6.3);
- `EInvalidCast` – спроба недопустимого приведення типів за допомогою операції **`as`**;
- `EInvalidGraphic` – спроба завантажити зображення з файлу, що має недопустимий формат;
- `EInvalidGraphicOperation` – недопустима графічна операція;
- `EVariantError` – помилка при роботі з варіантним типом;
- `EInvalidOperation` – спроба виконання віконної операції компонентом, що не має вікна;
- `ELowCapacityError` – спроба виділення пам'яті на пристрої, що не має вільної пам'яті;

- `EMenuError` – помилка при роботі з меню (спроба видалення пункту з порожнього меню, спроба пункту меню вставити себе у власний список пунктів, посилання на неіснуюче меню);
- `EOSError` – помилка операційної системи;
- `EPrinter` – помилка принтера;
- `EPropertyConvertError` – помилка типу при читанні або записі властивості;
- `EPropertyError` – помилка доступу при записі властивості;
- `EResNotFound` – не знайдено зазначений ресурс у файлі ресурсів або DFM-файлі;
- `EAppletException` – помилка при створенні аплет-додатків;
- `EAssertionFailed` – логічний вираз, що тестується за допомогою налаштовувальної процедури `Assert`, має значення `False`;
- `EBitsError` – при звертанні до властивості `Bits` об'єкта `TBits` задано індекс менший від нуля або більший за максимально допустиме значення;
- `EBrokerException` – об'єкт-брокер не може знайти сервер;
- `ECacheError` – помилка в наборі даних для компонента `TDecisionCube`;
- `EComponentError` – помилка при роботі з компонентом;
- `ECorbaDispatch` – розбіжність інтерфейсів сервера та брокера даних при застосуванні в програмі технології CORBA;
- `ECorbaException` – помилка в програмі, що використовує технологію CORBA;
  - `ECorbaUserException` – визначена користувачем реакція на помилки інтерфейсу;
- `EDatabaseError` – помилка в базі даних;
  - `EDBClient` – неправильна робота `TClientDataSet`;
- `EReconcileError` – помилка оновлення даних у `TClientDataset`;
  - `EDBEngineError` – помилка BDE;
  - `EIBError` – помилка технології IBX;
    - `EIBClientError` – помилка, пов'язана з функціонуванням IBX-клієнта;
    - `EIBInterbaseError` – помилка, пов'язана з функціонуванням сервера в технології IBX;
    - `EIBInterbaseRoleError` – помилка технології IBX, пов'язана з недостатністю прав;

  - EUpdateError – помилка оновлення провайдерського набору даних;
- EDBEditError – вживання даних, несумісних із заданою маскою;
- EDimensionMapError – набір даних, що використовується у кубі розв'язків, не має належного формату;
- EDimIndexError – порушення розмірності масиву даних для куба розв'язків;
- EDSWriter – помилка при підготовці провайдером даних для пакета даних;
- EInterpreterError – неможлива інтерпретація даних блоку даних компонентом класу TDataBlockInterpreter;
- EMCIDeviceError – помилка при роботі з медіаплеєром;
- EOleCtrlError – неможливе встановлення зв'язку з елементом Active;
- EIntfCastError – спроба неприпустимого приведення типів за допомогою операції **as** в OLE-об'єктах;
- EOleError – низькорівнева помилка в технології OLE;
  - EOleRegistrationError – помилка реєстрації OLE-об'єкта в реєстрі Windows;
  - EOleSysError – неправильне виконання команди OLE-автоматизації;
    - EOleException – застосовується неправильний OLE-інтерфейс;
- EOutlineError – помилка доступу до компонента класу TOutline;
- EPackageError – помилка доступу до пакета;
- EPropReadOnly – спроба присвоєння значення властивості «тільки для читання» (при використанні технології OLE);
- EPropWriteOnly – спроба прочитати властивість «тільки для запису» (при застосуванні технології OLE);
- ERegistryException – помилка при роботі з реєстром Windows;
- ESocketConnectionError – помилка при роботі із сокетами Windows (при відправленні або одержанні повідомлень з використанням TSocketConnection);
- ESocketError – помилка при спробі створення, використання або закриття сокетів Windows;

- `EThread` – ситуація боротьби за спільний ресурс у програмі з декількома потоками команд;
- `ETreeViewError` – помилка відображення дерева;
- `EUnsupportedTypeError` – обрано неприпустимий тип поля як вимір у кубі розв'язків;
- `EWin32Error` – помилкове звертання до API-функції Windows.

Один і той самий виняток може оброблятися декількома опрацьовувачами винятків. Наприклад, опрацьовувач для класу `EAbort` здатний обробити будь-який виняток, а такі опрацьовувачі, як `EIntError` або `EMathError` і т. д., можуть обробити винятки для всіх своїх нащадків. Подібного роду опрацьовувачі повинні розміщатися в списку опрацьовувачів винятків після тих опрацьовувачів, дію яких вони можуть перехопити (перекрити). Таким чином, якщо використовується опрацьовувач `EAbort`, він має бути останнім у списку опрацьовувачів, а, наприклад, опрацьовувач `EMathError` потрібно розташовувати після опрацьовувачів винятків `EInvalidArgument`, `EInvalidOp`, `EOverflow`, `Eunderflow` та `EZeroDivide`.

При застосуванні механізму винятків необхідно враховувати те, що середовище за умовчанням налаштоване таким чином, що воно реагує на виникнення виняткових ситуацій, перехоплюючи керування програмою та ускладнюючи тим самим налаштовування. Тому при запуску із середовища Delphi програм, що використовують механізм виняткових ситуацій, треба провести переналаштування середовища. Для цього потрібно виконати команду Tools ► Debugger Options (Сервіс ► Опції Налаштовувача) і в діалоговому вікні, що при цьому відкриється, зняти прапорець Stop on Delphi Exceptions (Зупинятися на винятках Delphi) на вкладці Language Exceptions (Винятки мови), скасувавши тим самим реакцію середовища Delphi на винятки.

Наступний приклад, а також приклад 17.7 ілюструють найпростіші випадки використання стандартних класів винятків.

```
//Приклад 13.1
//Переписати в текстовий файл out.txt по одному в рядок
//усі числа, що записані у вигляді окремих слів у текстовому
//файлі in.txt. Одержати суму цих чисел.
```

Скористаємося для розв'язання задачі формою з прикладу 2.2, видаливши з неї мітку виведення та однорядкове редаговане текстове поле і присвоївши властивості Caption компонента Button1 значення Виконати.

У розділі **`implementation`** визначимо такий опрацьовувач події OnClick кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FIn, FOut: TextFile;
  r, Sum: Extended;
begin
  AssignFile(FIn, 'in.txt');
  AssignFile(FOut, 'out.txt');
  try                    //Початок зовнішнього захищеного блоку
    Reset(FIn);         //У цих двох операторах можливі помилки
    Rewrite(FOut);              //відкриття файлу (EInOutError)
    while not SeekEof(FIn) do begin
      try               //Початок внутрішнього захищеного блоку
        Read(FIn, r);           //Якщо у файлі не тільки числа,
                       //можлива помилка введення (EInOutError)
        WriteLn(FOut, r);
      except
        on EInOutError do;      //Пропускаємо слова - не числа
      end;               //Кінець внутрішнього захищеного блоку
    end;
    CloseFile(FIn);
    CloseFile(FOut);
    Reset(FOut);
    Sum := 0;
    while not Eof(FOut) do begin
      ReadLn(FOut, r);
      Sum := Sum + r;       //Можливе переповнення (EOverflow)
    end;
    CloseFile(FOut);
    mmOutput1.Lines.Add('Сума чисел файлу дорівнює ' +
                        FloatToStr(Sum));
  except
    on EIOError: EInOutError do       //Реакція на помилку при
                                                //відкритті файлу
     ShowMessage('Помилка № ' + IntToStr(EIOError.ErrorCode)
                 + ' при роботі з файлом');
    on EOverflow do  mmOutput1.Lines.Add(
              'Виникло переповнення при підсумовуванні');
  end;                    //Кінець зовнішнього захищеного блоку
end;
```

У тексті підпрограми ілюструється можливість вкладеності захищених блоків **try**...**except**. У зовнішньому захищеному блоці можуть виникнути помилки при відкритті файлів, які спричинять збудження винятку `EInOutError`. У випадку, якщо збуджується цей виняток, у програмі передбачена видача повідомлення, що, крім інформації про помилку при роботі з файлом, містить також номер помилки. В опрацьовувачі винятку `EInOutError` з метою забезпечення одержання

номера помилки описано ідентифікатор `EIOError` для доступу до поля `ErrorCode` класу `EInOutError`, що містить номер помилки.

Формально в зовнішньому захищеному блоці при підсумовуванні можливе переповнення, що призводить до збудження винятку `EOverflow`. У цьому випадку програма видасть повідомлення без яких-небудь додаткових дій.

Внутрішній захищений блок відслідковує тільки одну виняткову ситуацію – `EInOutError`. При читанні даних з текстового файлу в числові змінні виконується автоматичне перетворення рядкового подання в числове. Якщо у файлі, крім чисел, містяться слова, що не являють собою запис дійсних чисел, може виникнути виняток `EInOutError`. Оскільки у внутрішньому захищеному блоці жодної реакції на появу виняткової ситуації в цьому випадку не передбачено, слова, що не є записом чисел, будуть просто пропускатися, не впливаючи на підсумовування.

На використанні винятків можуть будуватися алгоритми розв'язання задач. Розглянемо, наприклад, наступну задачу.

```
//Приклад 13.2
//Дано квадратний масив з N*N дійсних чисел, де
//N – натуральне число. Обчислити добуток усіх елементів,
//розташованих 1) на головній діагоналі та 2) на всіх лініях,
//які паралельні головній діагоналі і розташовані правіше та
//вище неї з продовженням ліній у дописані до масиву праворуч
//перших його N-1 стовпців. Обчислити також суму отриманих
//добутків. Дописування стовпців не проводити.
```

Умову задачі ілюструє рис. 13.1.

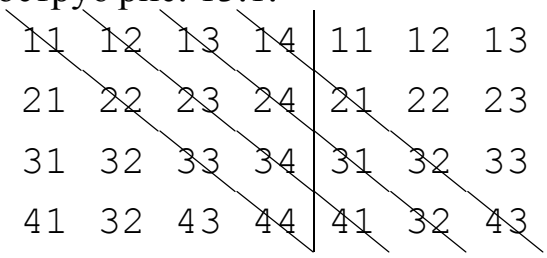


Рис. 13.1. Розглянуті у прикладі 13.2 лінії

Помістимо на форму компонент Memo, присвоївши йому ім'я mmOutpt1, а також звичайну кнопку Button1. Тоді для розв'язання задачі можна скористатися таким опрацьовувачем події OnClick кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  N: Integer;
  i, j, k, c: Integer;
  a: array of array of Real;
  Sum, r, Product: Real;
```

```
begin                    //Контроль правильності перетворення,
                  //контроль допустимості значень, що вводяться,
           //і контроль правильності відведення пам'яті опущено
  N := StrToInt(InputBox('Введення розмірності масиву',
                         'Уведіть розмірність', '1'));
  SetLength(a, N, N);
  for i := 0 to N - 1 do
    for j := 0 to N - 1 do
      a[i, j] := StrToFloat(InputBox('Введення масиву',
                        'Уведіть елемент a[' + IntToStr(i + 1) +
                        ',' + IntToStr(j + 1) + ']', '1'));
  {$R+}                          //Включаємо контроль діапазону
  try                     //Початок зовнішнього захищеного блоку
    Sum := 0.0;
    for c := 0 to N - 1 do begin       //Обчислюємо N добутків
      Product := 1.0;
      j := c;
      k := 0;                                  //k - номер рядка
      while k < N do begin
        try             //Початок внутрішнього захищеного блоку
          r := a[k, j];   //Тут можливий неправильний індекс j
          Product := Product * r;   //Тут можливе переповнення
        except
          on ERangeError do begin      //При помилці діапазону
            j := -1;                  //змінюємо індекс стовпця
            k := k - 1;     //і рядка (нижче вони збільшаться)
          end;
        end;              //Кінець внутрішнього захищеного блоку
        j := j + 1;
        k := k + 1;
      end;
      Sum := Sum + Product;   //Тут також можливе переповнення
    end;
    mmOutput1.Lines.Add(FloatToStr(Sum));
  except
    on EOverflow do mmOutput1.Lines.Add('Переповнення');
  end;                     //Кінець зовнішнього захищеного блоку
end;
```

У внутрішньому захищеному блоці можливий виняток ERangeError, якщо другий індекс вийде за допустимий діапазон (опиниться рівним N). У випадку збудження цього стандартного винятку виконується його опрацьовувач, у якому змінюються індекси таким чином, щоб при поверненні на початок циклу номер рядка залишався тим самим, але здійснювався перехід до стовпця з номером 0 (значення індексів доводиться брати на 1 меншим, оскільки наприкінці циклу вони збільшу-

ються на 1). Природно, можна було замість механізму збудження винятку скористатися оператором **if**.

Процедура демонструє також можливість вкладеності захищених блоків.

13.3. Створення власних класів і примусове збудження винятків

У Delphi передбачено можливість роботи не тільки зі стандартними винятками, розглянутими в підрозд. 13.2, але і з винятками, створюваними за вказівкою програміста. У такому разі, визначивши деякий стан програми як особливий, програміст передбачає аналіз результатів виконання оператора, у якому цей стан може бути досягнутий, або перевіряє початкові дані на допустимість. Для забезпечення можливості реакції програми на нестандартний стан програміст може створити власний клас обробки винятків.

Створюваний клас обробки винятків визначається як нащадок класу Exception або одного з його спадкоємців. Звичайно як прабатько нестандартного класу обробки винятків виступає клас EAbort. При цьому новий клас-нащадок у складних програмах перекриває деякі з методів прабатька або розширює набір його полів, властивостей і методів.

Якщо стандартні винятки збуджуються автоматично, то винятки, створені програмістом, збуджуються тільки примусово. Для примусового збудження винятку (у тому числі й стандартного) використовується службове слово **raise** (збудити). При виході програми на це службове слово збуджується виняток, конструктор якого зазначено зразу після службового слова **raise**, після чого виконується перехід у секцію **except**...**end** або **finally**...**end** для пошуку відповідного опрацьовувача. Якщо за допомогою слова **raise** виняток збуджений у секції **except**...**end** або **finally**...**end**, то виконується вихід з даного захищеного блоку на більш вищий рівень вкладеності.

Формат оператора **raise** такий:

```
raise виклик_конструктора;
```

Наприклад:

```
raise EMathError.Create;
```

Оператор **raise** може не містити звертання до конструктора:

```
raise;
```

У цьому випадку збуджується виняток класу Exception.

При виконанні оператора **raise** за допомогою викликаного конструктора створюється екземпляр класу відповідного типу із виходом з контрольованого блоку з метою пошуку опрацьовувача, здатного обробити виняток, що був збуджений. Якщо звертання до оператора **raise** здійснене поза захищеним блоком, то викликається конструктор з пошуком відповідного опрацьовувача винятків нижче по тексту програми. Якщо такий опрацьовувач не виявляється, то дія обмежується виконанням конструктора. Звертання до оператора **raise** у форматі

```
raise;
```

поза захищеним блоком заборонене компілятором.

```
//Приклад 13.3
//Дані два цілих числа M та N. Знайти їх найбільший спільний
//дільник НСД(M,N) за допомогою алгоритму Евкліда,
//який може бути описаний наступною послідовністю дій.
//Якщо M>0, N>0, M>N, то НСД(M,N)=НСД(N,R),
//де R – остача від ділення M на N. При N=0 НСД(M,N)=M.
```

Розмістимо на панелі форми з прикладу 2.2 ще одне поле введення з ім'ям edInput2, над яким розташуємо мітку lbOutput2. Запишемо в поле Text компонента edInput2 значення 0, а у властивості міток lbOutput1 та lbOutput2 відповідно тексти Число M і Число N.

У секції **interface** модуля визначимо власний клас обробки винятків як нащадок класу EAbort:

```
EVal = class(EAbort)              //Визначення власного класу
end;                  //обробки винятків (відомий поза модулем)
```

У цій же секції опишемо заголовок функції обчислення найбільшого спільного дільника:

```
function NSDEucl(M, N: Integer): Integer;
```

У розділі **implementation** визначимо описану в секції **interface** функцію NSDEucl, а також опрацьовувач події OnClick кнопки Button1:

```
function NSDEucl(M, N: Integer): Integer;
var
  R: Integer;
begin
  if M < N then begin
    R := M;
    M := N;
    N := R;
  end;
```

```
  while N > 0 do begin
    R := M mod N;
    M := N;
    N := R;
  end;
  Result := M;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  M, N: Integer;
begin
  try                                  //Початок захищеного блоку
    M := StrToInt(edInput1.Text);        //Тут можливі помилки
    N := StrToInt(edInput2.Text);              //перетворення
    if M < 0 then
                                               //Збудження винятку
      raise EVal.Create('Перше число від''ємне');
    if N < 0 then
                                               //Збудження винятку
      raise EVal.Create('Друге число від''ємне');
    if (M = 0) and (N = 0) then
                                               //Збудження винятку
      raise EVal.Create('Обидва числа дорівнюють нулю');
    mmOutput1.Lines.Add(IntToStr(NSDEucl(M, N)));
  except
   on E: EVal do                  //Реакція на від'ємне значення
     ShowMessage(E.Message);        //або два нульових значення
   on EConvertError do        //Реакція на помилку перетворення
     ShowMessage('Запис числа є неправильним');
  end;                                  //Кінець захищеного блоку
end;
```

У процедурі `TForm1.Button1Click` у трьох місцях передбачено примусове збудження винятку `EVal`, що не є стандартним винятком Delphi. Клас `EVal` оголошений як нащадок стандартного класу `EAbort` і не має додаткових полів, властивостей і методів. Опрацьовувач винятку `EVal` виконує тільки видачу повідомлення за допомогою процедури `ShowMessage`. Для забезпечення можливості передачі повідомлення в процедуру в опрацьовувачі винятку `EVal` описується об'єкт `E`. Виняток `EVal` збуджується як реакція на те, що одне з оброблюваних значень від'ємне або від'ємні обидва введених значення. У програмі відслідкована також можливість появи помилки перетворення до цілого типу значень, що вводяться.

Взагалі кажучи, винятки введено в мову в основному для надання програмісту можливості під час виконання (run-time) програми проводити

обробку виникаючих ситуацій, з якими не може впоратися виконувана підпрограма. Основна ідея полягає в тому, що «функція, яка стикається з нерозв'язною проблемою, формує виняток у сподіванні на те, що функція, яка здійснила її виклик (прямо або побічно) зможе обробити проблему»[1]. Механізм виключень дозволяє переносити аналіз та обробку ситуації з точки її виникнення в інше місце програми, спеціально призначене для її обробки.

У наступному прикладі показано, що «генератор» винятку (**raise**) може перебувати поза контрольованим блоком усередині підпрограми, звертання до якої здійснюється в контрольованому блоці. У контрольованому блоці розміщається звертання до підпрограми (у цьому випадку до функції NSDEucl) незалежно від того, правильним або помилковими є значення її параметрів.

```
//Приклад 13.4
//Задача з прикладу 13.3 з рознесенням місця розташування
//генератора винятку і його опрацьовувача.
```

Скористаємося формою та описами з прикладу 13.3, але дещо змінимо функцію NSDEucl і опрацьовувач події OnClick кнопки Button1:

```
function NSDEucl(M, N: Integer): Integer;
var
  R: Integer;
begin
  if M < 0 then
      raise EVal.Create('Перше число від''ємне');
    if N < 0 then
      raise EVal.Create('Друге число від''ємне');
    if (M = 0) and (N = 0) then
      raise EVal.Create('Обидва числа дорівнюють нулю');
  if M < N then begin
    R := M;
    M := N;
    N := R;
  end;
  while N > 0 do begin
    R := M mod N;
    M := N;
    N := R;
  end;
  Result := M;
```

[1] Страуструп Б. Язык программирования С++. Часть 1: Пер. с англ.– К.: ДиаСофт, 1993.– 264 с.

```
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  M, N: Integer;
begin

  try                                  //Початок захищеного блоку
    M := StrToInt(edInput1.Text);        //Тут можливі помилки
    N := StrToInt(edInput2.Text);                //перетворення
    mmOutput1.Lines.Add(IntToStr(NSDEucl(M, N)));
  except
    on E: EVal do                 //Реакція на від'ємне значення
      ShowMessage(E.Message);       //або два нульових значення
    on EConvertError do       //Реакція на помилку перетворення
      ShowMessage(' Запис числа є неправильним');
  end;                                  //Кінець захищеного блоку
end;
```

У цьому випадку правильність оброблюваних значень контролюється усередині функції `NSDEucl` обчислення найбільшого спільного дільника двох цілих чисел. У тому випадку, коли значення переданих усередину функції параметрів `M` і `N` виявляться неприпустимими (один з них менший від нуля або обидва дорівнюють нулю), один з розташованих усередині функції операторів **`raise`** збуджує виняток `EVal` і виконує вихід з функції `NSDEucl` з передачею керування опрацьовувачу винятку, що перебуває в захищеному блоці процедури `TForm1.Button1Click`. Далі робота програми продовжується звичайним чином.

Якщо функція буде викликана поза захищеним блоком, що містить опрацьовувач, здатний обробити виняток `EVal`, програма буде виконуватися так, неначебто функція `NSDEucl` не викликалася.

Підкреслимо, що для перевірки особливих ситуацій замість створення власних винятків можна скористатися умовними операторами.

Запитання для контролю і самоконтролю

1. Що таке захищений блок?
2. Який клас є прабатьком усіх винятків?
3. Які види захищених блоків існують у Delphi?
4. Як виконується захищений блок при виникненні винятку?
5. Як оформляється опрацьовувач винятку?
6. Яке призначення секції **`else`**…**`end`** у захищеному блоці?
7. У чому особливість захищеного блоку **`finally`**...**`end`**?
8. Чи може мати місце вкладеність захищених блоків?

9. Що таке стандартний виняток?
10. Як створюється власний виняток?
11. Як виконати примусове збудження винятку?

Завдання для практичного відпрацьовування матеріалу

1. Визначити клас із динамічно створюваним особистим полем – масивом цілих чисел, розмір якого за умовчанням дорівнює 10. Його конструктор за допомогою свого параметра повинен забезпечувати передачу в створюваний об'єкт розміру масиву й створення останнього. Передбачити обробку таких винятків: від'ємний, нульовий, занадто малий (менше 10) і занадто великий (більше 30000) розмір. Ініціалізувати масив порядковими номерами елементів. Збудження винятків проводити поза конструктором. Реакція на виняток – видача повідомлення.
2. Вирішити те ж саме завдання з розміщенням захищеного блоку поза конструктором, але зі збудженням винятків усередині нього.
3. Доповнити клас із завдання 1 властивістю `Index`, пов'язаною з особистим полем `FIndex`. Забезпечити збудження об'єктом винятку «Неприпустимий індекс» при виході індексу за обумовлений діапазон. Опрацьовувач винятку, як і у завданні 1, розмістити поза класом. Для перевірки працездатності програми після створення об'єкта виконати зміну знака в перших 100 елементів його масивного поля (якщо розмір масиву менше 100, повинен збуджуватися виняток).

14. ГРАФІЧНІ МОЖЛИВОСТІ DELPHI

Графічні можливості Delphi визначаються, насамперед, чотирма спеціалізованими класами – `TCanvas` (канва, полотно), `TFont` (шрифт), `TPen` (перо) і `TBrush` (пензель). Об'єкти, пов'язані з цими класами, автоматично створюються для всіх видимих елементів і є доступними через їх властивості `Canvas`, `Font`, `Pen` і `Brush`. Крім того, важливе місце займають класи `TGraphic`, `TPicture`, `TBitmap`, `TIcon`, `TMetafile` та ін.

14.1. Клас TCanvas

Клас `TCanvas` створює канву (полотно), на якій можна малювати пером, пензлем, шрифтом. Основними властивостями цього класу, насамперед, є властивості `Pixels`, `Brush`, `Pen` і `Font`. Крім того, у нього є ряд властивостей і методів, що забезпечують процес малювання.

Власні властивості класу `TCanvas` наведені в таблиці 14.1.

Таблиця 14.1 – Власні властивості класу `TCanvas`

Властивість	Опис
`property Brush: TBrush;`	Об'єкт-пензель
`property ClipRect: TRect;`	Визначає поточні розміри області, що потребує промальовування (властивість «тільки для читання»)
`property CopyMode: LongInt;`	Встановлює спосіб взаємодії растрового зображення з кольорами фону
`property Font: TFont;`	Об'єкт-шрифт

`property Handle: HDC;`
`type`
`  HDC = type LongWord;`
	Дескриптор канви. Використовується при безпосередньому звертанні до *API*-функцій Windows

Продовження табл. 14.1

Властивість	Опис
property LockCount: Integer;	Лічильник блокувань канви. Збільшується на одиницю при кожному звертанні до методу Lock і зменшується на одиницю при звертанні до Unlock (властивість «тільки для читання»)
property Pen: TPen;	Об'єкт-перо
property PenPos: TPoint;	Визначає поточне положення пера в пікселях щодо лівого верхнього кута канви
property Pixels [X, Y: Integer]: TColor;	Масив пікселів канви
property TextFlags: LongInt;	Спосіб виведення тексту методами TextOut або TextRect

Канва являє собою прямокутник, заповнений точками (пікселями), що розташовуються на однаковій відстані один від одного по горизонталі й вертикалі. Положення пікселів на канві визначається пов'язаною з нею системою координат і задається їх номерами по горизонталі (координата X) і вертикалі (координата Y) з відліком від 0. Пікселю з координатами (0, 0) відповідає лівий верхній кут канви. Кольори пікселів задаються властивістю Pixels, що являє собою двовимірний цілочисловий масив, кожен з елементів якого визначає колір відповідного пікселя. Змінюючи значення елементів масиву Pixels, можна в ході виконання програми змінювати кольори відповідних пікселів, тим самим, здійснюючи попіксельне промальовування зображення без використання пера, пензля та шрифту. Елементи масиву Pixels мають тип TColor, визначений у модулі Graphics у такий спосіб:

```
type
  TColor = -$7FFFFFFF-1..$7FFFFFFF;
```

Тип TColor використовується для задавання кольорів об'єкта. Значенням цього типу визначається властивість Color, яка є в графічних інструментів. Властивість Color визначає колірні характеристики багатьох компонентів.

У модулі Graphics визначено ряд констант для задавання кольорів. Ці константи задають або колір, що є найближчим до одного з кольорів колірної палітри операційної системи, або колір, що відповідає кольору одного з елементів екрана відповідно до установок, визначених у Панелі керування Windows. Імена та зміст більшості із цих констант наведені в таблиці 14.2.

Таблиця 14.2 – Константи для задавання кольору

Колір, що відповідає колірній палітрі		Колір, що відповідає кольору елементів екрана	
Ім'я константи	Колір	Ім'я константи	Колір
clNone	білий у Windows 9x, чорний у NT	clScrollBar	поточний колір смуг скролінга
clAqua	аквамариновий (блакитний)	clBackground	поточний колір фону Робочого стола Windows
clBlack	чорний	clActiveCaption	поточний колір смуги заголовка активного вікна
clBlue	синій (ультрамариновий)	clInactiveCaption	поточний колір смуги заголовка пасивних вікон
clCream	кремовий	clMenu	поточний колір меню
clDkGray	темно-сірий	clWindow	поточний фоновий колір вікон
clFuchsia	фуксиновий (рожевий)	clWindowFrame	поточний колір меж вікон
clGray	сірий	clMenuText	поточний колір тексту в меню
clGreen	зелений	clWindowText	поточний колір тексту у вікнах
clLime	лимонний (ясно-зелений)	clCaptionText	поточний колір тексту заголовка активного вікна
clLtGray	ясно-сірий	clActiveBorder	поточний колір меж активного вікна
clMaroon	мароновий (червонясто-коричневий)	clInactiveBorder	колір меж пасивних вікон
clMedGray	сірий середнього рівня	clAppWorkSpace	поточний колір прикладного робочого простору
clMoneyGreen	сіро-зелений	clHighlight	поточний колір фону виділеного тексту
clNavy	темно-синій	clHightlightText	поточний колір виділеного тексту
clOlive	маслиново-зелений	clBtnFace	поточний колір лицьової частини кнопки
clPurple	фіолетовий	clBtnShadow	поточний колір тіні від кнопки

Продовження табл. 14.2

Колір, що відповідає колірній палітрі		Колір, що відповідає кольору елементів екрана	
Ім'я константи	Колір	Ім'я константи	Колір
clRed	червоний	clGrayText	поточний колір затуманеного тексту
clSilver	сріблясто-сірий	clBtnText	поточний колір тексту над кнопкою
clSkyBlue	небесно-синій	clInactiveCaption Text	поточний колір тексту заголовка пасивного вікна
clTeal	бірюзовий (зеленувато-блакитний)	clBtnHighlight	поточний колір верхньої частини висунутої вперед кнопки
clWhite	білий	clDefault	колір за умовчанням для активного елемента керування
clYellow	жовтий		

Значення типу TColor являє собою чотирибайтне ціле число, кожен з байтів якого має певне призначення. Старший байт цього значення зумовлює зміст інших байтів:

1. Якщо в старший байт записане значення 0 ($00), то колір визначається змішуванням інтенсивностей червоної, синьої і зеленої його складових, що задаються значеннями, записаними в молодші байти. При цьому байт, найближчий до найстаршого, визначає інтенсивність синьої складової, а наймолодший байт – інтенсивність червоної складової. Таким чином, значення $00FF0000, $0000FF00 і $000000FF визначають відповідно чистий синій, чистий зелений та чистий червоний колір найбільшої інтенсивності, білому кольору відповідає значення $00FFFFFF, а чорному – значення $00000000.
2. Якщо старший байт містить 1 ($01), то колір визначається значенням двох молодших байтів, у які в цьому випадку записується номер однієї з логічних палітр (усього їх 65536).
3. Якщо старший байт дорівнює 2 ($02), то береться колір поточної логічної палітри, найближчий до вказаного значення.
4. Одиниця, записана в найстарший розряд значення типу TColor, завжди визначає чорний колір.

```
//Приклад 14.1
//Побудувати графік функції y=Sin(x) на інтервалі [d1, d2]
//у припущенні, що вісь X знаходиться посередині
```

```
//клієнтської області форми, а вісь Y спрямована знизу вгору.
//Графік масштабувати за розміром клієнтської області
//форми (інтервалу [d1, d2] повинна відповідати її
//ширина, а розмаху графіка по осі Y – висота).
```

Створимо форму з однією кнопкою Button1. Щоб не організовувати введення меж інтервалу, у розділі **implementation** визначимо їх як константи, наприклад, з такими значеннями:

```
const
  d1 = -3.5;                          //Ліва межа інтервалу
  d2 = 13.5;                         //Права межа інтервалу
```

У цьому ж розділі визначимо наступну процедуру побудови графіку:

```
procedure Draw;                    //Процедура виведення графіка
var
  Nx, Ny: Integer;  //Номери останнього пікселя по осях X та Y
  X, Y: Integer;                  //Координати поточного пікселя
  Max, Min, dX: Real;           //Найбільше й найменше значення
                 //функції на інтервалі та крок зміни аргументу
begin
  Nx := Form1.ClientWidth - 1;      //Останній піксель по осі X
  Ny := Form1.ClientHeight - 1;     //Останній піксель по осі Y
  dX := (d2 - d1) / Nx;      //Кількість радіан на один піксель
  Max := Sin(d1);                   //Визначаємо максимальне та
  Min := Max;                     //мінімальне значення функції
  for X := 1 to Nx do
    if Sin(d1 + X * dX) > Max then
      Max := Sin(d1 + X * dX)
    else
      if Sin(d1 + X * dX) < Min then
        Min := Sin(d1 + X * dX);
  for X := 0 to Nx do begin //Перебираємо всі пікселі по осі X
   //і для кожного з них обчислюємо значення аргументу функції
     //та координату Y пікселя, що відповідає точці на графіку
    Y := Ny - Round((Sin(d1 + X * dX) - Min) *
                    (Ny / (Max - Min)));
    Form1.Canvas.Pixels[X, Y] := clBlue;      //Змінюємо колір
  end;                                       //пікселя на синій
end;
```

Одразу після коду цієї процедури розмістимо такий опрацьовувач події OnClick кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Visible := False;                //Приховуємо кнопку
  Draw;                                      //Будуємо графік
end;
```

Принцип побудови графіка наступний. Оскільки інтервалу зміни аргументів відповідає ширина клієнтської області форми (див. п. 15.3.2), насамперед, визначається кількість радіан d, що припадають на 1 піксель по горизонталі, а також найбільше (Max) і найменше (Min) значення функції на заданому інтервалі. Далі проводиться перебирання всіх пікселів по горизонталі, перерахування їх у значення аргументу, обчислення відповідного значення функції та обчислення відповідної координати пікселя по вертикалі (змінна Y) зі зміною кольору знайденого пікселя за допомогою зміни значення властивості Pixels[X, Y] канви.

Властивості Font, Pen і Brush будуть розглянуті в наступному підрозділі.

Оскільки при зміні зображення на канві (генеруванні події OnPaint) відбувається перемальовування останньої, природною вимогою є прагнення до прискорення перемальовування, якщо воно має виконуватись на невеликій ділянці. Для цього ділянка канви, на якій дозволене перемальовування, обмежується прямокутником зі сторонами, паралельними межам форми. Цей прямокутник визначається значенням властивості ClipRect канви. Властивість ClipRect має тип, що описаний у такий спосіб:

```
type
  TRect = packed record
   case Integer of
     0: (Left, Top, Right, Bottom: Integer);
     1: (TopLeft, BottomRight: TPoint);
  end;
  TPoint = packed record
    X: LongInt;
    Y: LongInt;
  end;
```

Властивість ClipRect визначає координати лівого верхнього (Left або TopLeft.X і Top або TopLeft.Y) та правого нижнього (Right або BottomRight.X і Bottom або BottomRight.Y) кутів прямокутника, у якому дозволене перемальовування зображення на канві, причому координати вершин прямокутника задаються в системі координат компонента – власника даної канви (наприклад, для форми – у системі координат її клієнтської області).

Якщо промальовування зображення за точками здійснюється зміною властивості Pixels, то методи класу TCanvas дозволяють промальовувати зображення за допомогою пера, пензля та шрифту.

Для роботи з графічними інструментами Pen, Brush і Font у класі TCanvas передбачено низку методів, які можна розділити на такі групи:

1. Методи для роботи з пером:

- **procedure** `Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)` – креслить дугу еліпса, вписаного в прямокутник зі сторонами, паралельними межам канви, і двома кутами в точках (*X*1, *Y*1) і (*X*2, *Y*2); початок дуги знаходиться в точці перетину променя, що проходить через центр еліпса і точку (*X*3, *Y*3), а кінець дуги – на перетині променя, що проходить через центр еліпса і точку (*X*4, *Y*4);
- **procedure** `Polyline(Points: array of TPoint)` – креслить ламану, що послідовно з'єднує точки, координати яких зазначені в масиві – параметрі виклику;
- **procedure** `LineTo(X, Y: Integer)` – креслить відрізок прямої від поточного положення пера до зазначеної точки, не захоплюючи останню (перо переміщається в зазначену точку);
- **procedure** `MoveTo(X, Y: Integer)` – переміщає перо в зазначену точку без креслення лінії (цього ж можна домогтися явною зміною властивості `PenPos` пера);

2. Методи для роботи з пером та пензлем:

- **procedure** `Ellipse(X1, Y1, X2, Y2: Integer)` – креслить еліпс в охоплюючому прямокутнику (див. `Arc`) і заповнює його поточним пензлем відповідно до властивостей останнього;
- **procedure** `Ellipse(`**const** `Rect: TRect)` – креслить еліпс в охоплюючому прямокутнику, що заданий параметром типу `TRect`, і заповнює його поточним пензлем відповідно до його властивостей;
- **procedure** `Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)` – креслить дугу еліпса аналогічно методу `Arc`, з'єднуючи кінці дуги хордою і заповнюючи отриману фігуру пензлем за аналогією з методом `Ellipse`;
- **procedure** `Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)` – аналогічний методу `Chord`, але кінці дуги з'єднуються відрізками прямої з центром еліпса (будується сектор еліпса);
- **procedure** `Polygon(Points: array of TPoint)` – аналогічний методу `Polyline`, але остання точка з'єднується з першою, після чого отриманий багатокутник заповнюється поточним пензлем;
- **procedure** `Rectangle(X1, Y1, X2, Y2: Integer)` – креслить прямокутник зі сторонами, що паралельні межам канви,

й діагоналлю, заданою вершинами в точках (*X*1, *Y*1) і (*X*2, *Y*2), із заповненням його поточним пензлем;

- **procedure** Rectangle(**const** Rect: TRect) – перевантажена процедура, що відрізняється від попередньої способом задання вершин;
- **procedure** RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer) – аналогічний методу Rectangle, однак створюваний прямокутник має згладжені кути; перші 4 параметри аналогічні параметрам методу Rectangle, а параметри X3 і Y3 задають відповідно ширину і висоту прямокутника, що охоплює еліпс, дуги якого використовуються для згладжування кутів.

3. Методи для роботи з пензлем:

- **procedure** FillRect (**const** Rect: TRect) – заповнює прямокутник поточним пензлем, не зачіпаючи праву та нижню межі;
- **procedure** FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle) – заповнює поточним пензлем канву, починаючи з точки, координати якої задаються першими двома параметрами; заливання здійснюється у всіх напрямках. Якщо четвертий параметр дорівнює fsBorder, то третій параметр задає колір межі заливання (заливання припиняється на точках з таким кольором). Якщо ж четвертий параметр дорівнює fsSurface, заливання поширюється на всі сусідні точки з кольором, заданим третім параметром. Тип TFillStyle визначено у такий спосіб:

```
type TFillStyle = (fsSurface, fsBorder);
```

- **procedure** FrameRect(**const** Rect: TRect) – обводить зазначений прямокутник поточним пензлем (не пером), не здійснюючи його заливання.

4. Методи для роботи зі шрифтом:

- **procedure** TextOut(X, Y: Integer; **const** Text: **string**) – виводить рядок Text таким чином, щоб лівий верхній кут прямокутника, що його охоплює, розміщався в точці (*X*, *Y*); перо при цьому переміщається в правий верхній кут виведеного прямокутника;
- **procedure** TextRect(Rect: TRect; X, Y: Integer; **const** Text: **string**) – виводить рядок Text таким чином, щоб лівий верхній кут прямокутника, що його охоплює, розміщався в точці (*X*, *Y*); якщо рядок виходить за межі прямокутника,

заданого першим параметром, він буде обрізаний; перо не переміщується, а стиль пензля повинен бути заданий значенням `bsClear`.

5. Допоміжні методи для роботи зі шрифтом:

- **function** `TextExtent(`**const** `Text:` **string**`):` `TSize` – повертає ширину `cx` і висоту `cy` прямокутника, що охоплює рядок `Text` (у пікселях); тип значення, що повертається, визначено у такий спосіб:

```
type
  TSize = packed record
    cx: LongInt;
    cy: LongInt;
  end;
```

- **function** `TextHeight(`**const** `Text:` **string**`):` `Integer` – повертає висоту (у пікселях) прямокутника, що охоплює рядок `Text`;
- **function** `TextHeight(`**const** `Text: WideString):` `Integer` – те ж для рядка `Text` типу `WideString`;
- **function** `TextWidth(`**const** `Text:` **string**`):` `Integer` – повертає ширину (у пікселях) прямокутника, що охоплює рядок `Text` типу **string**;
- **function** `TextWidth(`**const** `Text: WideString):` `Integer` – те ж для рядка `Text` типу `WideString`.

При застосуванні методів `TextOut` і `TextRect` виведення виконується залежно від значення властивості `TextFlags`. У модулі `Graphics` визначено ряд констант, комбінуючи які можна керувати процесом виведення тексту. Зокрема, константа `ETO_CLIPPED` указує, що текст виводиться тільки в межах заданого прямокутника (прямокутник задається або явно параметром методу `TextRect`, або автоматично при використанні методу `TextOut`), а константа `ETO_OPAQUE` задає непрозорість фона виведеного тексту (у цьому випадку прискорюється процес виведення).

Подані далі приклад 14.2 і рис. 14.1 ілюструють використання методів `Arc`, `Pie`, `Chord`, `Polyline`, `RoundRect`, `Ellipse`, `LineTo`, `MoveTo` і деяких інших методів.

Крім малювання за допомогою графічних інструментів, на канву можна копіювати деяке зображення. Властивість `CopyMode` визначає методику взаємодії двох зображень (джерела і приймача) при копіюванні і задається одним з таких значень:

- `cmBlackness` – заповнює область малювання чорним кольором;
- `cmDstInvert` – заповнює область малювання інверсним кольором фону;
- `cmMergeCopy` – поєднує зображення на канві та зображення, що копіюється, побітовою операцією **`and`**;
- `cmMergePaint` – поєднує зображення на канві та зображення, що копіюється, побітовою операцією **`or`**;
- `cmNotSrcCopy` – копіює на канву інверсне зображення джерела;
- `cmNotSrcErase` – поєднує зображення на канві і зображення, що копіюється, побітовою операцією **`or`** та інвертує отримане зображення;
- `cmPatCopy` – копіює зразок джерела;
- `cmPatInvert` – комбінує зразок джерела із зображенням на канві за допомогою побітової операції **`xor`**;
- `cmPatPaint` – комбінує зображення джерела з його зразком за допомогою побітової операції **`or`**, потім отриманий результат об'єднується із зображенням на канві також за допомогою **`or`**;
- `cmSrcAnd` – поєднує зображення джерела і канви за допомогою побітової операції **`and`**;
- `cmSrcCopy` – копіює зображення джерела на канву;
- `cmSrcErase` – інвертує зображення на канві і за допомогою побітової операції **`and`** поєднує результат із зображенням джерела;
- `cmSrcInvert` – поєднує зображення на канві та джерело побітовою операцією **`xor`**;
- `cmSrcPaint` – поєднує зображення на канві та джерело побітовою операцією **`xor`**;
- `cmWhiteness` – заповнює область малювання білим кольором;
- `cmCreateMask` – застосовує джерело для видалення всіх частин зображення, які з'являються як колір `clDontMask` у джерелі.

Якщо значення властивості `CopyMode` не задане явно, то воно задається за умовчанням рівним `cmSrcCopy`.

Для копіювання зображення і промальовування графічних об'єктів використовуються такі методи канви:

- **`procedure`** `BrushCopy(`**`const`** `Dest: TRect; Bitmap: TBitMap;` **`const`** `Source: TRect; Color: TColor)` – прямокутний фрагмент зображення `Bitmap`, що задається третім параметром (див. п. 14.3.1), копіюється в заданий першим параметром прямокутник поточної канви із заміною кольором пензля

одного з кольорів скопійованого зображення (задається четвертим параметром). У довідковій системі Delphi замість цього методу рекомендується застосовувати клас `TImageList`;

- **`procedure`** `CopyRect(`**`const`** `Dest: TRect; Canvas: TCanvas;` **`const`** `Source: TRect)` – з канви `Canvas` у прямокутник `Dest` поточної канви копіюється прямокутний фрагмент зображення `Source` (при цьому, якщо буде потреба, відбувається стиснення або розтягування зображення);
- **`procedure`** `Draw(X, Y: Integer; Graphic: TGraphic)` – промальовує графічний об'єкт (піктограму, метафайл, растрове зображення або JPEG-зображення), заданий третім параметром, таким чином, щоб його верхній лівий кут збігся з пікселем, координати якого задані першими двома параметрами (при копіюванні растрових зображень можна використовувати різні значення властивості `CopyMode`);
- **`procedure`** `StretchDraw(`**`const`** `Rect: TRect; Graphic: TGraphic)` – аналог методу `Draw`, однак при цьому графічний об'єкт розміщається в прямокутнику, визначеному другим параметром, з масштабуванням виведеного зображення за розміром прямокутника.

Для відновлення в канві пера, пензля та шрифту за умовчанням використовується метод

`Refresh`,

що є процедурою без параметрів.

Зазначимо, що канва, крім перелічених вище властивостей і методів, має ще ряд інших.

14.2. Графічні інструменти

14.2.1. Клас TPen

Даний клас служить для створення об'єкта-пера, призначеного для креслення ліній і контурів на канві. Клас має такі властивості:

- **`property`** `Color: TColor` – визначає колір ліній, що кресляться пером;
- **`property`** `Mode: TPenMode` – визначає те, як колір ліній, що кресляться пером, взаємодіє з кольором фону на канві (визначення типу `TPenMode` і зміст можливих значень даної властивості див. нижче);

- **property** Style: TPenStyle – визначає стиль (вид) ліній (визначення типу TPenStyle і зміст можливих значень даної властивості див. нижче);
- **property** Width: Integer – товщина ліній у пікселях.

Тип TPenMode визначено як перелічений тип, можливі значення якого (у порядку зростання) наведені в таблиці 14.3.

Таблиця 14.3 – Режими креслення

Режим	Значення
pmBlack	Завжди чорні лінії
pmWhite	Завжди білі лінії
pmNop	Невидимі лінії
pmNot	Інверсія кольору фону канви
pmCopy	Колір ліній визначається значенням властивості Color
pmNotCopy	Інверсія кольору пера
pmMergePenNot	Комбінація кольору пера й інверсного кольору фону канви
pmMaskPenNot	Комбінація спільних кольорів для пера й інверсного кольору фону канви
pmMergeNotPen	Комбінація кольору фону канви й інверсного кольору пера
pmMaskNotPen	Комбінація спільних кольорів для кольору фону канви й інверсного кольору пера
pmMerge	Комбінація кольорів пера й фону канви
pmNotMerge	Інверсний щодо pmMerge колір
pmMask	Комбінація спільних кольорів для пера й фону канви
pmNotMask	Інверсний щодо pmMask колір
pmXor	Комбінація кольорів пера й фону операцією **xor**
pmNotXor	Інверсний щодо pmXor колір

Зазначимо, що для режимів pmNotCopy, pmMaskPenNot, pmMaskNotPen і pmNotMerge властивість Style ігнорується, а для режимів pmBlack, pmWhite і pmNot ігноруються властивості Style і Color.

Тип TPenStyle визначено як перелічений тип, можливі значення якого (у порядку зростання) наведені в таблиці 14.4.

Властивості класу TPen за умовчанням мають такі значення: Color=clBlack, Mode=pmCopy, Style=psSolid, Width=1.

14.2.2. Клас TBrush

Клас TBrush визначає колір і зразок для заповнення (заливання) замкнутих фігур. Він має такі властивості:

- **property** Bitmap: TBitMap – служить для запису растрового зображення, що використовується при заповненні пензлем (якщо значення цієї властивості визначене);
- **property** Color: TColor – колір пензля;
- **property** Style: TBrushStyle – визначає стиль пензля (стиль заповнення).

Таблиця 14.4 – Стилі ліній

Стиль	Значення
psSolid	Суцільна лінія (значення за умовчанням)
psDash	Штрихова лінія
psDot	Лінія із крапок (пунктирна)
psDashDot	Штрих-пунктирна лінія
psDashDotDot	Штрих-пунктир-пунктирна лінія
psClear	Відсутність лінії
psInsideFrame	Суцільна лінія

Якщо значення властивості Bitmap визначене, то значення властивостей Color і Style ігноруються.

Тип TBrushStyle, що визначає стиль заповнення, є переліченим типом з наступними можливими значеннями (у порядку їх збільшення):

- ✓ bsSolid – суцільне заливання кольором, визначеним значенням властивості Color;
- ✓ bsClear – заливання кольором фону;
- ✓ bsBDiagonal – заповнення діагональними лініями, що йдуть зліва направо, знизу вгору;
- ✓ bsFDiagonal – заповнення діагональними лініями, що йдуть справа наліво, зверху вниз;
- ✓ bsCross – заповнення вертикальною сіткою;
- ✓ bsDiagCross – заповнення діагональною сіткою;
- ✓ bsHorizontal – заповнення горизонтальними лініями;
- ✓ bsVertical – заповнення вертикальними лініями.

За умовчанням властивість Color має значення clWhite, а властивість Style – значення bsSolid.

Якщо властивості Style присвоєне значення bsClear, значення властивості Color ігнорується, але в неї записується значення clWhite. Надалі при зміні значення властивості Style властивість Color буде мати значення clWhite (якщо воно не буде змінене явно).

Методи класу TBrush не розглядатимуться.

14.2.3. Клас TFont

При виведенні текстових повідомлень за допомогою графічних пристроїв (екрана, принтера тощо) використовуються об'єкти класу `TFont`, що описує такі характеристики шрифту, як ім'я, розмір, стиль і т. д.

Клас має наступні властивості:

- **property** `Charset: TFontCharset` – номер набору символів;
- **property** `Color: TColor` – колір шрифту;
- **property** `Height: Integer` – висота шрифту в екранних пікселях;
- **property** `Name: TFontName` – ім'я шрифту (за умовчанням визначене значенням `Sans Serif`);
- **property** `Pitch: TFontPitch` – спосіб розташування літер у тексті (тип `TFontPitch` – це перелічений тип з такими значеннями: `fpDefault` – символи мають ширину за умовчанням, що залежить від поточного шрифту; `fpFixed` – всі символи мають однакову ширину; `fpVariable` – ширина символів залежить від їх накреслення);
- **property** `PixelsPerInch: Integer` – кількість екранних пікселів на 1 дюйм;
- **property** `Size: Integer` – висота шрифту в пунктах (1 пункт = 1/72 дюйма ≈ 0,36 мм);
- **property** `Style: TFontStyles` – стиль шрифту.

Зупинимося на особливостях деяких із зазначених властивостей.

Властивості `Charset` за умовчанням присвоюється значення `DEFAULT_CHARSET`. При цьому для україномовних (або російськомовних) програм можна встановлювати значення `RUSSIAN_CHARSET`, а для відображення текстів MS DOS в альтернативному кодуванні – значення `OEM_CHARSET`.

Властивості `Height` і `Size` пов'язані одна з одною. Зміна значення однієї з них спричиняє автоматичну зміну іншої. Зв'язок між цими властивостями визначається співвідношенням

```
Font.Size = -Font.Height * 72 / Font.PixelsPerInch.
```

Значення ж властивості `PixelsPerInch` змінювати не рекомендується, бо воно стосується шрифтів принтера.

Тип `TFontStyles` властивості `Style` визначено у такий спосіб:

```
type
  TFontStyle = (fsBold,fsItalic,fsUnderline,fsStrikeOut);
  TFontStyles = set of TFontStyle;
```

Значення типу `TFontStyle` мають наступний зміст:

- ✓ `fsBold` – жирний шрифт;
- ✓ `fsItalic` – похилий шрифт (курсив);
- ✓ `fsUnderline` – підкреслений однією прямою лінією шрифт;
- ✓ `fsStrikeOut` – перекреслений однією прямою лінією шрифт.

Сам по собі стиль шрифту визначається комбінацією цих значень у вигляді множини. Наприклад, стиль `[fsBold, fsUnderline, fsItalic]` відповідає жирним похилим перекресленим символам. За умовчанням властивість `Style` має значення `[]`, що відповідає звичайним символам. Це ж значення треба присвоювати властивості `Style`, якщо необхідно перейти до звичайних символів після використання іншого стилю.

У прикладі, що наводиться нижче, ілюструється креслення різного роду фігур за допомогою методів класу `TCanvas` із зазначенням місця розташування точок, координати яких визначають процес креслення. У прикладі також показано, яким чином виконується зміна кольору ліній, кольору шрифту, стилів ліній, пензля та шрифту, визначення розміру області, у якій виводиться текст, а також безпосередньо виведення тексту.

```
//Приклад 14.2
//Зобразити дугу та сектор еліпса, а також усічений еліпс
//і прямокутник зі згладженими кутами. Зобразити також
//прямокутники, що охоплюють названі фігури, і точки
//(у вигляді кіл малого радіуса), що визначають процес
//рисування. Накреслити еліпс, що визначає згладжування кутів
//прямокутника, і підписати рисунки. Проілюструвати роботу
//зі стилями та кольором.
```

Для розв'язання задачі скористаємося формою з прикладу 14.1, а в секції **`implementation`** модуля опишемо константи (щоб уникнути введення даних):

```
const
  X1 = 20;  Y1 = 40;      //Лівий верхній кут прямокутника
  X2 = 250; Y2 = 140;     //Правий нижній кут прямокутника
  X3 = 220; Y3 = 30;    //Точка, що визначає початок дуги
  X4 = 160; Y4 = 100;     //Точка, що визначає кінець дуги
  X3R = 100; Y3R = 80; //Розміри еліпса, що згладжує кути
```

Скористаємося наступним опрацьовувачем події OnClick кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  L: Integer;
begin
  Button1.Visible := False;
  with Canvas do begin
    Font.Name := 'Arial Narrow';                    //Вибір шрифту
    Pen.Width := 2;                        //Товщина пера 2 пікселя
  //Дуга, сектор, "хорда", згладжений прямокутник
    Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4);
    Pie(X1+240,Y1, X2 + 240, Y2, X3 + 240, Y3, X4 + 240, Y4);
    Chord(X1, Y1+170, X2, Y2+170, X3, Y3+170, X4, Y4+170);
    RoundRect(X1+240, Y1+170, X2+240, Y2+170, X3R, Y3R);
                         //Підписи під рисунками різними стилями
    L := TextWidth('Метод Arc');                  //Довжина підпису
    TextOut(X1 + (X2 - X1 - L) div 2, Y2 + 15, 'Метод Arc');
    Font.Style := [fsBold];                        //Стиль "жирний"
    L := TextWidth('Метод Pie');                  //Довжина підпису
    TextOut(X1+240+(X2-X1-L) div 2, Y2+15, 'Метод Pie');
    Font.Style := [fsBold, fsItalic];            //"Жирний, курсив"
    L := TextWidth('Метод Chord');                //Довжина підпису
    TextOut(X1+(X2-X1-L) div 2, Y2+170+15, 'Метод Chord');
                  //Нижче стиль "жирний, курсив, підкреслений"
    Font.Style := [fsBold, fsItalic, fsUnderline];
    L := TextWidth('Метод RoundRect');            //Довжина підпису
                                        //Змінюємо колір шрифту
    Font.Color := clNavy;
    TextOut(X1 + 240 + (X2 - X1 - L) div 2, Y2 + 170 + 15,
            'Метод RoundRect');
                                    //Відновлюємо звичайний стиль
    Font.Style := [];
                                                  //і чорний колір
    Font.Color := clBlack;
    Pen.Width := 1;                 //Далі товщина ліній 1 піксель
                         //Малюємо прямокутник у вигляді ламаної
    Polyline([Point(X1, Y1), Point(X2, Y1),
              Point(X2, Y2), Point(X1, Y2), Point(X1, Y1)]);
    Brush.Style := bsClear;   //Стиль пензля - заливання фоном
    Pen.Style := psDash;                //Стиль пера - "штриховий"
                       //Креслення еліпса усередині прямокутника
    Ellipse(X1, Y1, X2, Y2);
    Pen.Style := psDashDot;      //Стиль пера - "штрих-пунктир"
    MoveTo(X3, Y3);                               //Переміщення пера
              //Нижче зображуються два відрізки, напрямки яких
            //визначали креслення дуги методом Arc (див. вище)
    LineTo((X1 + X2) div 2, (Y1 + Y2) div 2);
                                             //Перо перемістилося
    LineTo(X4, Y4);
```

```
    TextOut(X1 - 18, Y1 - 15,'(X1,Y1)');
    TextOut(X2 - 18, Y2 + 3,'(X2,Y2)');
    TextOut(X3 - 18, Y3 - 15,'(X3,Y3)');
    TextOut(X4 + 5, Y4 - 15, '(X4,Y4)');
    Brush.Color := clBlack;            //Колір пензля – чорний
  //Точки зображуємо як чорні кола радіусом 2
    Ellipse(X1 - 2, Y1 - 2, X1 + 2, Y1 + 2);
    Ellipse(X2 - 2, Y2 - 2, X2 + 2, Y2 + 2);
    Ellipse(X3 - 2, Y3 - 2, X3 + 2, Y3 + 2);
    Ellipse(X4 - 2, Y4 - 2, X4 + 2, Y4 + 2);
           //Креслимо прямокутник навколо еліпса, що визначає
                                    //форму згладжування кутів
    Polyline([Point(X1 + 240,Y1 + 170), Point(X1 + 240 + X3R,
             Y1 + 170), Point(X1 + 240 + X3R, Y1 + 170 + Y3R),
             Point(X1 + 240, Y1 + 170 + Y3R),
             Point(X1 + 240, Y1 + 170)]);
         //Точки, що визначають процес роботи методу RoundRect
    Ellipse(X1 + 240 - 2, Y1 + 170 - 2,
           X1 + 240 + 2, Y1 + 170 + 2);
    Ellipse(X2 + 240 - 2, Y2 + 170 - 2,
           X2 + 240 + 2, Y2 + 170 + 2);
    Ellipse(X1 + 240 + X3R - 2, Y1 + 170 + Y3R - 2,
           X1 + 240 + X3R + 2, Y1 + 170 + Y3R + 2);
                                    //Змінюємо колір пензля
    Brush.Color := clBtnFace;
                       //Нижче дуга еліпса, що визначає процес
                            //згладжування кутів прямокутника
    Arc(X1 + 240, Y1 + 170, X1 + 240 + X3R, Y1 + 170 + Y3R,
        X1 + (240 + X3R) div 2, Y1 + 170,
        X1 + 240, Y1 + (170 + Y3R) div 2);
                          //Виводимо координати вузлових точок
                               //для згладженого прямокутника
    TextOut(X1 + 240, Y1 + 170 - 15, '(X1,Y1)');
    TextOut(X2 + 240 - 25, Y2 + 170 + 5, '(X2,Y2)');
    TextOut(X1 + 240 + X3R + 5, Y1 + 170 + Y3R - 2,
           '(X1+X3,Y2+Y3)');
  end;
end;
```

Результати роботи програми наведені на рис. 14.1.

З метою скорочення тексту програми в ній використано оператор **with** для забезпечення доступу до властивостей і методів об'єкта `Canvas`. Якщо не застосовувати оператор **with**, то у всіх операторах, що звертаються до властивостей і методів класу `TCanvas`, необхідно приєднати ім'я екземпляра класу (наприклад, `Canvas.Pen.Width := 2`).

У програмі, що наведена вище, параметр методу `Polyline` задавався у вигляді константного масиву. Оскільки елементи цього масиву

повинні мати тип `TPoint`, а координати точок задавалися парами цілих чисел, для одержання значень типу `TPoint` було вжито функцію `Point`, визначену в модулі `Types` у такий спосіб:

```
function Point(AX, AY: Integer): TPoint;
```

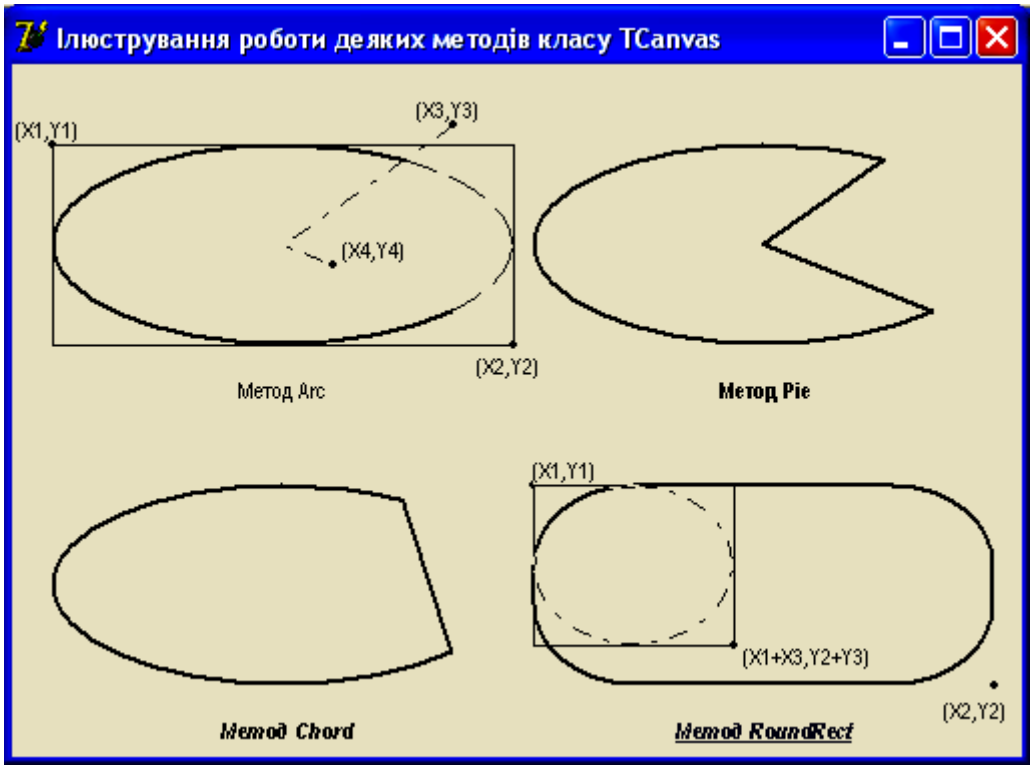


Рис. 14.1. Робота з графічними інструментами

14.3. Класи для роботи із зображеннями

14.3.1. Класи TGraphic, TIcon, TBitmap, TMetafile та TJPEGImage

Графічні можливості Delphi дозволяють працювати з готовими графічними зображеннями із завантаженням їх з файлу, буфера міжпрограмного обміну або потоку. Базовим класом для цього є описаний у модулі `Graphics` абстрактний клас `TGraphic`, безпосередніми нащадками якого є такі класи:

- `TIcon` (піктограма) – растрове зображення невеликого розміру із засобами, що керують його прозорістю (звичайно зберігається у файлі зі стандартним розширенням .ICO);
- `TBitmap` (растрове зображення) – довільне растрове зображення, що зберігається у файлі зі стандартним розширенням .BMP;

- `TMetafile` (метафайл) – зображення, побудоване на графічному пристрої за допомогою спеціальних команд і збережене у файлі з розширенням .EMF (розширений метафайл Win32) або .WMF (метафайл Windows 3.1 із заголовком);
- `TImage` (JPEG-зображення) – зображення у форматі так званої об'єднаної групи фотографічних експертів (Joint Photographic Expert Group), яке зберігається у файлі зі стандартним розширенням .JPG.

Класи `TIcon`, `TBitmap` і `TMetafile` оголошені в модулі `Graphics`, а клас `TJPEGImage` – у модулі `Jpeg`, який потрібно підключати спеціально.

Працюючи з об'єктами класу `TGraphic`, варто не забувати про необхідність попереднього їхнього створення, для чого використовується метод `Create`, успадкований класом `TGraphic` по ланцюжку `TObject` – `TPersistent` – `TInterfacedPersistent` від класу `TObject`.

Клас `TGraphic` має такі властивості:

- **property** `Empty: Boolean` – якщо з екземпляром класу не зв'язане графічне зображення, у цій властивості, що є властивістю «тільки для читання», міститься значення `True`;
- **property** `Height: Integer` – висота графічного зображення в пікселях;
- **property** `Modified: Boolean` – у властивість записується значення `True`, якщо графічне зображення було змінене (якщо `Modified=False`, то це свідчить про безглуздість збереження графічного зображення у файлі). Властивість змінюється тільки для об'єктів класу `TBitmap`, але не змінюється для об'єктів класів `TIcon` і `TMetafile` навіть у випадку їхньої зміни;
- **property** `Palette: HPALETTE` – містить номер колірної палітри графічного об'єкта або значення `0`, якщо об'єкт не використовує палітру (тип `HPALETTE` є типом `LongWord`);
- **property** `PaletteModified: Boolean` – у цю властивість записується значення `True`, якщо в об'єкта змінилася колірна палітра;
- **property** `Transparent: Boolean` – значення вказує, що об'єкт прозорий для фону, на якому він виводився (об'єкти класів `TIcon` і `TMetafile` завжди прозорі, у той час як для об'єктів класу `TBitmap` зміна значення цієї властивості має сенс);

- **property** Width: Integer – ширина графічного зображення в пікселях.

Власні методи класу TGraphic такі:

- **procedure** LoadFromClipboardFormat(AFormat: Word; AData: THandle; APalette: HPALETTE) – абстрактний метод, що шукає в буфері міжпрограмного обміну формат AFormat (він повинен бути зареєстрований) і при позитивному результаті пошуку заміняє поточне зображення зображенням AData, що міститься у буфері, з палітрою APalette (тип THandle є типом LongWord);
- **procedure** LoadFromFile(**const** FileName: **string**) – читає файл FileName і завантажує з нього зображення;
- **procedure** LoadFromStream(Stream: TStream) – абстрактний метод, що завантажує зображення з потоку Stream;
- **procedure** SaveToClipboardFormat(**var** AFormat: Word; **var** AData: THandle; **var** APalette: HPALETTE) – абстрактний метод, що поміщає зображення AData і його палітру APalette у буфер міжпрограмного обміну, застосовуючи формат AFormat, який попередньо повинен бути зареєстрований;
- **procedure** SaveToFile(**const** FileName: **string**) – записує зображення у файл FileName;
- **procedure** SaveToStream(Stream: TStream) – записує зображення в потік Stream.

Оскільки клас TGraphic є абстрактним, найчастіше працюють з об'єктами класів TIcon, TBitmap, TMetafile або TJPEGImage, що є безпосередніми нащадками класу TGraphic. У цих класах здійснюється перекриття абстрактних методів класу TGraphic власними методами, що забезпечують урахування особливостей відповідних форматів зображень.

Наступний приклад демонструє варіант використання в програмі класу TBitmap при заповненні областей за допомогою пензля.

```
//Приклад 14.3
//Продемонструвати заливання областей растровими
//зображеннями, що зберігаються в папці C:\WINDOWS.
```

Помістимо на порожній формі в правій її частині панель шириною, достатньою для розміщення на ній кнопки. Очистимо властивість Caption панелі, присвоїмо її властивості Align значення alRight і розташуємо в середній частині панелі кнопку Button1. Для компонента Button1 створимо такий опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FRec: TSearchRec;
  i, Code: Integer;
  BitMap: TBitmap;
begin               //Властивість Tag містить кількість раніше
                              //переглянутих файлів (спочатку 0)
      //Створюємо об'єкт - "растрове зображення"
  BitMap := TBitmap.Create;
  Canvas.Brush.Bitmap := BitMap;     //Пензель буде виконувати
                            // заповнення растровим зображенням
                 //Шукаємо BMP-файл
  Code := FindFirst('C:\WINDOWS\*.bmp', faAnyFile, FRec);
  if Code = 0 then begin                 //Якщо файл знайдений,
    for i := 0 to Tag - 1 do                  //пропускаємо вже
      Code := FindNext(FRec);               //переглянуті файли
    if Code = 0 then begin
                          //і завантажуємо файл в об'єкт BitMap
      BitMap.LoadFromFile('C:\Windows\' + FRec.Name);
      Canvas.Rectangle(0, 0, Form1.ClientWidth -      //Зразок
         Panel1.Width, Form1.ClientHeight);       //заповнення
      Tag := Tag + 1;                   //Змінюємо значення Tag
    end
    else begin
      BitMap.Free;                     //Руйнуємо об'єкт "растр"
      Canvas.Brush.Bitmap := nil;     //Тепер пензель звичайний
      Form1.Caption := 'Більше немає зображень';
      Button1.Hide;
    end;
  end
  else begin
    Canvas.TextOut(Form1.ClientHeight div 2,
          (Form1.ClientWidth - Panel1.Width -
          Canvas.TextWidth('Немає зображень')) div 2,
          'Немає зображень');
    Button1.Hide;
  end;
end;
```

Підкреслимо, що використання змінної `BitMap` не обов'язкове, оскільки можна було б безпосередньо створювати об'єкт `Canvas.Brush.Bitmap`, з яким, власне, і проводиться робота. Для цього у виконуваній частині процедури `TForm1.Button1Click` необхідно перші два оператори замінити оператором

```
    Canvas.Brush.Bitmap := TBitmap.Create,
```

а оператор

```
    BitMap.LoadFromFile('C:\Windows\' + FRec.Name)
```

замінити оператором

```
    Canvas.Brush.Bitmap.LoadFromFile('C:\Windows\'+FRec.Name).
```

Зазначимо також, що тільки в ілюстративних цілях у текст програми включені оператори `BitMap.Free` та `Canvas.Brush.Bitmap:=nil`, оскільки вони будуть виконані безпосередньо перед завершенням роботи програми і не впливають на її виконання.

Наведений нижче приклад орієнтований на застосування в програмі класу `TBitmap`, а також зміну в програмі растрового зображення та збереження його у файлі.

```
//Приклад 14.4
//У поточній папці знаходиться файл Image.bmp. Якщо зображення
//не є квадратним, перекреслити його по діагоналях прямими
//лініями і зберегти у файлі Image1.bmp. Вивести зображення
//прозорим на форму.
```

Скористаємося формою з прикладу 14.3 і використаємо такий опрацьовувач події OnClick для кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BitMap: TBitmap;
begin
                                  //Створюємо об'єкт - "растрове
  BitMap := TBitmap.Create;                              //зображення"
                                 //і завантажуємо файл
  BitMap.LoadFromFile('Image.bmp');
  BitMap.Modified := False;//Зображення поки ще не змінювалося
  with BitMap do
                               //Height, Width, Canvas для BitMap
    if Height <> Width then begin          //Перевірка розмірів
      Canvas.LineTo(Width, Height);                //і креслення
      Canvas.MoveTo(0, Height);
      Canvas.LineTo(Width, 0);
    end;
  if BitMap.Modified = True then            //Була модифікація?
    BitMap.SaveToFile('Image1.bmp');                //Збереження
  BitMap.Transparent := True;              //Зображення прозоре
  Canvas.Draw(0, 0, BitMap);
  BitMap.Free;                               //Знищення об'єкта
end;
```

Зазначимо, що тут при кресленні ліній на зображенні автоматично змінюється значення його властивості `Modified`, яка відразу ж після зчитування зображення з файлу отримала значення `False`. Завдяки цьому

зображення у файл буде виводитись тільки за умови, воно було змінене при виконанні програми.

14.3.2. Клас TPicture

Даний клас містить засоби з роботи із графічними зображеннями у вигляді піктограми, метафайлу, растра або JPEG-зображення. Основні властивості цього класу такі:

- **property** `Bitmap: TBitmap` – містить растрове зображення;
- **property** `Graphic: TGraphic` – містить будь-який із чотирьох графічних об'єктів – піктограму, метафайл, растр або JPEG-зображення;
- **property** `Height: Integer` – висота графічного зображення в пікселях (властивість «тільки для читання»);
- **property** `Icon: TIcon` – містить піктограму;
- **property** `Metafile: TMetafile` – містить метафайл;
- **property** `Width: Integer` – ширина графічного зображення в пікселях (властивість «тільки для читання»).

Об'єкти класу `TPicture` можуть використовуватися графічними методами, що як параметр мають параметр типу `TGraphic` або похідного від нього типу.

При роботі з екземплярами класу `TPicture` застосовуються такі його найпростіші методи:

- **procedure** `Assign(Source: TPersistent)` – зв'язує власний графічний об'єкт (властивість `Graphic`) з об'єктом `Source`;
- **procedure** `LoadFromFile(`**const** `FileName:` **string**`)` – див. однойменний метод у `TGraphic`;
- **procedure** `SaveToFile(`**const** `FileName:` **string**`)` – див. однойменний метод у `TGraphic`.

Характерною рисою даного класу є те, що при завантаженні за допомогою методу `LoadFromFile` зображення з файлу воно поміщається у властивість `Graphic`, а одна з властивостей `Bitmap`, `Icon`, `Metafile` також отримує своє значення відповідно до характеристик графічного об'єкта, що міститься у властивості `Graphic`. Далі при виведенні зображення можна використовувати як властивість `Graphic`, так і ту з властивостей `Bitmap`, `Icon`, `Metafile`, що пов'язана з графічним зображенням. Аналогічно тому, як на формат зображення у файлі налаштовується метод `LoadFromFile`, відбувається налаштування методу

SaveToFile при виведенні у файл зображення, що міститься у властивості Graphic.

```
//Приклад 14.5
//Вивести по черзі файли IMAGE.BMP, IMAGE.ICO, IMAGE.WMF,
//IMAGE.JPG, що знаходяться у поточному каталозі.
```

Як і при розв'язанні попередньої задачі, скористаємося формою з прикладу 14.3. У початок розділу **implementation** модуля додамо опис

```
var
  Picture: TPicture;
```

Підключимо модуль Jpeg і створимо опрацьовувач події OnCreate для форми, а також опрацьовувач події OnClick для кнопки Button1:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Picture := TPicture.Create;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  case Tag of
    0: Picture.LoadFromFile('IMAGE.BMP');
    1: Picture.LoadFromFile('IMAGE.ICO');
    2: Picture.LoadFromFile('IMAGE.WMF');
    3: Picture.LoadFromFile('IMAGE.JPG');
  end;
  if Tag = 3 then Tag := 0
  else Tag := Tag + 1;
  Canvas.Draw(0, 0, Picture.Graphic);
end;
```

На початку роботи програми процедура TForm1.FormCreate створює об'єкт Picture, а безпосереднє виведення зображень проводиться процедурою TForm1.Button1Click. Оскільки в цьому випадку зображення завантажуються в екземпляр класу TPicture, метод Picture.LoadFromFile автоматично налаштовується на тип зображення, що завантажується з файлу, після чого метод Canvas.Draw виконує промальовування зображення, яке зберігається у властивості Picture.Graphic. Звертання до цієї властивості при виведенні забезпечує правильне виведення без звертання до властивостей Bitmap, Icon, Metafile.

Виведення растра, піктограми або метафайлу можливе за допомогою звертання до властивостей Bitmap, Icon, Metafile відповідно. Наприклад, це можна зробити за допомогою оператора

```
Canvas.Draw(0, 0, Picture.Bitmap).
```

Виведення ж JPEG-зображення потрібно виконувати за допомогою звертання до властивості `Picture.Graphic`.

Зазначимо також, що підключення модуля `Jpeg` необхідне тільки для забезпечення можливості роботи з JPEG-зображенням; якщо робота з JPEG-зображеннями не передбачається, цей модуль підключати непотрібно.

Запитання для контролю і самоконтролю

1. Для чого служить клас `TCanvas`?
2. Назвіть основні властивості класу `TCanvas`.
3. Яким чином задається колір у Delphi?
4. Як здійснюється попіксельна робота?
5. Які стандартні фігури можна зобразити в Delphi?
6. Який метод служить виключно для переміщення пера без креслення ліній?
7. Як можна зобразити відрізок прямої?
8. За допомогою яких засобів можна зобразити інші стандартні для Delphi фігури?
9. Як можна вивести текст на канві?
10. Як взнати ширину й висоту виведеного тесту?
11. Яке призначення класу `TPen`?
12. За допомогою яких засобів виконується керування виглядом ліній, що накреслюються?
13. Яке призначення класу `TBrush`?
14. Для чого служить клас `TFont`?
15. Охарактеризуйте основні можливості класу `TFont`.
16. Які засоби існують у Delphi для роботи з готовими графічними зображеннями?
17. Зображення яких графічних форматів можна виводити в Delphi?
18. Яке призначення класу `TPicture`?
19. Які засоби є в класі `TPicture`?

Завдання для практичного відпрацьовування матеріалу

1. Нехай дві точки задані своїми декартовими координатами (x_1, y_1) та (x_2, y_2). Пряма, що проходить через них, описується такими параметричними рівняннями: $x = x_1 + (x_2 - x_1)t$, $y = y_1 + (y_2 - y_1)t$.

Якщо $0 < t < 1$, то точка (x, y) лежить усередині відрізка, ділячи його у відношенні $t/(1-t)$. При $t = 0$ й $t = 1$ точка лежить на одному з кінців відрізка, а при $t < 0$ й $t > 1$ – перебуває поза відрізком. Дано натуральні числа x_1, y_1, x_2, y_2 і дійсне число μ $(0 \le \mu < 1)$. Побудувати відрізок з координатами (x_1, y_1), (x_2, y_2) і точку, що ділить відрізок у відношенні $\mu/(1-\mu)$.

2. Дано натуральне число *n*. На інтервалі [–3, 3] побудувати графіки функції $\sqrt[3]{(x+2)^2} - \sqrt[3]{(x-2)^2}$, вивівши *n* точок. Зобразити й підписати координатні осі.
3. Дано натуральні числа x_1, y_1, *h*, *w*, *r*, x_2, y_2. Побудувати прямокутник із центром у точці (x_1, y_1), висотою *h* і шириною *w*, а також коло радіусом *r* із центром у точці (x_2, y_2). З'єднати відрізком прямої центри кола та прямокутника.
4. Виконати попереднє завдання за умови, що частини відрізка, розташовані усередині фігур, повинні бути невидимими.
5. Дано два прямокутники зі сторонами, паралельними межам форми. Кожен із прямокутників задається координатами двох його несуміжних вершин (припускається, що задані координати дозволяють розмістити прямокутники на формі). Чи правильно, що прямокутники перетинаються, причому перший з них розташовується правіше й нижче від другого. При позитивній відповіді вивести зображення прямокутників, зафарбувавши їх спільну частину кольором, що задається випадково.
6. Реалізувати «електронний калейдоскоп». Побудову калейдоскопа виконувати в такий спосіб. У центрі канви має бути зображений правильний шестикутник, вершини якого з'єднані з його центром. Один із трикутників повинен бути розсічений декількома прямими, кількість і розташування яких вибирається за допомогою датчика випадкових чисел. Після цього зображення в кожному наступному трикутнику (при русі за або проти годинникової стрілки) потрібно отримувати симетричним відображенням зображення, сформованого раніше в кожному попередньому трикутнику, відносно їх спільної сторони.
7. Виконати попереднє завдання із зафарбовуванням отриманих частин першого трикутника випадково обраним кольором і побудовою з урахуванням цього зображень у наступних трикутниках.

Див. також завдання 4–8 після розд. 17.

15. ПОНЯТТЯ КОМПОНЕНТІВ ТА ХАРАКТЕРИСТИКА ЇХ БАЗОВИХ КЛАСІВ

15.1. Класи TComponent і TPersistent

Звичайно під компонентом розуміють деякий елемент, що має свою функціональність і може бути розміщений програмістом на формі на етапі проектування програми. У певному розумінні в працюючій програмі компоненти являють собою об'єкти «реального світу»: їх можна переміщати за допомогою миші або клавіш, вони реагують на натискання клавіш і кнопок миші. Використання компонентів забезпечує як наочність і зручність роботи при проектуванні програми (насамперед її інтерфейсної частини), так і зручність її експлуатації. Основна різниця між компонентами та об'єктами інших класів полягає в тому, що компонентами можна маніпулювати на формі, а об'єктами ні. Формально ***компонент*** – це будь-який клас, породжений від класу `TComponent` або його нащадка.

Клас `TComponent` є нащадком класу `TPersistent`, який, у свою чергу, є прямим нащадком класу `TObject` (див. рис. 15.1).

Як це видно з рис. 15.1, усі класи бібліотеки класів VCL можна розбити на ***потокові*** (тобто ті, які, здатні зберігати свій стан на якому-небудь носії інформації та відновлювати його) і ***непотокові***. Компоненти, будучи потоковими класами, також поділяються на дві групи: ***візуальні*** (ті, що відображаються на формі як на етапі проектування, так і на етапі виконання додатка; ці компоненти досить часто називають ***елементами керування***) і ***невізуальні*** (вони відображаються на формі тільки на етапі проектування). Два різновиди візуальних компонентів (графічні та віконні) відрізняються за можливістю використання їх для керування процесом виконання додатка. ***Графічні*** (інакше ***невіконні***) ***компоненти*** служать лише для відображення інформації й не керують виконанням програми. ***Віконні компоненти*** можуть отримувати так званий фокус введення і

застосовуються для керування програмою, вимагаючи при цьому значно більше системних ресурсів, ніж графічні компоненти.

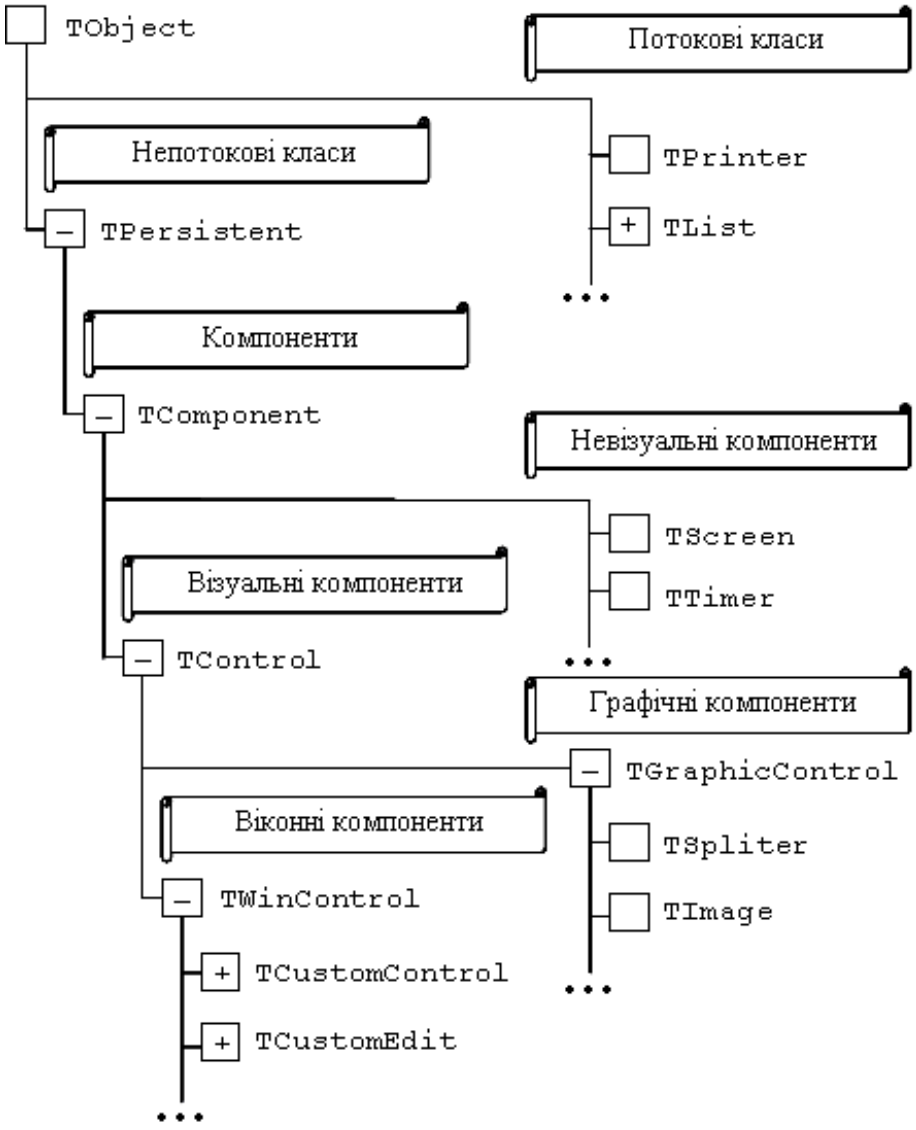


Рис. 15.1. Верхні рівні ієрархії класів VCL

Клас `TPersistent`, будучи родоначальником потокових класів, інкапсулює у собі методи, що забезпечують можливість завантаження в потік (збереження в потоці) властивостей інших об'єктів, які є його нащадками (не обов'язково прямими). Підтримка класом `TPersistent` потоків не є вбудованою – вона забезпечується іншими класами. Крім того, за допомогою його методу `Assign` можна копіювати об'єкти – спадкоємці класу `TPersistent` та їх властивості.

В описі класу `TPersistent` метод `Assign` оголошено у такий спосіб:

```
procedure Assign(Source: TPersistent); virtual;
```

Метод `Assign` при звертанні до нього вказує об'єкту на необхідність копіювання в себе вмісту об'єкта `Source`, що є параметром виклику. Якщо об'єкт `Source` порожній (має значення **`nil`**), то збуджується виняток `EConvertError`. У більшості випадків нащадки класу `TPersistent` перекривають метод `Assign`.

Метод `Assign` при його виконанні копіює об'єкт `Source`, а не встановлює посилання на нього. Цим він відрізняється від оператора присвоювання, що тільки встановлює посилання на існуючий об'єкт. Зазначимо також, що метод `Assign` може бути застосований для копіювання не лише об'єктів, але й деяких їх властивостей. У цьому випадку він поводиться так само, як і оператор присвоювання.

У прикладі 14.3 можна було скористатися (щоправда, тільки в ілюстративних цілях) методом `Assign`. Для цього в наведеному тексті процедури `TForm1.Button1Click` потрібно замінити оператор `Canvas.Brush.Bitmap:=BitMap` оператором створення об'єкта «растрове зображення» `Canvas.Brush.Bitmap:=TBitmap.Create` і вставити за ним оператор `Canvas.Brush.Bitmap.Assign(BitMap)`. У цьому випадку буде використаний метод `Assign` класу `TBitmap`, що перекриває однойменний віртуальний метод його прабатька.

Клас `TComponent` є прабатьківським класом для спеціальних типів даних Delphi, які називаються компонентами.

Компоненти мають такі особливості:

- можливість їх реєстрації в палітрі компонентів і розміщення на відповідній панелі ICP Delphi, а також можливість їх використання конструктором форми;
- здатність породжувати інші компоненти і керувати ними;
- поліпшена потоковість;
- здатність до перетворення в компоненти ActiveX.

Одним з найважливіших понять, що визначає клас `TComponent`, є поняття ***приналежності***. Справа в тому, що компоненти можуть володіти іншими компонентами, а також мати власником інший компонент, що повинен відповідати за знищення даного компонента та завантаження і збереження його **`published`**-властивостей (форма й модуль даних не можуть належати іншим компонентам). При руйнуванні якого-небудь компонента його деструктор по черзі викликає деструктори всіх компонентів, власниками яких він є, у зв'язку з чим немає потреби спеціально піклуватися про руйнування кожного з них.

З поняттям приналежності пов'язані такі властивості, які, природно, характерні для всіх компонентів:

```
property Owner: TComponent;
property ComponentCount: Integer;
property ComponentIndex: Integer;
property Components[Index: Integer]: TComponent;
```

Доступна тільки для читання властивість `Owner` указує на власника даного компонента і може вживатися у випадку, коли необхідно довідатися, який компонент володіє даним компонентом.

Інші три властивості використовуються при перегляді компонентів, що належать даному компоненту. Властивість `Components`, що доступна тільки для читання, являє собою масив, який індексується від `0` і містить список компонентів свого власника. Загальна кількість компонентів, записаних у списку компонентів `Components`, зберігається у властивості `ComponentCount`. Положення компонента в списку `Components` свого власника (відлічуваний від нуля номер) визначається значенням властивості `ComponentIndex`. Підкреслимо, що номер останнього компонента в масиві `Components` на `1` менший від значення `ComponentCount`. Для використання властивостей, специфічних для визначених компонентів, необхідно виконувати приведення типів за допомогою операції **`as`**.

У Delphi компоненти повинні мати свої імена, хоча, якщо додаток не буде маніпулювати компонентом, ім'я компонента може бути вилучене з Інспектора Об'єктів. У межах власника компонента ім'я має бути унікальним (краще, якщо воно буде унікальним у масштабах усієї програми) і бути допустимим ідентифікатором мови Delphi (див. підрозд. 3.3). Ім'я компонента визначає властивість `Name`:

```
property Name: TComponentName;
```

Тип `TComponentName` є типом **`string`**.

За умовчанням Delphi призначає компонентам послідовні імена, засновані на типі компонента (наприклад, `'Button1'`, `'Button2'` і т. д.). При цьому в ім'я компонента автоматично включається числовий суфікс. Ім'я компонента рекомендується змінювати, вживаючи префікс, що вказує на тип компонента (наприклад, `fm1` – форма, `btView1` – кнопка показу, `lbOutput1` – мітка виведення тощо).

Delphi використовує ім'я компонента при створенні заданих за умовчанням імен методів, пов'язаних з його подіями. Так, опрацьовувач події OnClick компонента Button1 отримає ім'я `Button1Click`. Якщо ж ім'я компонента за умовчанням буде змінене, відбудеться автоматична зміна імен пов'язаних з ним методів (наприклад, аналогічний опрацьовувач події OnClick для компонента `btView1` отримає ім'я `btView1Click`).

Особливе місце займає ніяк не використовувана Delphi властивість `Tag`:

```
property Tag: LongInt;
```

Ця властивість є додатковим місцем у пам'яті, що служить для зберігання цілочислових користувальницьких значень. Воно є у всіх класів компонентів, і ним програміст може розпоряджатися на власний розсуд. Застосовуючи приведення типів, у цю властивість можна записати будь-яке значення, що має розмір 4 байти (наприклад, вказівник або об'єктне посилання). У деяких наведених раніше програмах ми користувалися цією властивістю як перемикачем або індексом.

З методів класу `TComponent` зупинимося тільки на двох:

```
constructor Create(AOwner: TComponent); virtual;
```

і

```
destructor Destroy; override;
```

Конструктор `TComponent.Create` має параметр звертання, за допомогою якого створюваному компоненту передається посилання на його власника. Природно, у програмі створюють не об'єкти типу `TComponent`, а об'єкти інших типів, що є дочірніми типами щодо `TComponent`. У процесі свого виконання конструктор `Create` записує у властивість `Owner` створеного компонента посилання на його власника і вставляє посилання на новий компонент у список `Components` власника.

Наприклад, для створення в процесі виконання програми на панелі `Panel1` кнопки `Button3`, повинен бути виконаний такий оператор:

```
Button3 := TButton.Create(Panel1);
```

Звичайно, якщо потрібна гарантія того, щоб як власник створюваного компонента був об'єкт, що генерував подію, як параметр виклику слід використовувати `Self`, а не посилання на конкретний екземпляр класу:

```
Button3 := TButton.Create(Self);
```

Відзначимо також, що створений візуальний об'єкт залишиться невидимим, якщо після його створення не вказати предка нового елемента керування, визначивши значення властивості `Parent`, відсутньої в класу `TComponent` (див. п. 15.3.1).

Особливістю роботи деструктора `TComponent.Destroy` є те, що він переглядає список `Components` і, по черзі викликаючи деструктори всіх компонентів із цього списку, спочатку руйнує ці компоненти (у тому числі й створені в процесі виконання програми), а потім руйнує і сам об'єкт.

15.2. Події та їх опрацьовувачі

Компоненти в Delphi програмуються за допомогою властивостей, методів і подій. З погляду користувача компонента ***подія*** – це індикатор (ознака) виникнення якої-небудь ситуації, на яку повинен відреагувати компонент. Реакція на події реалізується так званими ***опрацьовувачами подій***, які є методами, що найчастіше належать класу тієї форми, на якій розташований компонент. Така методика передачі об'єктом частини своїх функцій іншому об'єкту називається ***делегуванням***.

Якщо виконується яка-небудь дія з компонентом (переміщення або клік мишкою, натискання клавіші й т. д.), компонент генерує подію, результатом чого може бути генерування системою інших подій. Технічно більшість подій в Delphi запускається при одержанні компонентом повідомлення від операційної системи.

У підрозд. 11.9 говорилося, що формально подія – це властивість, яка має тип «вказівник методу». Будучи оголошуваними у секції **`published`**, події відображаються в Інспекторі Об'єктів, причому всі властивості, що є вказівниками на методи, Delphi відображає на вкладці Events (Події), а всі інші властивості – на вкладці Properties (Властивості).

За згодою події залежно від змістового навантаження йменуються ідентифікаторами, які мають один з префіксів `On`, `After`, `Before` (наприклад, `OnClick` або `OnChange`).

Методи, відповідальні за виклик події компонента, називаються ***методами диспетчеризації подій***. За згодою їх іменують або ім'ям події без префікса, або після відкидання префікса додають на початок імені методу префікс `Do` (наприклад, методом диспетчеризації події `OnClick` класу `TControl` є метод `Click`, у той час як метод диспетчеризації події `OnCreate` класу `TForm` названий `DoCreate`, оскільки ім'я `Create` дане конструкторові). У деяких випадках ім'я методу диспетчеризації починають із одного з префіксів `After` або `Before`.

При створенні заданих за умовчанням імен опрацьовувачів подій для компонентів Delphi як префікс використовує ім'я компонента (див. підрозд. 15.1).

Для створення опрацьовувачів подій на етапі проектування додатка може бути застосований Object Inspector (Інспектор Об'єктів). Насамперед, необхідно перейти в ньому на вкладку Events (Події) і кліком мишкою в лівому стовпчику вибрати потрібну подію. У результаті в правому стовпчику вкладки відкриється віконце, у правій частині якого розташована кнопка  . Клік мишкою над цією кнопкою приводить до відкриття списку вибору. У цьому списку можна вибрати ім'я опрацьовувача події, що вже

визначений у проекті і може бути використаний як опрацьовувач даної події. Якщо вміст списку, що буде відкритий, не задовольняє програміста (що буває досить часто) або такий список відсутній, то можна набрати ім'я створюваного опрацьовувача події, після чого перейти у вікно коду з метою безпосереднього написання тексту опрацьовувача події. Перехід у вікно коду може бути виконаний одним з наступних способів:

- натисканням клавіші <Enter> після набору імені опрацьовувача події;
- подвійним кліком мишки над набраним ім'ям опрацьовувача події;
- подвійним кліком мишкою над пустим віконцем праворуч від імені події на вкладці Events (Події).

При цьому буде автоматично змінений код (доданий відповідний новий метод в опис форми і вставлена заготівка для реалізації цього методу). Курсор розташується у виконуваній частині вставленої заготівки методу, після чого можна перейти до безпосереднього написання програмного коду, що реалізує опрацьовувач події.

Сказане вище досить легко реалізується на практиці, що й рекомендується зробити самостійно, виконавши, наприклад, завдання 1, наведене наприкінці даного розділу.

15.3. Клас TControl: загальні властивості та методи візуальних компонентів

Клас `TControl` є одним з найбільш важливих підкласів класу `TComponent`. Він визначає загальні концепції, закладені у видимі компоненти Delphi, а саме: місце розташування та розмір елемента керування, батьківський елемент керування, на якому розміщений даний компонент, і т. д.

Компоненти класу `TControl` і його підкласів часто називають елементами керування, орієнтуючись на те, що на їхній основі реалізуються елементи керування Windows: кнопки, списки тощо.

15.3.1. Властивість Parent елемента керування

Розглянута в підрозд. 15.1 властивість `Owner` показує, який елемент *створив* даний компонент (не обов'язково візуальний) і, будучи власником компонента, відповідає за його знищення. Щоб **візуальний** компонент був промальований, необхідно вказати, який елемент керування *відповідає за виведення* створеного компонента (є його ***предком***). При цьому предком візуального компонента може бути тільки компонент класу `TWinControl`

або його дочірнього класу. Інформація про предка візуального компонента записується у властивість Parent, визначену в класі TControl у такий спосіб:

```
property Parent: TWinControl;
```

Якщо компонент розміщався на формі конструктором форми, то саме форма буде і власником, і предком цього компонента. Якщо ж при проектуванні компонент був поміщений не безпосередньо на форму, а на будь-який компонент-контейнер (наприклад, на панель), то предком створеного компонента буде контейнер, навіть якщо його власником залишиться форма (при цьому значення властивості Parent визначиться автоматично).

Якщо компонент створюється під час виконання програми, необхідно у властивість Parent занести посилання на батьківський компонент. Наприклад, для створеної в підрозд. 15.1 кнопки Button3 зразу після виклику конструктора Create треба визначити значення властивості Parent:

```
Button3.Parent := Panel1;
```

або

```
Button3.Parent := Self;
```

Якщо після створення візуального компонента не визначити значення властивості Parent, створений компонент буде залишатися невидимим.

15.3.2. Положення і розміри візуальних компонентів

Положення компонента визначається, насамперед, двома властивостями:

```
property Left: Integer;        //Положення лівої крайки
property Top: Integer;      //Положення верхньої крайки
```

Аналогічно двома властивостями характеризуються розміри компонента (у пікселях):

```
property Height: Integer;          //Висота компонента
property Width: Integer;           //Ширина компонента
```

Положення будь-якого компонента, за винятком форми, визначається в координатах так званої клієнтської області батька й задається, як і розміри, у пікселях.

Клієнтською областю компонента називається вся його внутрішня частина, тобто область, що не включає заголовок, рамку та меню.

Координати і розміри форми визначаються в межах екрана.

Значення згаданих властивостей можуть задаватися і змінюватися як на етапі конструювання форми, так і під час виконання програми, причому зміна кожної з цих властивостей зумовлює миттєве візуальне виявлення цього на формі.

Розміри клієнтської області (висоту і ширину) визначають дві властивості:

```
property ClientHeight: Integer;
property ClientWidth: Integer;
```

Наступна властивість, що є властивістю тільки для читання, містить координати кутів клієнтської області компонента усередині клієнтської області його прабатька:

```
property ClientRect: TRect;
```

У класі `TControl` визначені два методи, що дозволяють перерахувати відносні координати точки усередині клієнтської області в абсолютні екранні координати та навпаки:

```
function ClientToScreen(const Point: TPoint): TPoint;
function ScreenToClient(const Point: TPoint): TPoint;
```

Зазначимо, що точка, розташована в лівому верхньому куті екрана або клієнтської області, має координати (0, 0).

Ще однією властивістю, що керує положенням і розмірами візуальних компонентів, є властивість `Align`:

```
property Align: TAlign;
```

Тип `TAlign` визначений як перелічений тип:

```
type TAlign = (alNone, alTop, alBottom, alLeft,
               alRight, alClient, alCustom);
```

Зміст можливих значень цієї властивості такий:

- ✓ `alNone` – компонент залишається там, куди він був поміщений (значення за умовчанням);
- ✓ `alTop` (`alBottom`) – компонент притискається до верхньої (нижньої) крайки свого батька і без зміни висоти розтягується на ширину клієнтської області;
- ✓ `alLeft` (`alRight`) – компонент притискається до лівої (правої) крайки свого батька і без зміни ширини розтягується на висоту клієнтської області;
- ✓ `alClient` – компонент змінює свої розміри так, щоб зайняти всю клієнтську область батька, не заповнену іншими компонентами.

Якщо в декількох компонентів властивість `Align` установлена в стан `alClient`, то вони накладаються один на одного;

- ✓ `alCustom` – положення компонента визначається результатом виклику методів `CustomAlignInsertBefore` та `CustomAlignPosition` класу `TWinControl`.

Кілька компонентів, що мають одне й те ж саме вирівнювання, послідовно притискаються до меж батька й один до одного в порядку їх створення або зміни способу вирівнювання.

Наприклад, для створюваної раніше кнопки `Button3` можна визначити такі значення властивостей:

```
Button3.Left := 30;
Button3.Top := 40;
Button3.Height := 10;
Button3.Width := 20;
Button3.Align := alRight;
```

Після виконання всіх цих операторів кнопка завширшки 20 пікселів притиснеться до правої межі батьківського компонента і розтягнеться на всю висоту його клієнтської області. Установлені значення властивостей `Left`, `Top` і `Height` не позначаться на положенні і розмірах кнопки.

Положення компонента може бути зафіксоване відносно меж контейнера, у якому він розміщений, навіть якщо батько змінює свої розміри. Спосіб фіксування компонента визначає властивість

```
property Anchors: TAnchors;
type TAnchors = set of TAnchorKind;
type TAnchorKind = (akTop, akLeft, akRight, akBottom);
```

Можливі значення типу мають такий зміст:

- ✓ `akTop` – положення компонента фіксується відносно верхньої межі батька;
- ✓ `akLeft` – положення компонента фіксується відносно лівої межі батька;
- ✓ `akRight` – положення компонента фіксується відносно правої межі батька;
- ✓ `akBottom` – положення компонента фіксується відносно нижньої межі батька.

За умовчанням компонент фіксується відносно лівого верхнього кута його батька, тобто `Anchors = [akTop, akLeft]`. Якщо компонент буде «поставлений на якір» відносно протилежних меж батька (наприклад,

[akTop, akBottom]), то він буде змінювати свої розміри, зберігаючи сталими відстані до цих меж.

Власна властивість

```
property AutoSize: Boolean;
```

що за замовченням має значення False, дозволяє (True) або забороняє (False) візуальному компоненту автоматично змінювати свої розміри при зміні його вмісту.

Максимальні й мінімальні розміри (висота та ширина) елементів керування можуть бути обмежені за допомогою зміни властивості

```
property Constraints: TSizeConstraints;
```

Тип TSizeConstraints є класом, що має чотири властивості (MaxHeight, MaxWidth, MinHeight й MinWidth), значення яких вибираються з діапазону 0..MaxInt.

Не слід допускати конфлікт між властивістю Constraints і значенням властивості Align або Anchors, оскільки за наявності такого роду конфлікту має місце нечіткість у виконуваних діях зі зміни розмірів.

15.3.3. Активізація та зміна видимості компонентів

Можливість активізації компонента визначає властивість

```
property Enabled: Boolean;
```

Якщо ця властивість має значення False, то відповідний компонент заборонений для вибору (недоступний) і не може бути активізований. Текст на таких компонентах виводиться сірим кольором.

Будь-який видимий компонент може бути прихований або показаний установленням у стан False (сховати) або True (показати) властивості

```
property Visible: Boolean;
```

Прихований компонент не реагує на події від миші або клавіатури та не бере участь у діленні клієнтської області батька. Приховати або показати компонент можна також, скориставшись методами

```
procedure Hide;                    //Приховує компонент
procedure Show;                    //Показує компонент
```

```
//Приклад 15.1
//При кліку мишкою над кнопкою Demo вона стає недоступною,
//у два рази збільшується та переміщається з панелі
//безпосередньо на форму так, щоб її лівий верхній кут
```

```
//знаходився на 5 пікселів лівіше та нижче від лівого
//верхнього кута клієнтської області форми.
```

Скористаємося формою з прикладу 11.2, розташувавши додатково на панелі кнопку Button3. Запишемо у властивість Caption цієї кнопки напис Demo і створимо для розв'язання задачі такий опрацьовувач події OnClick:

```
procedure TForm1.Button3Click(Sender: TObject);
begin                          //Можна використати with Button3 do
  Button3.Enabled := False;          //Тепер Button3 недоступний
  Button3.Height := Button3.Height * 2;     //Змінюємо висоту й
  Button3.Width := Button3.Width * 2;          //ширину Button3
  Button3.Parent := Self;              //Змінюємо предка Button3
               //Визначаємо положення Button3 усередині батька
  Button3.Left := Button3.Parent.ClientRect.Left + 5;
  Button3.Top := Button3.Parent.ClientRect.Top + 5;
end;
```

Компонент може бути повністю закритий яким-небудь іншим компонентом, що також робить його недоступним.

Звернувшись до методу

```
procedure BringToFront;
```

можна розташувати компонент над всіма іншими компонентами.

Виклик же методу

```
procedure SendToBack;
```

робить компонент нижнім, якщо компоненти накладаються один на одного повністю або частково.

Варто враховувати, що у випадку приналежності компонента якому-небудь контейнеру при приховуванні контейнера приховуваються й усі компоненти, що містяться в ньому, навіть якщо їх властивість `Visible` має значення `True`. Тому для визначення дійсної видимості **віконного** компонента при виконанні програми перевіряють видимість усіх його предків, звертаючись до визначеної в класі `TWinControl` і доступної тільки для читання властивості

```
property Showing: Boolean;
```

Ця властивість має значення `False`, якщо або сам компонент, або хоча б один з його предків прихований.

15.3.4. Оформлення компонентів

Для настроювання користувальницького інтерфейсу компонента досить часто використовуються властивості

```
property Color: TColor;
```

і

```
property Font: TFont;
```

Загальна характеристика типів `TColor` і `TFont` була дана при розгляді графічних можливостей Delphi (див. підрозд. 14.1 і п. 14.2.3).

Властивість `Color` визначає фоновий колір, яким заливається видима частина елемента керування. Будучи **`protected`**-властивістю, властивість `Color` є невидимою нащадкам класу, оголошеним в інших модулях (див. підрозд. 11.10). У зв'язку з цим у тих дочірніх щодо `TControl` класах, у яких дана властивість повинна бути видимою, виконується переміщення її в секцію з вищим рівнем видимості (найчастіше в секцію **`published`**).

Якщо властивість `Color` не задавалася явно, то вона визначається кольором батька даного компонента. При цьому до керування кольором підключається властивість

```
property ParentColor: Boolean;
```

Якщо значення останньої властивості дорівнює `True`, то компонент завжди має колір свого батька й змінює свій колір при зміні кольору батьківського компонента. Якщо `ParentColor=False`, то зміна кольору батька не відбивається на кольорі даного компонента. Явне задання кольору компонента зумовлює ігнорування значення властивості `ParentColor`.

Шрифт, використовуваний компонентом при виведенні рядків (тип шрифту, колір символів, стиль тощо), установлюється і змінюється або за допомогою Інспектора Об'єктів, або програмно на етапі виконання додатка. Для цього в об'єкта `Font` змінюють властивості `Charset`, `Color`, `Height`, `Name`, `Pitch`, `Size`, `Style`. За допомогою зміни значення властивості `Name` можна вибрати будь-який зі шрифтів, установлених в операційній системі. При цьому треба пам'ятати про те, що на іншому комп'ютері обраний шрифт може бути відсутній, результатом чого стане вибір Windows якого-небудь придатного шрифту з можливим руйнуванням формату виведення. Упевненість у тому, що формат виведення не порушиться на іншому комп'ютері, може бути тільки у випадку застосування стандартних для операційної системи шрифтів (`Arial`, `MS Sans Serif`, `Times New Roman`, `System` і т. д.).

Для забезпечення використання візуальним компонентом шрифту, властивого його прабатькові, застосовується властивість

```
property ParentFont: Boolean;
```

З елементом керуванням зв'язується текстовий рядок, доступний через одну з таких властивостей:

```
property Caption: TCaption;
property Text: TCaption;
type TCaption = type string;
```

При цьому конкретний елемент керування використовує **тільки одну** з цих властивостей, причому за умовчанням ця властивість містить ім'я компонента. Взагалі кажучи, властивість `Caption` (Заголовок) вживається для тексту, що з'являється як назва вікна або ярлик, у той час як `Text` (Текст) призначена для тексту, що з'являється як вміст компонента.

Усі наведені вище властивості класу `TControl`, що пов'язані з оформленням компонентів, є захищеними, у зв'язку з чим вони невидимі для дочірніх класів, описаних поза модулем `Controls`, у якому визначений клас `TControl`. Дочірні класи, якщо це необхідно, підвищують рівень видимості цих властивостей. Крім того, дочірні щодо `TControl` класи розширюють перелік властивостей, пов'язаних з оформленням.

До властивостей, що деяким чином пов'язані з оформленням компонентів, можна також віднести **`published`**-властивість

```
property Hint: string;
```

і **`public`**-властивість

```
property ShowHint: Boolean;
```

Властивість `Hint` служить для визначення підказки, що з'являється в невеликому віконці поруч із курсором при розміщенні курсору миші над компонентом. Підказка буде висвітлюватись, тільки якщо у властивість `ShowHint` записане значення `True`. Для кожного конкретного компонента властивість `ShowHint` переноситься в секцію **`published`**.

Текст, що записується у властивість `Hint`, може бути розбитий на дві частини за допомогою вертикальної риски. Якщо це зроблено, то як підказка поруч із компонентом буде висвітлюватись тільки та частина тексту, що знаходиться перед вертикальною рискою. Частина тексту, розташована за вертикальною рискою, служить для виведення її в рядку панелі стану (якщо вона є, див. п. 17.1.3).

15.3.5. Зміна форми вказівника миші

За форму, якої набуває вказівник миші при його розташуванні над візуальним компонентом, відповідає властивість

```
property Cursor: TCursor;
```

Тип `TCursor` є відрізком:

```
type TCursor = -32768..32767;
```

Для стандартних вказівників миші в модулі `Controls` визначені константи, які наведені в таблиці 15.1.

Таблиця 15.1 – Стандартні вказівники миші

Константа (Значення)	Ви-гляд	Константа (Значення)	Ви-гляд	Константа (Значення)	Ви-гляд
`crDefault` (0)	Від типу	`crSizeWE` (-9)		`crSQLWait` (-17)	
`crNone` (-1)		`crUpArrow` (-10)		`crNo` (-18)	
`crArrow` (-2)		`crHourGlass` (-11)		`crAppStart` (-19)	
`crCross` (-3)		`crDrag` (-12)		`crHelp` (-20)	
`crIBeam` (-4)		`crNoDrop` (-13)		`crHandPoint` (-21)	
`crSizeNESW` (-6)		`crHSplit` (-14)		`crSize` (-22)	
`crSizeNS` (-7)		`crVSplit` (-15)		`crSizeAll` (-22)	
`crSizeNWSE` (-8)		`crMultiDrag` (-16)			

У наведеній таблиці не відображено вигляду курсору для значення `crDefault`, для якого вигляд курсору визначається типом компонента. Якщо для компонента не задати явно властивість `Cursor`, то вона за умовчанням ініціалізується значенням `crDefault`. Значення `crSize` вважається застарілим, і воно відсутнє у списку можливих значень властивості `Cursor`, відображуваних Інспектором Об'єктів.

Програміст може передбачити зміну вигляду курсору для компонента по ходу виконання програми залежно від виконання тієї або іншої умови.

Фактично властивість `Cursor` є індексом у списку курсорів, підтримуваних глобальною змінною `Screen`, що є автоматично створюваним об’єктом.

15.3.6. Події класу TControl

У класі `TControl` оголошено близько 20 подій, що є захищеними властивостями класу. Зокрема, у цьому класі визначена подія

```
property OnClick: TNotifyEvent;
```

яка має тип

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
```

Ця подія виникає при кліку кнопкою миші над компонентом, заданим як параметр `Sender`.

Для забезпечення реакції на подвійний клік мишею передбачено подію

```
property OnDblClick: TNotifyEvent;
```

Визначені також наступні події від миші:

- **property** `OnMouseDown: TMouseEvent` – відбувається при натисканні кнопки миші над компонентом;
- **property** `OnMouseMove: TMouseMoveEvent` – відбувається при переміщенні вказівника миші над компонентом;
- **property** `OnMouseUp: TMouseEvent` – відбувається, коли користувач відпускає кнопку миші, що була натиснута з вказівником миші над компонентом.

Типи наведених вище подій від миші, що є вказівниками методів, визначені в такий спосіб:

```
type
 TMouseEvent=procedure(Sender:TObject;Button:TMouseButton;
             Shift: TShiftState; X, Y: Integer) of object;
 TMouseMoveEvent = procedure(Sender: TObject;
             Shift: TShiftState; X, Y: Integer) of object;
```

Параметр `Sender` служить для ідентифікації компонента, який згенерував подію.

У `TMouseEvent` параметр `Button` визначає, яка кнопка була натиснута (ліва, права або середня). Він має перелічений тип

```
type
  TMouseButton = (mbLeft, mbRight, mbMiddle);
```

Подія від миші може виникати із супутнім натисканням однієї з керуючих клавіш Shift, Alt, Ctrl або їх комбінації. Крім того, можливе одночасне натискання лівої й правої кнопок миші. Параметр `Shift` служить для уточнення ситуації. Він має тип

```
type
  TShiftState = set of (ssShift, ssAlt, ssCtrl,
                  ssLeft, ssRight, ssMiddle, ssDouble);
```

і задається комбінацією поданих нижче значень, що мають такий зміст:

- ✓ `ssShift` – натиснута клавіша Shift;
- ➢ `ssAlt` – натиснута клавіша Alt;
- ✓ `ssCtrl` – натиснута клавіша Ctrl;
- ✓ `ssLeft` – натиснута ліва кнопка миші;
- ✓ `ssRight` – натиснута права кнопка миші;
- ➢ `ssMiddle` – натиснута середня кнопка миші;
- ✓ `ssDouble` – одночасно натиснуті ліва й права кнопки миші.

Параметри `X` і `Y` визначають координати пікселя, над яким відбулася подія (визначаються в системі координат клієнтської області батьківського компонента).

Наступні події є подіями від коліщатка миші як такого (`OnMouseWheel`), а також залежно від напрямку обертання коліщатка миші (`OnMouseWheelDown` і `OnMouseWheelUp`):

```
property OnMouseWheel: TMouseWheelEvent;
property OnMouseWheelDown: TMouseWheelUpDownEvent;
property OnMouseWheelUp: TMouseWheelUpDownEvent;
```

Інформацію про типи цих подій можна отримати, скориставшись довідковою системою Delphi.

Оскільки всі події класу `TControl` є захищеними властивостями, дочірні класи для забезпечення можливості застосування тих або інших властивостей взагалі й в Інспекторі Об'єктів зокрема перевизначають їх, переносячи використовувані властивості в **`published`**-секцію визначення класу.

15.4. Спільні властивості і методи віконних компонентів

15.4.1. Віконні компоненти та їхній дескриптор

У Windows більшість елементів користувальницького інтерфейсу відображаються вікнами, які з технічної точки зору є записами у внутрішньосистемній таблиці. Номер цього запису називається ***дескриптором вікна***.

У більшій своїй частині вікна виконують роль елементів керування, причому деякі з них є прихованими від користувача. Кожне вікно має пов'язаний з ним програмний код, що називається ***віконною функцією*** або ***віконною процедурою***, яка обробляє ті повідомлення, що надходять у це вікно. При виникненні якої-небудь події операційна система посилає повідомлення одному з вікон, що у цей момент відповідає за обробку подій даного типу. Обробка ж події здійснюється віконною функцією.

Усі віконні компоненти є екземплярами класу `TWinControl` або його нащадків.

Характерна властивість віконних компонентів – доступна тільки для читання цілочислова властивість `Handle`, що служить для зберігання дескриптора вікна. При своєму створенні віконний компонент одержує дескриптор від Windows. За допомогою дескриптора вікна можна одержати доступ до API-функцій Windows.

При виведенні зображення у віконному компоненті досить часто виникає ефект мерехтіння. Для його усунення треба записати значення `True` у його властивість

```
property DoubleBuffered: Boolean;
```

За умовчанням в неї записане значення `False`.

15.4.2. Оформлення віконних компонентів

Для деяких з компонентів при їх відображенні на формі створюється ефект об'ємності, у той час як деякі з них завжди пласкі. У той же час у багатьох віконних компонентів ефект об'ємності можна змінювати за допомогою зміни значення властивості

```
property Ctl3D: Boolean;
```

При значенні `True` цієї властивості компонент виводиться об'ємним, а при значенні `False` – пласким.

Особиста властивість

```
property ParentCtl3D: Boolean;
```

установлює, який компонент забезпечує реалізацію ефекту об'ємності – сам компонент (`False`) або його батько (`True`).

Для забезпечення тривимірного ефекту віконний компонент може окреслюватися подвійною крайкою. Зміна крайок здійснюється за допомогою властивостей `BevelEdges`, `BevelInner`, `BevelKind`, `BevelOuter`, `BevelWidth` й `BorderWidth`. Сторони компонента, у яких буде крайка, визначає властивість

```
property BevelEdges: TBevelEdges;
```

Значенням цієї властивості є множина, тип якої визначений у такий спосіб:

```
type
  TBevelEdge = (beLeft, beTop, beRight, beBottom);
  TBevelEdges = set of TBevelEdge;
```

Включення в множину значень `beLeft`, `beTop`, `beRight`, `beBottom` забезпечує відповідно промальовування лівої, верхньої, правої та нижньої крайок компонента.

Тип внутрішньої і зовнішньої крайок визначають відповідно властивості

```
property BevelInner: TBevelCut;
property BevelOuter: TBevelCut;
```

Тип `TBevelCut` є переліченим типом:

```
type
  TBevelCut = (bvNone, bvLowered, bvRaised, bvSpace);
```

Значення цього типу означають:

- ✓ `bvNone` – відсутність крайки;
- ✓ `bvLowered` – втиснена крайка;
- ✓ `bvRaised` – опукла крайка;
- ✓ `bvSpace` – невидима крайка завтовшки в 1 піксель.

Властивість

```
property BevelKind: TBevelKind;
```

служить для зміни ефекту скошування крайки.

Перелічений тип

```
type
  TBevelKind = (bkNone, bkTile, bkSoft, bkFlat);
```

визначається такими значеннями:

- ✓ `bkNone` – відсутність крайок;
- ✓ `bkTile` – є дві крайки;
- ✓ `bkSoft` – дві крайки, причому зовнішня крайка менш контрастна;
- ✓ `bkFlat` – є тільки зовнішня крайка завтовшки в 1 піксель.

Властивості

```
property BevelWidth: TBevelWidth;
type TBevelWidth = 1..MaxInt;
```

і

```
property BorderWidth: TBorderWidth;
type TBorderWidth = 0..MaxInt;
```

визначають відповідно товщину ліній крайки і відстань у пікселях між внутрішньою та зовнішньою крайками.

Усі зазначені вище властивості, що пов’язані з оформленням віконних компонентів, є захищеними.

15.4.3. Фокус введення

Серед усіх віконних компонентів, наявних на формі, у кожен конкретний момент часу тільки один може виконувати клавіатурне введення. У цьому випадку говорять, що компонент має ***фокус введення***.

При звертанні до методу

```
function Focused: Boolean; dynamic;
```

компонента, який володіє в цей момент фокусом введення, повертається значення `True`. Якщо компонент не володіє фокусом введення, метод `Focused` повертає значення `False`. Цим методом можна користуватися для перевірки приналежності фокуса введення компоненту:

```
if Edit1.Focused then оператор;
```

Передача віконному компоненту фокуса введення виконується методом

```
procedure SetFocus; virtual;
```

Для перевірки можливості передачі фокуса введення компоненту служить метод

```
function CanFocus: Boolean; dynamic;
```

Метод повертає значення `True`, якщо фокус введення може бути переданий компоненту, і значення `False` у противному випадку. Значення `False` повертається, якщо сам компонент або якийсь із його прабатьків заборонений до вибору або невидимий.

Якщо віконний компонент у конкретний момент часу не володіє фокусом введення, то фокус введення може бути переданий йому кліком мишею на цьому компоненті. Передача фокуса введення забезпечується також клавішею Tab. Порядок вибору компонентів за допомогою клавіші Tab визначається властивістю

```
property TabOrder: TTabOrder;
```

тип якої визначений як відрізок:

```
type
  TTabOrder = -1..32767;
```

Властивість `TabOrder` указує на порядковий номер компонента в списку його батька. Ці номери присвоюються в порядку створення компонентів, причому перший зі створених компонентів одержує номер `0`, і саме цьому компоненту буде належати фокус введення з появою форми на екрані.

Значення властивості TabOrder можна змінювати, змінюючи тим самим порядок обходу компонентів за допомогою клавіші Tab. Зміна номерів інших компонентів із забезпеченням унікальності номерів при цьому відбувається автоматично.

Щоб елемент міг бути обраний за допомогою клавіші Tab, необхідна установка в стан True його властивості

```
property TabStop: Boolean;
```

Для даного компонента формування списку усіх дочірніх компонентів, що можуть вибиратися за допомогою клавіші Tab, здійснюється звертанням до методу

```
procedure GetTabOrderList(List: TList);
```

```
//Приклад 15.2
//При кліку над кнопкою, що здійснює настроювання, всі
//віконні компоненти за винятком панелі здобувають
//колір clSkyBlue, а елемент, якому належить фокус
//введення, змінює накреслення символів на курсивне.
```

Скористаємося формою з прикладу 15.1, записавши у властивість Caption кнопки Button3 слово Налаштовування. Тоді задача може розв'язуватися таким опрацьовувачем події OnClick кнопки Button3:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  i: Integer;
begin
                  //Перебираємо усі компоненти, породжені формою
  for i := 0 to ComponentCount - 1 do begin
    if Components[i] is TWinControl then begin      //Віконний
                                                    //компонент?
      if not (Components[i] is TPanel) and          //Не панель?
                        //Чи може компонент мати фокус введення?
         (Components[i] as TWinControl).CanFocus then
                                                  //Змінюємо колір
         TEdit(Components[i]).Color := clSkyBlue;
                     //Якщо компоненту належить фокус введення,
      if (Components[i] as TWinControl).Focused then
                                    //змінюємо стиль його шрифту
        TEdit(Components[i]).Font.Style := [fsItalic];
      end;
  end;
                        //Перетворення до TEdit виконувалося для
end;       //гарантування доступу до властивостей Color і Font
```

У даному випадку ілюструється можливість використання методів CanFocus і Focused. Оскільки елементи властивості Components є

екземплярами класу TComponent, у якого відсутні методи CanFocus і Focused, звичайний виклик цих методів (наприклад, Components[i].CanFocus) неможливий, навіть якщо як компонент, на який посилається елемент Components[i], виступає візуальний компонент. У зв'язку із цим при звертанні до методів CanFocus і Focused здійснюється перетворення типу компонента Components[i] до типу TWinControl за допомогою операції **as**. Аналогічно елементи масиву в загальному випадку не мають властивості Color і Font. Тому перед зміною цих властивостей тип компонента перетвориться до типу TEdit, оскільки в класу TEdit зазначені властивості є опублікованими (замість класу TEdit при перетворенні типу можна було застосувати будь-який клас, у якого доступні властивості Color і Font). Наведений приклад ілюструє також можливість використання властивостей Components і ComponentCount при перегляді наявних на формі компонентів.

Зазначимо, що в даному прикладі для забезпечення доступу до властивостей Color і Font віконних компонентів можна було б спробувати виконати приведення типу візуального компонента не до типу TEdit, а до типу TControl, що має зазначені властивості. Однак оскільки код підпрограми TForm1.Button3Click розташовується не в модулі Controls, що містить опис класу TControl, а в іншому модулі, то властивості Color і Font в цьому випадку виявляться недоступними, і буде діагностуватися помилка з видачею діагностичного повідомлення про відсутність оголошення ідентифікатора.

Іншим методом доступу до захищених елементів предка є так званий *хакерський* підхід, про який згадувалося наприкінці підрозд. 11.10. У цьому випадку в модулі, у якому потрібно звернутися до захищених елементів предка, описується клас, породжений класом, що містить ці елементи.

Наприклад, в інтерфейсній частині модуля призначеного для розв'язання прикладу 15.2, можна описати такий тип:

```
type
  THacker = class(TControl);
```

Оскільки визначений подібним чином допоміжний клас буде знаходитися в тому ж самому модулі, що і програмний код, пов'язаний з доступом до захищених елементів, конфліктна ситуація буде розв'язана.

Таким чином, у наведеному вище тексті замість операторів

```
TEdit(Components[i]).Color := clSkyBlue;
```

і

```
TEdit(Components[i]).Font.Style := [fsItalic];
```

можна записати відповідно оператори

```
THacker(Components[i]).Color := clSkyBlue;
```

і

```
THacker(Components[i]).Font.Style := [fsItalic];
```

за умови, що в інтерфейсній частині модуля є наведений вище опис класу `THacker`.

15.4.4. Події класу TWinControl

Усього в класі визначено вісім властивостей-подій. Розглянемо тільки ті з них, які пов'язані з фокусом введення та клавіатурою:

- **property** `OnEnter: TNotifyEvent` – виникає, коли віконний компонент отримує фокус введення;
- **property** `OnExit: TNotifyEvent` – виникає, коли віконний компонент втрачає фокус введення;
- **property** `OnKeyDown: TKeyEvent` – виникає при натисканні будь-якої клавіші;
- **property** `OnKeyPress: TKeyPressEvent` – виникає при натисканні алфавітно-цифрової клавіші;
- **property** `OnKeyUp: TKeyEvent` – виникає при відпусканні клавіші.

Типи `TKeyEvent` й `TKeyPressEvent` визначені так:

```
type
  TKeyEvent = procedure(Sender: TObject; var Key: Word;
                        Shift: TShiftState) of object;
  TKeyPressEvent = procedure(Sender: TObject;
                             var Key: Char) of object;
```

Параметр `Key` у `TKeyEvent` містить віртуальний код клавіші, а в `TKeyPressEvent` – ASCII-код. У `TKeyEvent` параметр `Shift` уточнює, чи натискалася одночасно зі звичайною клавішею одна або кілька керуючих клавіш <Shift>, <Ctrl>, <Alt> (див. п. 15.3.6). Зчитавши значення параметрів `Key` і `Shift` усередині відповідного опрацьовувача події, можна визначити клавішу й організувати реакцію на її натискання або відпускання.

Отримати інформацію про віртуальні коди клавіш у довідковій системі Delphi не можна через її відсутність, але це можна зробити, звернувшись до файлу Source/Rtl/Win/Windows.pas, у якому визначені відповідні константи (див. таблицю 15.2).

Таблиця 15.2 – Віртуальні коди клавіш

Код (десятк.)	Код (шістн.)	Константа	Клавіша
8	$8	VK_BACK	BackSpace
9	$9	VK_TAB	Tab
12	$C	VK_CLEAR	[5]
13	$D	VK_RETURN	Enter
16	$10	VK_SHIFT	Shift
17	$11	VK_CONTROL	Ctrl
18	$12	VK_MENU	Alt
19	$13	VK_PAUSE	Pause
20	$14	VK_CAPITAL	Caps Lock
27	$1B	VK_ESCAPE	Esc
32	$20	VK_SPACE	Пробіл
33	$21	VK_PRIOR	Page Up
34	$22	VK_NEXT	Page Down
35	$23	VK_END	End
36	$24	VK_HOME	Home
37	$25	VK_LEFT	←
38	$26	VK_UP	↑
39	$27	VK_RIGHT	↑
40	$28	VK_DOWN	↓
45	$2D	VK_INSERT	Insert
46	$2E	VK_DELETE	Delete
48 … 57	$30 … $39		0 … 9
65 … 90	$41 … $5A		A … Z
91	$5B	VK_LWIN	Windows (ліва)
92	$5C	VK_RWIN	Windows (права)
93	$5D	VK_APPS	Контекстне меню
Клавіші додаткової цифрової клавіатури			
96 … 105	$60 … $69	VK_NUMPAD0 … VK_NUMPAD9	[0] … [9]
106	$6A	VK_MULTIPLY	[*]
107	$6B	VK_ADD	[+]
109	$6D	VK_SUBTRACT	[-]
110	$6E	VK_DECIMAL	[Del]
111	$6F	VK_DIVIDE	[/]
Клавіші верхньої частини клавіатури			
112 … 123	$70 … $7B	VK_F1 … VK_F12	F1 … F12
124 … 135	$7C … $80	VK_F13 … VK_F24	F13 … F24
144	$90	VK_NUMLOCK	Num Lock

Продовження табл. 15.2

Код (десятк.)	Код (шістн.)	Константа	Клавіша
145	$91	VK_SCROLL	Scroll Lock
Клавіші з символами-роздільниками			
186	$BA		:
187	$BB		+
188	$BC		<
189	$BD		-
190	$BE		>
191	$BF		?
192	$C0		~
219	$DB		[
220	$DC		\
221	$DD		]

Слід зазначити, що не для всіх віртуальних кодів клавіш у файлі WINDOWS.PAS визначені іменовані константи. Крім того, у тексті цього файлу рекомендується самостійно визначати, якщо це необхідно, константи VK_0 … VK_9 і VK_A … VK_Z з встановленням їх відповідності клавішам <0> … <9> і <A> … <Z> (природно, можна користуватися числовими значеннями кодів).

Зазначимо також, що ліві й праві клавіші <Shift>, <Ctrl> і <Alt>, взагалі кажучи, мають різні коди, причому у файлі WINDOWS.PAS для них визначені відповідні константи:

```
VK_LSHIFT = 160;
VK_RSHIFT = 161;
VK_LCONTROL = 162;
VK_RCONTROL = 163;
VK_LMENU = 164;
VK_RMENU = 165.
```

При цьому оброблювачі подій OnKeyDown і OnKeyUp не розрізнюють місцезнаходження клавіш <Shift>, <Ctrl> і <Alt>. Для цього потрібно звертатись до API-функцій Windows. Можна також використовувати оброблювач події OnMessage системної змінної Application, аналізуючи його параметр Msg.lParam.

Клавіші <Enter> на основній і додатковій частинах клавіатури також можна розрізнити за допомогою зазначених вище засобів.

Клавіші, які не передають ASCII-символи, не збуджують подію OnKeyPress, що дозволяє забезпечити відключення реакції на натискання таких клавіш.

У багатьох випадках потрібно забезпечити реакцію на натискання клавіш до того, як на цю подію зреагує компонент, що володіє фокусом введення. Для того щоб форма отримала подію до надходження її в елемент із фокусом введення, слід помістити значення `True` у її властивість

```
property KeyPreview: Boolean;
```

За умовчвнням ця властивість має значення `False`.

Як приклад, що ілюструє реакцію на подію OnKeyPress, можна розглянути приклад 18.3.

15.5. Невізуальні та графічні компоненти

Невізуальні компоненти відображаються тільки на етапі проектування програми, хоча деякі з них можуть бути видимі й на етапі її виконання. Загалом кажучи, невізуальними компонентами називаються всі класи, що походять від `TComponent`, за винятком `TControl`.

Підкреслимо, що описані в модулі `Dialogs` стандартні діалогові вікна, незважаючи на те, що вони є дочірніми класами класу `TCommonDialog`, який є прямим нащадком `TComponent`, не тільки видимі на етапі проектування програми, але при звертанні до них стають видимими й під час виконання додатка. Мало того, клас `TCommonDialog`, а отже, і його нащадки мають загальнодоступну властивість `Handle` для одержання дескриптора вікна, а також **`published`**-властивість `Ctl3D`, що керує ефектом об'ємності при виведенні компонента.

Базовим класом для графічних елементів керування є клас `TGraphicControl`. Графічні компоненти не мають потреби в одержанні фокуса введення, а отже, не вимагають ідентифікатора вікна, у зв'язку із чим клас `TGraphicControl` і його нащадки утворюють окрему гілку в ієрархії класів Delphi, паралельну гілці, що виходить від класу `TWinControl`.

Будучи спадкоємцями класу `TControl`, об'єкти дочірніх щодо `TGraphicControl` класів можуть відповідати на події від миші, для чого в цих класів підвищується рівень видимості відповідних властивостей-подій.

Клас `TGraphicControl` не має власного зображення, але в ньому визначені захищений віртуальний метод

```
procedure Paint;
```

і захищена властивість

```
property Canvas : TCanvas;
```

що визначає полотно для малювання.

Ці елементи після їх перевизначення в спадкоємців з підвищенням рівня видимості дозволяють екземплярам дочірніх класів виводити своє зображення.

Запитання для контролю і самоконтролю

1. Що в Delphi розуміють під терміном «компонент»?
2. У чому відмінність візуальних і невізуальних компонентів?
3. Чим відрізняються графічні компоненти від віконних?
4. Що таке події і які елементи програми здійснюють реакцію на них?
5. Як створити опрацьовувач події на етапі проектування програми?
6. Яке призначення класу `TControl`?
7. Яка функція властивості `Parent`?
8. У чому відмінність властивостей `Owner` і `Parent`?
9. Які властивості відповідають за положення та розміри компонентів?
10. Які властивості та методи відповідають за активацію й зміну видимості компонентів?
11. Чи може видимий компонент бути недоступним?
12. Чи може бути доступним невидимий компонент?
13. За допомогою яких засобів можна змінювати оформлення компонентів?
14. Як здійснюється зміна форми вказівника миші?
15. Охарактеризуйте основні події класу `TControl`.
16. Як забезпечується реакція на події від миші?
17. Що таке дескриптор вікна?
18. Які додаткові властивості відповідають за оформлення віконних компонентів?
19. Що таке фокус введення?
20. За допомогою яких засобів виконується керування фокусом введення?
21. Наведіть основні події класу `TWinControl`.
22. Які події пов’язані з реакцією на натискання/відпускання клавіш?
23. Що можна сказати про події `OnKeyPress` та `OnKeyUp`, що генеруються при натисканні лівих та правих керуючих клавіш <Shift>, <Ctrl>, <Alt>?
24. Як забезпечити реакцію форми на натискання клавіші до відповідної реакції компонента, що володіє фокусом введення?
25. Дайте коротку характеристику невізуальних і графічних компонентів.

Завдання для практичного відпрацьовування матеріалу

1. Створити форму і написати для неї опрацьовувач події OnCanResize (За зміною розміру). При зміні розмірів форми в її рядку заголовка повинні відображатися висота і ширина форми. Примітка: в опрацьовувача події OnCanResize його параметри `NewWidth` і `NewHeight`, що мають тип `Integer`, забезпечують повернення нових розмірів компонента (відповідно висоти та ширини).
2. Забезпечити, щоб при створенні форми в її центрі автоматично створювалася та висвітлювалася кнопка, яка при кліку мишею над нею повинна займати випадкові координати в клієнтській області форми, не виходячи за її межі.
3. Реалізувати «гумову лінію». Кліком лівою кнопкою миші над канвою форми вибрати початкову точку ламаної. Далі при натиснутій лівій кнопці миші переміщати вказівник миші, слідом за яким повинна тягтися пряма, що виходить з початкової точки та змінює своє положення залежно від позиціювання вказівника. При відпусканні лівої кнопки миші точка останнього положення курсору миші фіксується, і подальше малювання ламаної має відбуватися щодо неї. Вихід із процесу малювання організувати за кліком правою кнопкою миші.
4. Виконати попереднє завдання, забезпечуючи малювання за допомогою клавіатури. Вивести точку на канву і переміщати її за допомогою клавіш зі стрілками до натискання клавіші <Enter>. З зафіксованої точки промалювати пряму за допомогою клавіш зі стрілками. Пряма фіксується за натисканням клавіші <Enter>, після чого остання промальована точка стає початковою для продовження процесу малювання. За натисканням клавіші <Esc> виконувати перехід до вибору нової початкової точки.

16. ФОРМА ТА КОНТЕЙНЕРИ ЗАГАЛЬНОГО ПРИЗНАЧЕННЯ

16.1. Форма – компонент TForm

Форма (клас `TForm`) є тим компонентом, без якого не обходиться жодна програма, написана у середовищі Delphi. Це віконний компонент, що визначений у модулі `Forms` і спадкує властивості та методи `TWinControl` за ланцюжком `TWinControl` – `TScrollingWinControl` – `TCustomForm` – `TForm`. Компонент `TForm` не міститься на жодній з вкладок палітри компонентів і на етапі конструювання створюється одним з таких способів:

- автоматично при створенні нового додатка за командою File ► New ► Application (Файл ► Створити ► Додаток) або при кліку мишкою над піктографічною кнопкою  , розташованою на панелі Стандартна, з наступним вибором команди Application (Додаток) на вкладці New (Створити) у діалоговому вікні, що при цьому відкриється;
- за командою File ► New ► Form (Файл ► Створити ► Форма) або при кліку мишкою над піктографічною кнопкою  , розташованою на панелі Стандартна, з наступним вибором команди Form (Форма) на вкладці New (Створити) у діалоговому вікні, що при цьому відкриється, або при кліку мишкою над піктографічною кнопкою  (Нова форма) на панелі Вид.

На етапі конструювання можливе також створення спеціалізованих форм, у тому числі діалогових вікон. Для цього треба або виконати команду File ► New ► Other (Файл ► Створити ► Ще), або клацнути мишкою над кнопкою  на панелі Стандартна, після чого у вікні, яке при цьому відкриється, перейти на одну із вкладок Forms (Форми) чи Dialogs (Діалоги) і вибрати потрібну форму. Зазначимо, що на вкладці Dialogs (Діалоги) є

команда Dialog Wizard (Майстер Діалогу), яка дозволяє викликати проектувальник форм, за допомогою якого можна спроектувати одну зі стандартних діалогових форм.

Якщо додаток містить кілька форм, то одна з них є головною. ***Головною формою*** називається форма, що з'являється на екрані в момент запуску програми. Найчастіше головною формою є форма, що створюється при проектуванні додатка.

Якщо в додатку є кілька форм, то будь-яку з них можна оголосити головною. Для цього необхідно виконати команду Project► Options (Проект► Опції), після чого у вікні, що при цьому відкриється, перейти на вкладку Forms (Форми) і, розкривши на ній список Main form (Головна форма), вибрати форму, яка повинна бути головною.

Форма має досить багато властивостей, методів і подій, що забезпечують її функціонування.

Серед властивостей форми зазначимо, насамперед, властивість

```
property FormStyle: TformStyle;
```

що має тип `TFormStyle`, визначений у такий спосіб:

```
type
  TFormStyle = (fsNormal, fsMDIChild,
                fsMDIForm, fsStayOnTop);
```

Зміст значень типу `TFormStyle` такий:

- ✓ `fsNormal` – звичайна форма, у тому числі головна;
- ✓ `fsMDIChild` – дочірнє MDI-вікно (MDI – Multi Document Interface);
- ✓ `fsMDIForm` – головне MDI-вікно (рамкове вікно), усередині якого можуть з'являтися дочірні MDI-вікна;
- ✓ `fsStayOnTop` – вікно-«поплавець», що повинне завжди розташовуватися над іншими вікнами.

Головне MDI-вікно може мати своє головне меню, у той час як його дочірні вікна фактично свого власного головного меню не мають. У той же час на дочірнє MDI-вікно можна помістити головне меню, яке, однак, у момент активізації цього вікна перевизначає головне меню рамкового вікна-власника. Дочірні елементи не можуть виходити за межі свого власника. Такі вікна зараз застосовуються досить рідко. Для створення багатовіконних додатків сьогодні найчастіше використовується стиль SDI (SDI – Single Document Interface), що базується на задаванні як стиль вікна значення `fsNormal`, яке, по-перше, не накладає обмеження на положення й розміри допоміжних форм, використовуваних як вікна, і, по-друге, дозволяє створювати форми (вікна) зі своїм головним меню.

Вікна, що реалізуються за допомогою форм, можуть бути модальними або немодальними.

Модальне вікно характеризується тим, що з появою його на екрані воно повністю бере на себе керуючі функції з неможливістю виконання оператора, розташованого за оператором, який вивів це вікно на екран. Цей оператор виконується тільки після закриття вікна.

Немодальне вікно після своєї появи на екрані працює разом з іншими вікнами, не замикаючи на себе процес керування програмою. Після виведення немодального вікна відразу ж виконується оператор, що розташовується слідом за оператором, який забезпечив виведення вікна на екран.

Модальність вікна визначається методом виведення його на екран. Для виведення немодального та модального вікна застосовують відповідно такі методи класу `TForm`:

```
procedure Show;
function ShowModal: Integer; virtual;
```

Ці методи успадковані відповідно від класів `TControl` і `TCustomForm`.

Метод `ShowModal`, будучи функцією, повертає у вигляді цілого числа результат діалогу, що організується у вікні, причому він не завершує свою роботу, поки вікно не буде закрите. Щоб закрити модальну форму (модальне вікно), необхідно записати відмінне від нуля значення в її властивість

```
property ModalResult: TModalResult;
```

Тип цієї властивості визначений у такий спосіб:

```
type TModalResult = Low(Integer)..High(Integer);
```

Якщо модальне вікно містить кнопки з відмінним від нуля значенням властивості `ModalResult`, то при кліку мишкою над однією з цих кнопок вікно автоматично закривається, а функція `ShowModal` повертає значення, що дорівнює значенню властивості `ModalResult` кнопки, над якою був зроблений клік мишкою.

Звичайно при аналізі значення властивості `TModalResult` модального вікна використовують визначені у модулі `Controls` константи, що вказують, за якою з кнопок здійснено вихід:

- ✓ `mrNone` – значення за умовчанням (що дорівнює `0`), яке позначає те, що не натискалася жодна кнопка;
- ✓ `mrOk { = 1 }` – вихід за кнопкою OK;
- ✓ `mrCancel { = 2 }` – вихід за кнопкою CANCEL;
- ✓ `mrAbort { = 3 }` – вихід за кнопкою ABORT;

- ✓ `mrRetry { = 4 }` – вихід за кнопкою RETRY;
- ✓ `mrIgnore { = 5 }` – вихід за кнопкою IGNORE;
- ✓ `mrYes { = 6 }` – вихід за кнопкою YES;
- ✓ `mrNo { = 7 }` – вихід за кнопкою NO;
- ✓ `mrAll { = mrNo + 1}` – вихід за кнопкою ALL;
- ✓ `mrNoToAll { = mrAll + 1 }` – вихід за кнопкою NO TO ALL;
- ✓ `mrYesToAll { = mrNoToAll + 1 }` – вихід за кнопкою YES TO AL.

Крім двох згаданих вище властивостей, вікна мають такі властивості:

- ➢ **property** `Active: Boolean` – вказує, належить (`True`) або не належить (`False`) вікну фокус введення (властивість тільки для читання);
- ➢ **property** `ActiveControl: TWinControl` – визначає приналежний формі дочірній елемент керування, що володіє фокусом введення;
- ➢ **property** `AlphaBlend: Boolean` – визначає, чи є форма напівпрозорою (`True`) чи ні (`False`), впливаючи тільки при використанні Windows 2000 і вище;
- ➢ **property** `AlphaBlendValue: Byte` – встановлює ступінь напівпрозорості, впливаючи тільки при `AlphaBlend = True` (значення `0` – повністю прозора форма; значення `255` – непрозора форма);
- ➢ **property** `AutoSize: Boolean` – визначає, чи буде форма встановлювати свої розміри автоматично залежно від свого вмісту (`True` – буде, `False` – ні);
- ➢ **property** `BorderIcons: TBorderIcons` – визначає, які кнопки містяться в заголовку вікна (див. нижче);
- ➢ **property** `BorderStyle: TFormBorderStyle` – встановлює вигляд і поведінку рамки вікна (див. нижче);
- ➢ **property** `Canvas: TCanvas` – канва для малювання;
- ➢ **property** `ClientHeight: Integer` – висота клієнтської області вікна;
- ➢ **property** `ClientRect: TRect` – прямокутник клієнтської області (тільки для читання);
- ➢ **property** `ClientWidth: Integer` – ширина клієнтської області вікна;
- ➢ **property** `HelpFile:` **`string`** – ім'я індивідуального файлу допомоги, що використовує форма (якщо ім'я файлу не зазначено, використовується файл допомоги додатка);

- **property** Icon: TIcon – визначає зображення, що з'являється при мінімізації форми, а для головної форми одночасно і значок програми (якщо значення властивості не визначене, вживається значок операційної системи за умовчанням);
- **property** KeyPreview: Boolean – при значенні True форма отримує події від клавіатури до її активного елемента керування, а при значенні False (за умовчанням) події від клавіатури отримує тільки активний елемент керування форми;
- **property** Menu: TMainMenu – визначає головне меню форми;
- **property** Position: TPosition – визначає розмір і положення форми (див. нижче);
- **property** TransparentColor: Boolean – при значенні True указує, що колір, зумовлений значенням властивості TransparentColorValue, є прозорим (впливає тільки при використанні Windows 2000 і вище);
- **property** TransparentColorValue: TColor – визначає колір прозорості форми при TransparentColor = True й ігнорується при TransparentColor = False;
- **property** WindowState: TWindowState – визначає стан вікна в момент появи його на екрані (див. нижче).

Тип TBorderIcons властивості BorderIcons є множиною:

```
type
  BorderIcon = (biSystemMenu, biMinimize,
                biMaximize, biHelp);
  TBorderIcons = set of TBorderIcon;
```

За умовчанням вікно має маленький значок, підключений до системного меню, а також кнопки згортання, розгортання та закриття. Значення типу BorderIcon визначають, які значки можуть бути присутні у смузі заголовка:

- ✓ biSystemMenu – є кнопка виклику системного меню;
- ✓ biMinimize – є кнопка мінімізації (згортання) вікна;
- ✓ biMaximize – є кнопка максимізації вікна;
- ✓ biHelp – є кнопка виклику довідкової служби.

Змінюючи значення властивості BorderIcons, можна змінювати вигляд смуги заголовка вікна. За умовчанням властивість BorderIcon має значення [biSystemMenu,biMinimize,biMaximize]. При видаленні системного меню зникають усі значки смуги заголовка вікна. Кнопки мінімізації і максимізації можна видалити тільки одночасно. Повністю видалити тільки кнопку мінімізації або кнопку максимізації не можна – при

видаленні однієї з цих кнопок вона стає затіненою та втрачає активність. Кнопка виклику довідкової служби з'являється в смузі заголовка вікна тільки при видаленні обох кнопок мінімізації та максимізації або при установці властивості BorderStyle значення bsDialog (у цьому випадку згадані кнопки відсутні).

Тип TFormBorderStyle властивості BorderStyle є переліченим типом:

```
type
  TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
            bsDialog, bsToolWindow, bsSizeToolWin);
```

з такими значеннями:

- ✓ bsNone – вікно не має рамки та заголовка і не може переміщатися, а також змінювати свої розміри (цей стиль використовують тільки у виняткових випадках й усередині інших форм);
- ✓ bsSingle – вікно, що не змінює свої розміри і має рамку завтовшки 1 піксель;
- ✓ bsSizeable – звичайна рамка (значення за умовчанням);
- ✓ bsDialog – рамка діалогового вікна (не змінює розміри);
- ✓ bsToolWindow – подібно до bsSingle, але з меншим за висотою заголовком;
- ✓ bsSizeToolWin – подібно до bsSizeable, але з меншим по висоті заголовком.

Положення та розміри вікна при його виведенні на екран визначаються властивістю Position, що має тип

```
type
  TPosition = (poDesigned, poDefault,
               poDefaultPosOnly, poDefaultSizeOnly,
               poScreenCenter, poDesktopCenter,
               poMainFormCenter, poOwnerFormCenter);
```

Зміст можливих значень цього типу такий:

- ✓ poDesigned – розміри та положення ті ж самі, що й на етапі конструювання форми;
- ✓ poDefault – положення та розміри визначаються операційною системою;
- ✓ poDefaultPosOnly – розміри, як на етапі конструювання, а положення визначає операційна система;
- ✓ poDefaultSizeOnly – положення, як на етапі конструювання, а розміри визначає операційна система;

- ✓ `poScreenCenter` – форма з розмірами, установленими на етапі конструювання, виводиться в центрі екрана;
- ✓ `poDesktopCenter` – подібно до `poScreenCenter`, але без урахування можливості багатомоніторних додатків;
- ✓ `poMainFormCenter` – розміри форми ті ж самі, що й на етапі конструювання, але вона поміщається в центр головної форми (для головної форми є еквівалентним `poScreenCenter`);
- ✓ `poOwnerFormCenter` – розміри форми ті ж самі, що й при її створенні, але вона поміщається в центр форми, яка визначена властивістю `Owner` (якщо властивість `Owner` не визначена, є аналогом `poMainFormCenter`).

Значення типу

```
type
  TWindowState = (wsNormal, wsMinimized, wsMaximized);
```

використовуються для зазначення виду форми в момент появи її на екрані:

- ✓ `wsNormal` – нормальний стан;
- ✓ `wsMinimized` – форма, що згорнута до кнопки;
- ✓ `wsMaximized` – форма, розгорнута до максимального розміру.

В окрему групу можна виділити **`public`**-властивості, які мають відношення до рамкових MDI-вікон, тобто до вікон зі значенням властивості `FormStyle`, що дорівнює `fsMDIForm`:

- ➢ **`property`** `ActiveMDIChild: TForm` – визначає дочірнє MDI-вікно, яке володіє фокусом введення (якщо форма не є рамковим вікном, то `ActiveMDIChild` містить порожній вказівник **`nil`**);
- ➢ **`property`** `MDIChildCount: Integer` – визначає кількість дочірніх MDI-вікон, пов'язаних з формою (є значимою тільки для рамкової форми);
- ➢ **`property`** `MDIChildren[I: Integer]: TForm` – забезпечує доступ до дочірнього MDI-вікна з номером `I`;
- ➢ **`property`** `TileMode: TTileMode` – визначає, як розташовуються дочірні форми при виклику методу `Tile` (горизонтально або вертикально).

Можливими значеннями переліченого типу

```
type
  TTileMode = (tbHorizontal, tbVertical);
```

є такі константи:

- ✓ `tbHorizontal` – вікна розташовуються уздовж висоти батьківського вікна (горизонтальні вікна);

- ✓ `tbVertical` – вікна розташовуються уздовж ширини батьківського вікна (вертикальні вікна).

Зазначимо, що перелік властивостей форми не вичерпується тими, що були розглянуті вище.

Закриття форми здійснюється методом

```
procedure Close;
```

успадкованим класом `TForm` від класу `TCustomForm`. При цьому якщо закривається головна форма, виконання додатка припиняється.

```
//Приклад 16.1
//Створити проект, що містить три SDI-форми. Головна форма
//повинна забезпечувати видимість двох форм, що займають ліву
//та праву половини екрана і містять багаторядкові редактори
//у припущенні, що перший з них служить для набору тексту
//російською мовою, а другий – для перекладу цього тексту на
//українську мову (або навпаки). Передбачити кнопки для
//забезпечення збереження набраних текстів, а також захист від
//закриття вікон без збереження тексту.
```

Насамперед, створимо головну форму невеликого розміру й розташуємо на ній дві кнопки – звичайну кнопку та кнопку з зображенням. Змінимо деякі властивості цих компонентів:

- Форма:
  Caption — Головна форма
  Name — FormMain
  Position — poScreenCenter
- Кнопка Button1:
  Caption — Робота
- Кнопка BitBtn1:
  Kind — bkClose

Кліком мишкою над піктографічною кнопкою  на панелі Вид або яким-небудь іншим методом створимо форму для роботи з російським текстом, у нижній частині якої помістимо панель висотою приблизно в 40 пікселів із двома кнопками Button1 і Button2. Крім того, розташуємо на формі багаторядкове редаговане текстове поле Memo1.

Змінимо властивості компонентів:

- Форма:
  Caption — Російський текст
  Name — FormRus
- Панель:
  Align — alBottom

BevelOuter — bvNone
Caption — очистити

- Кнопка Button1:
  Caption — Зберегти
  Name — btRusSave1
- Кнопка Button2:
  Caption — Закрити
  Name — btRusClose2
- Багаторядкове поле:
  Align — alClient
  Lines — очистити
  Name — MemoRus1
  ScrollBars — ssBoth

Створимо також форму для роботи з українським текстом, аналогічну описаній вище формі для роботи з російським текстом. Виконаємо ті ж самі зміни властивостей її компонентів за винятком таких:

- Форма:
  Caption — Український текст
  Name — FormUkr
- Багаторядкове поле:
  Name — MemoUkr1
- Кнопка Button1:
  Name — btUkrSave1
- Кнопка Button2:
  Name — btUkrClose2

Збережемо тексти модулів для «російського» й «українського» вікон відповідно у файлах UnitRus.pas й UnitUkr.pas, а текст модуля, що відповідає головній формі, – у файлі UnitMain.pas. При цьому всі необхідні зміни в тексті проекту, що зумовлені перейменуванням модулів, ICP Delphi виконає самостійно.

Врахуємо також наступне. Розроблювальний додаток повинен забезпечувати роботу з текстовим редактором, а отже, необхідно подбати про те, щоб при випадковому закритті редакторського вікна не було втрачено незбережені дані. Тому для кожного з редакторських вікон створимо вікно, що буде відкриватися в модальному режимі при спробі закриття редакторського вікна без збереження змін у його вмісті.

Отже, створимо нову форму невеликого розміру, розділивши її за висотою двома панелями, і розташуємо на нижній панелі три кнопки

Button1, Button2 і Button3, які будуть служити відповідно для подачі команд про необхідність збереження, про вихід без збереження та про відмову від закриття вікна.

Змінимо властивості компонентів:

- Форма:
  Caption — очистити
  BorderStyle — bsDialog
  Name — FormMesRus
  Position — poScreenCenter
- Нижня панель:
  Align — alBottom
  BevelOuter — bvNone
  Caption — очистити
  Name — pnButtons2
- Верхня панель:
  Align — alClient
  BevelOuter — bvNone
  Caption — Текст у російському вікні змінився! Зберегти зміни?
  Name — pnMes1
- Кнопка Button1:
  Caption — Так
  Name — Yes
- Кнопка Button2:
  Caption — Hi
  Name — No
- Кнопка Button3:
  Caption — Скасування
  Name — Cancel

Створимо також ще одне вікно, що відрізняється від описаного вище вікна тільки таким:

- Форма:
  Name — FormMesUkr
- Верхня панель:
  Caption — Текст в українському вікні змінився! Зберегти зміни?

Збережемо відповідні модулі у файлах MesRus.pas та MesUkr.pas.

Створимо також опрацьовувачі події OnClick для кнопок, розташованих на описаних вище формах.

Для форми FormMain:

```
procedure TFormMain.Button1Click(Sender: TObject);
begin
                                        //Приховуємо головну форму
  Hide;
                                       //Приховуємо кнопку Button1
  Button1.Enabled := False;
                                    //Показуємо "російське" вікно
  FormRus.Show;
                      //Нижче використаний об'єкт Screen – екран
          //Screen.WorkAreaWidth – ширина вільної частини екрану
  FormRus.Width := Screen.WorkAreaWidth div 2;
  FormRus.Align := alLeft;              //Ліва половина екрана
                                  //Показуємо "українське" вікно
  FormUkr.Show;
  FormUkr.Width := Screen.WorkAreaWidth div 2;
  FormUkr.Align := alRight;            //Права половина екрана
end;
```

Для форми FormRus:

```
procedure TFormRus.btRusSave1Click(Sender: TObject);
begin
                      //Оскільки йде збереження, то на майбутнє
                             //вважаємо, що вміст не змінювався
   MemoRus1.Modified := False;
              //Тут потрібно зберегти вміст Memo-вікна
              //Це можна зробити, наприклад, так:
   MemoRus1.Lines.SaveToFile('Rus.txt');
end;

procedure TFormRus.btRusClose2Click(Sender: TObject);
begin
  if MemoRus1.Modified then            //Якщо текст змінювався
    case FormMesRus.ShowModal of  //Виведення модального вікна
      mrYes: btRusSave1Click(Self);       //У модальному вікні
            //натискалася кнопка YES – збереження за допомогою
             //виклику опрацьовувача події OnClick кнопки Save
      mrCancel: Exit;          //У модальному вікні натискалась
             //кнопка CANCEL – повернення в редакторське вікно
    end;
  with FormMain do begin
    Tag := Tag + 1;                   //Тут Tag форми FormMain
    if Tag = 2
      then Show;                     //Тут Show форми FormMain
  end;
  Close;                          //Закриття "російського" вікна
end;
```

Для форми FormUkr:

```
procedure TFormUkr.btUkrSave1Click(Sender: TObject);
begin
//Оскільки йде збереження, то вважаємо, що вміст не змінювався
  MemoUkr1.Modified := False;
    //Тут потрібно зберегти вміст Memo-вікна:
  MemoUkr1.Lines.SaveToFile('Ukr.txt');
end;

procedure TFormUkr.btUkrClose2Click(Sender: TObject);
begin
  if MemoUkr1.Modified then            //Якщо текст змінювався
    case FormMesUkr.ShowModal of  //Виведення модального вікна
      mrYes: btUkrSave1Click(Self);       //У модальному вікні
            //натискалася кнопка YES - збереження за допомогою
             //виклику опрацьовувача події OnClick кнопки Save
      mrCancel: Exit;          //У модальному вікні натискалась
             //кнопка CANCEL - повернення в редакторське вікно
    end;
  with FormMain do begin
    Tag := Tag + 1;                   //Тут Tag форми FormMain
    if Tag = 2 then Show;            //Тут Show форми FormMain
  end;
  Close;                        //Закриття "українського" вікна
end;
```

Для форми FormMesRus:

```
procedure TFormMesRus.YesClick(Sender: TObject);
begin
  ModalResult := mrYes;                //Вихід за кнопкою YES
end;

procedure TFormMesRus.NoClick(Sender: TObject);
begin
  ModalResult := mrNo;                 //Вихід за кнопкою NO
end;

procedure TFormMesRus.CancelClick(Sender: TObject);
begin
  ModalResult := mrCancel;           //Вихід за кнопкою CANCEL
end;
```

Для форми FormMesUkr тексти опрацьовувачів події OnClick кнопок відрізняються від наведених вище текстів тільки тим, що вони стосуються не класу `TFormMesRus`, а класу `TFormMesUkr`.

Підкреслимо, що в текстах модулів `UnitMain`, `UnitRus` і `UnitUkr` виконується звертання до ресурсів, оголошених в інших модулях, у зв'язку з чим у модулі `UnitMain` необхідно здійснити підключення модулів

UnitRus і UnitUkr, у модулі UnitRus – модулів UnitMain і MesRus, а в модулі UnitUkr – модулів UnitMain і MesUkr. Якщо запустити на компіляцію проект, то будуть видаватися повідомлення про непідключені модулі, що ввійшли в проект, із пропозицією про автоматичне їх підключення. При позитивній відповіді відсутні посилання на модулі будуть включені у відповідні **implementation**-секції. Природно, підключення модулів можна виконати і вручну.

Розглянемо стисло особливості функціонування додатка.

При запуску програми на виконання виводиться головна форма, клік мишкою над кнопкою Робота якої приводить до виконання методу TFormMain.Button1Click, що забезпечує виведення «українського» та «російського» редакторських вікон у немодальному режимі. При цьому «російське» вікно притискається до лівої межі екрана, розтягуючись на всю його висоту, а «українське» – до правої межі.

При роботі з «російським» редакторським вікном можна виконувати набір тексту в його Memo-полі, а за кліком мишкою над кнопкою Зберегти – зберігати його (безпосереднє збереження в опрацьовувачі події OnClick кнопки btRusSave1 у пропонованому тексті не реалізовувалося). Особливістю компонента Memo є те, що в нього є властивість Modified типу Boolean, яка автоматично набуває значення True, якщо текст усередині цього компонента змінювався. Тому при збереженні тексту в цю властивість заноситься значення False, яке вказує на те, що збереження тексту виконувалося.

При закритті «російського» вікна кнопкою Закрити за допомогою аналізу значення властивості MemoRus1.Modified спочатку перевіряється збереження тексту. Якщо збереження не проводилося, то в модальному режимі виводиться вікно FormMesRus, причому метод ShowModal повертає результат діалогу (яка з кнопок у цьому вікні натискалася). Якщо натискалася кнопка Так (mrYes), то відбувається збереження тексту за допомогою звертання до методу TFormRus.btRusSave1Click. Якщо результат діалогу говорить про те, що натискалася кнопка Скасування (mrCancel), то функціонування редакторського вікна продовжується. При натисканні ж у модальному вікні кнопки Ні (mrNo) редакторське вікно закривається без збереження.

«Українське» редакторське вікно функціонує аналогічно російському.

Зазначимо, що властивість Tag головної форми FormMain використовується для підрахунку кількості закритих редакторських вікон. Головна форма виводиться тільки після того, як будуть закриті обидва редакторських вікна. Опрацьовувачі події OnClick кнопок модальних вікон

FormMesRus і FormMesUkr забезпечують запис у властивість ModalResult відповідної кнопки необхідного значення (mrYes, mrNo або mrCancel). Воно автоматично записується у властивість ModalResult відповідного модального вікна і передається в метод TFormRus.btRusClose2Click при роботі з «російським» редакторським вікном (або в метод TFormUkr.btUkrClose2Click при роботі з «українським» вікном) як результат роботи методу ShowModal для застосування переданого значення як перемикача в операторі **case**.

Розглянемо призначення деяких методів, успадкованих класом TForm від класу TCustomForm:

- **function** CloseQuery: Boolean – перевіряє можливість закриття вікна, повертаючи значення True, якщо вікно може бути закрите, і значення False при неможливості закриття вікна;
- **procedure** FocusControl(Control: TWinControl) – передає фокус введення дочірньому елементу Control;
- **function** GetFormImage: TBitmap – повертає поточне зображення вікна;
- **procedure** Hide – приховує форму, надаючи властивості Visible значення False;
- **procedure** Print – друкує вікно на принтері, використовуючи метод GetFormImage;
- **procedure** Release – знищує форму і звільняє пов'язану з нею пам'ять (знищення не проводиться до закінчення обробки всіх подій як форми, так і всіх її дочірніх елементів);
- **procedure** SendCancelMode(Sender: TControl) – відновлює початковий стан форми, звільняючи мишу, припиняючи скролінг і закриваючи меню;
- **procedure** SetFocus – передає формі фокус введення за умови, що форма є видимою та активною;
- **function** SetFocusedControl(Control:TWinControl): Boolean – передає фокус введення елементу Control форми, повертаючи значення False, якщо цей елемент уже мав фокус введення, і значення True у противному випадку (значення True повертається навіть якщо елемент керування не може отримати фокус введення, наприклад, будучи невидимим).

Безпосередньо в класі TForm визначено такі методи (здійсненні тільки для рамкових вікон, тобто при значенні властивості FormStyle форми, що дорівнює fsMDIForm):

- **procedure** Cascade – розташовує дочірні MDI-вікна каскадом;

- **procedure** Next – активізує наступне дочірнє MDI-вікно форми;
- **procedure** Previous – активізує попереднє дочірнє MDI-вікно форми;
- **procedure** Tile – розташовує дочірні MDI-вікна плиткою (мозаїкою), установлюючи їм однакові розміри.

Зазначимо, що фактичний список методів класу TForm значно більший.

Форма може обробляти такі події, що виникають:

- **property** OnActivate: TNotifyEvent – при одержанні формою фокуса введення (при активації форми);
- **property** OnCanResize: TCanResizeEvent – при зміні розмірів вікна;
- **property** OnClick: TNotifyEvent – при кліку лівою кнопкою мишки над формою;
- **property** OnCreate: TNotifyEvent – при створенні форми;
- **property** OnDblClick: TNotifyEvent – при подвійному кліку лівою кнопкою мишки над формою;
- **property** OnDeactivate: TNotifyEvent – при втраті формою фокуса введення (при деактивізації форми);
- **property** OnDestroy: TNotifyEvent – при руйнуванні форми;
- **property** OnHide: TNotifyEvent – при прихованні форми (установці її властивості Visible у стан False);
- **property** OnPaint: TNotifyEvent – при перемальовуванні форми;
- **property** OnResize: TNotifyEvent – відразу ж за зміною розмірів форми;
- **property** OnShow: TNotifyEvent – при показі форми (установці її властивості Visible у стан True).

Тип TCanResizeEvent події OnCanResize визначений так:

```
type
  TCanResizeEvent = procedure(Sender: TObject;
        var NewWidth, NewHeight: Integer;
        var Resize: Boolean) of object;
```

де Sender – об’єкт, що змінив розмір; NewWidth та NewHeight – нові розміри відповідно по вертикалі та горизонталі; Resize – дозвіл (True) або заборона (False) на зміну розмірів.

При створенні форми, якщо її властивість `Visible` має значення `True`, наступні події відбуваються в такому порядку: OnCreate → OnShow → OnActivate → OnPaint.

Усі перелічені властивості успадковані формою від її батьків, причому вище наведено далеко не всі властивості класу `TForm`.

16.2. Фрейм – компонент TFrame

Фрейм (або ***рама***, ***кадр***) є контейнером, що відрізняється від форми тим, що його можна поміщати в інші компоненти, у тому числі й на форму. Піктограма фрейму знаходиться на вкладці Standard палітри компонентів, а сам компонент визначений у модулі `Forms`, розташовуючись у такому ланцюжку ієрархії віконних компонентів Delphi: `TWinControl – TScrollingWinControl – TCustomFrame – TFrame`.

За допомогою фреймів досить часто створюють ***шаблони*** – набори компонентів, пристосованих для конкретних потреб.

Фрейм не може бути відразу вставлений у проект. Якщо шаблон зберігається на диску, то його можна підключити до проекту, виконавши команду Project ► Add To Project (Проект ► Додати До Проекту), якій відповідає кнопка  на панелі Standard (Стандартна) ICP Delphi або натискання клавіші Shift+F11. При виконанні цієї команди відкривається діалогове вікно, у якому треба вибрати модуль, який відповідає шаблону, що підключається.

Якщо потрібний шаблон відсутній, то його створюють за допомогою команди File ► New ► Frame (Файл ► Створити ► Фрейм). При виконанні цієї команди на екран завантажується вікно фрейму, за виглядом ідентичне вікну форми (але без координатних точок), яке можна наповнювати компонентами аналогічно тому, як це робиться при проектуванні форми.

Після створення шаблона він може бути доданий до проекту за описаною вище методикою (однак попередньо необхідно зберегти спроектований шаблон).

Якщо фрейм підключений до проекту, він може розміщатися на формі або усередині інших фреймів, причому кількість рівнів вкладеності фреймів один в одного не обумовлена. Для цього достатньо клацнути мишкою над піктографічною кнопкою  на вкладці Standard палітри компонентів, у результаті чого відкриється діалогове вікно зі списком підключених до проекту фреймів, у якому за допомогою мишки слід вибрати потрібний фрейм.

Якщо до проекту не підключений жоден фрейм, то спроба помістити на форму компонент-фрейм спричиняє видачу повідомлення про відсутність фреймів у проекті.

Створений шаблон може бути зареєстрований у палітрі компонентів. Для цього після збереження фрейму потрібно клацнути правою кнопкою миші над вікном фрейму й вибрати в локальному меню пункт Add To Palette (Додати в Палітру), у результаті чого відкриється діалогове вікно для реєстрації шаблона в палітрі компонентів (рис. 16.1).

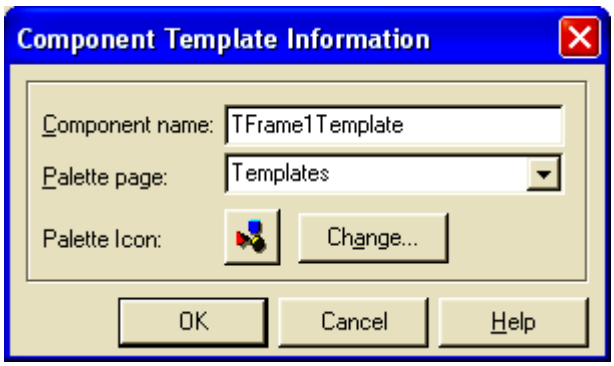


Рис. 16.1. Вікно реєстрації фрейму в палітрі компонентів

За умовчанням для шаблонів передбачається вкладка Templates, що у початковому переліку сторінок палітри компонентів відсутня. Замість імені Templates можна вказати будь-яке ім'я сторінки, у тому числі вибрати існуючу сторінку палітри компонентів. Якщо фрейм реєструється на сторінці, відсутній у палітрі компонентів у момент реєстрації, то створюється нова вкладка, на якій розміщається піктографічна кнопка із зображенням піктограми, що відображалася у вікні реєстрації (піктограму можна змінити). Запропоноване у вікні реєстрації ім'я класу треба змінювати відповідно до призначення шаблона, оскільки це ім'я з'являється на ярличку-підказці при розміщенні курсору миші над піктографічною кнопкою шаблона в палітрі компонентів.

Для видалення шаблона необхідно в палітрі компонентів перейти на відповідну сторінку палітри компонентів і, клацнувши над нею правою кнопкою миші, викликати контекстне меню, з наступним вибором опції Properties (Властивості). У вікні Palette Properties (Властивості Палітри), що після цього відкриється, виконується вибір шаблона і його видалення. Для цих же цілей можна скористатися командою головного меню Tools ► Environment Options (Інструменти ► Властивості Оточення) з переходом у відкритому в результаті цього вікні на вкладку Palette (Палітра), де вже можна вибрати сторінку палітри та потрібний компонент.

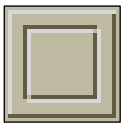
Наголосимо, що зареєстрований у палітрі компонентів шаблон може бути розміщений на формі тільки за умови, що до проекту підключений

модуль із відповідним фреймом. Якщо модуль не підключений, то при спробі розміщення на формі компонента-шаблона зі сторінки палітри компонентів на екрані виводиться вікно з повідомленням про необхідність підключення модуля та пропозицією про автоматичне його додавання.

Як шаблон може бути зареєстрована будь-яка група компонентів, розміщених на формі. Для цього достатньо виділити групу компонентів, що цікавить проектувальника додатка, з наступною реєстрацією шаблона за допомогою команди Component ► Create Component Template (Компоненти ► Створити Шаблон). Виділення проводиться кліками лівою кнопкою мишки над компонентами при натиснутій клавіші Shift. При реєстрації відкривається вікно, аналогічне вікну, зображеному на рис. 16.1. Дії з реєстрації збігаються з тими діями, що виконуються реєстрації шаблонів, які базуються на фреймах.

Властивості, методи та події компонента `TFrame` успадковуються від його прабатьків.

16.3. Панель – компонент TPanel



Компонент `TPanel` визначений у модулі `ExtCtrls` і міститься на нижньому рівні такого ланцюжка віконних компонентів: `TWinControl – TCustomControl – TCustomPanel – TPanel`.

Піктографічна кнопка компонента `TPanel` розміщається на вкладці Standard палітри компонентів.

Панель може бути вставлена в будь-яке місце форми, фрейму або іншого контейнера. Звичайно компонент `TPanel` поміщають таким чином, щоб його дочірні елементи розташовувалися уздовж однієї зі сторін того контейнера, що містить цей компонент. Так, у більшості розглянутих раніше прикладів панель розміщалася в нижній частині форми, а в прикладах 12.1, 12.2 та 14.3–14.5 розташування панелі вибиралося таким чином, щоб її дочірні елементи розташовувалися уздовж правої границі форми.

Панель може бути обведена рамкою завтовшки 1 піксель. Наявність рамки визначає властивість

```
property BorderStyle: TBorderStyle;
```

Тип `TBorderStyle` визначений у такий спосіб:

```
type
  TBorderStyle = bsNone..bsSingle;
```

де значення `bsNone` вказує на відсутність рамки, а значення `bsSingle` – на наявність рамки завтовшки 1 піксель (див. також підрозд. 16.1).

Інші властивості, а також методи та події (принаймні, найчастіше використовувані) описані раніше при розгляді класів `TWinControl` і `TControl` (див. підрозд. 15.3 і 15.4).

Іноді властивість `Caption` компонента `TPanel` застосовують для виведення невеликих повідомлень, але це недоцільно, оскільки для виведення текстів існують спеціальні компоненти.

16.4. Панель групування – компонент TGroupBox

Компонент `TGroupBox`, піктографічна кнопка якого розміщається на вкладці Standard палітри компонентів, визначений у модулі `StdCtrls`, перебуваючи на нижньому рівні ланцюжка віконних компонентів `TWinControl – TCustomControl – TCustomGroupBox – TGroupBox`.

За призначенням цей компонент аналогічний компоненту `TPanel`, але, на відміну від останнього, у верхній частині рамки (у її розриві) містить текст, що задається значенням властивості `Caption` і служить заголовком для групи компонентів.

Рамка в цього компонента, на відміну від `TPanel`, знищена бути не може.

16.5. Панель зі скролінгом – компонент TScrollBox

Компонент `TScrollBox` визначений у модулі `Forms` і розташований на нижньому рівні ланцюжка віконних компонентів `TWinControl – TScrollingWinControl – TScrollBox`. Його піктограма розміщається на вкладці Additional палітри компонентів.

Компонент `TScrollBox` аналогічний компоненту `TPanel`, але при цьому передбачено можливість вбудовування в нього смуг скролінга, що дозволяє розміщати в ньому інші компоненти таким чином, що вони, навіть виходячи за його межі, залишаються доступними за рахунок можливості прокручування. Даний компонент успадковує властивості, методи й події своїх прабатьків. Його специфічна властивість `BorderStyle` збігається з аналогічною властивістю компонента `TPanel`. Від свого прабатька `TScrollingWinControl` цей компонент успадковує властивість

```
property AutoScroll: Boolean;
```

яке при значенні `True` забезпечує автоматичну вставку однієї або двох смуг скролінга у випадку, коли якийсь з дочірніх виходить за його межі.

При `AutoScroll = False` смуги скролінга не вставляються, а частини дочірніх компонентів, що виходять за розміри даного контейнера, відтинаються.

Характеристиками горизонтальної й вертикальної смуг скролінга керують відповідно властивості `HorzScrollBar` та `VertScrollBar`.

16.6. Панель з однобічним прокручуванням – компонент TPageScroller

Компонент `TPageScroller` є прямим спадкоємцем `TWinControl`, будучи визначеним у модулі `ComCtrls`. У нього можна помістити віконний компонент, причому ним зазвичай є довгий вузький горизонтальний або вертикальний компонент.

Піктографічна кнопка компонента `TPageScroller` розташовується на вкладці Win32 палітри компонентів.

Якщо дочірній компонент не вміщується в розміри компонента `TPageScroller`, то по боках даного контейнера з'являються стрілки прокручування, що забезпечують переміщення всередині нього дочірнього компонента вліво – вправо (–) або вгору – вниз (–), залежно від того, як налаштований контейнер. Відмінність даного компонента від компонента `TScrollBox` полягає в тому, що, по-перше, у нього замість смуг скролінга є стрілки прокручування, і, по-друге, прокручування може виконуватися тільки або по горизонталі, або по вертикалі.

Властивості даного компонента здебільшого визначені в класі `TWinControl`, спадкоємцем якого він є. Крім того, у нього є такі специфічні властивості:

- **property** `AutoScroll: Boolean` – дозволяє (`True`) або забороняє (`False`) автоматичне прокручування при розміщенні вказівника миші над кнопкою прокручування (при `AutoScroll = True` прокручування здійснюється автоматично без кліків мишею над кнопкою, а при `AutoScroll = False` – тільки по кліках мишею);
- **property** `ButtonSize: Integer` – ширина (при горизонтальній орієнтації) або висота (при вертикальній орієнтації) кнопок прокручування;
- **property** `Margin: Integer` – відстань у пікселях від меж компонента до відповідних сторін дочірнього вікна;
- **property** `Orientation: TPageScrollerOrientation` – орієнтація компонента;

- **property** Position: Integer – поточне положення (зсув у пікселях) дочірнього (того, що прокручується) вікна щодо меж компонента.

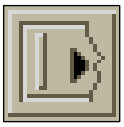
Тип TPageScrollerOrientation, що визначає орієнтацію даного компонента, є переліченим типом

```
type
  TpageScrollerOrientation = (soHorizontal,
                              soVertical);
```

з такими можливими значеннями:

- ✓ soHorizontal – компонент орієнтований по горизонталі;
- ✓ soVertical – компонент орієнтований по вертикалі.

16.7. Панель зі смугами – компонент TCoolBar



Компонент TCoolBar є віконним компонентом – спадкоємцем другого рівня компонента TWinControl (TWinControl – TToolWindow – TCoolBar). Він визначений у модулі ComCtrls, а його піктографічна кнопка розміщена на вкладці Win32 палітри компонентів. Компонент призначений для створення налаштовуваних панелей, що мають смуги, які виглядають і діють як розсувні штори. Біля лівої межі смуг (при їх горизонтальному розташуванні) звичайно вставляється вертикальна вішка, за допомогою якої при її захопленні мишкою розкриваються або закриваються смуги (штори). Клік мишкою над цією вішкою забезпечує автоматичне виконання згаданих дій.

Смуги (штори) у панель додаються кліком над кнопкою  поруч із ім'ям її властивості

```
property Bands: TCoolBands;
```

в Інспекторі Об'єктів, у результаті чого відкривається Редактор Смуг (Bands Editor), кліками над кнопкою  якого здійснюється створення нових смуг (див. рис. 16.2). Виклик Редактора Смуг може бути виконаний також кліком мишею над зображенням компонента на формі (подвійний клік лівою кнопкою або клік правою кнопкою з вибором команди Bands Editor у контекстному меню, що буде відкрите). У Редакторі Смуг є також кнопка  , що служить для видалення виділеної смуги, а також кнопки зі стрілками для організації переходу від однієї смуги до іншої.

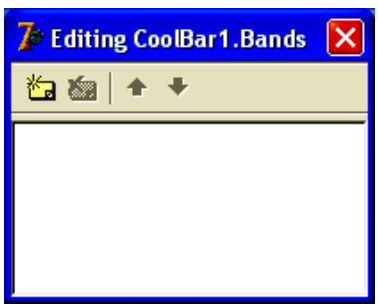


Рис. 16.2 – Вікно редактора смуг

Властивість Bands – це список (масив) смуг, кожна з яких має тип TCoolBand, що є класом. Доступ до пов'язаного зі смугою елементу керування забезпечується за допомогою її властивості

```
property Control: TWinControl;
```

Програмно доступ до окремих смуг списку Bands може бути здійснений за її індексом за допомогою властивості

```
property Items[Index: Integer]: TCoolBand;
```

Наприклад, якщо є об'єкт CoolBar1 типу TCoolBar, а також кнопка Button3, то може бути виконаний такий оператор:

```
Coolbar1.Bands.Items[2].Control := Button3;
```

Цей оператор помістить кнопку Button3 на смугу з номером 2 (за умови, що смуга з таким номером існує).

Будучи контейнером, смуга може містити тільки один віконний компонент. Це обмеження можна обійти: оскільки компонентом, розміщуваним на смузі, може бути інший контейнер, відкривається можливість розміщення на ній декількох компонентів.

Крім властивості Control, кожна смуга має такі властивості:

- **property** Bitmap: TBitmap – визначає зображення, що циклічно повторюється по всій смузі;
- **property** BorderStyle: TBorderStyle – є аналогічною однойменній властивості панелі (див. підрозд. 16.3);
- **property** Break: Boolean – при значенні True смуга розташовується в тому ж рядку, що й попередня смуга, а при значенні False (значення за умовчанням) – у новому рядку;
- **property** FixedBackground: Boolean – забороняє (True) або дозволяє (False) повторення на поверхні смуги зображення, визначеного властивістю Bitmap;
- **property** FixedSize: Boolean – забороняє (True) або дозволяє (False – за умовчанням) зміну розмірів смуги;
- **property** Height: Integer – зчитує висоту смуги (тільки для читання);
- **property** HorizontalOnly: Boolean – при значенні True смуга не відображається, якщо властивість Vertical панелі (див. нижче) встановлює вертикальне виведення смуг;
- **property** ImageIndex: TImageIndex – визначає індекс зображення, що виводиться на смузі й міститься у властивості Images панелі;

- **property** ParentBitmap: Boolean – при значенні True, якщо для смуги не визначена властивість Bitmap, у ній виводиться зображення, зумовлене властивістю Bitmap компонента TCoolBar;
- **property** MinHeight: Integer – мінімальна висота смуги при зміні її розмірів;
- **property** MinWidth: Integer – мінімальна ширина смуги при зміні її розмірів;
- **property** Text: **string** – містить текст, що виводиться поруч із зображенням компонента, розміщеного на смузі;
- **property** Width: Integer – ширина смуги.

Компонент TCoolBar, крім згадуваної вище властивості Bands, має такі властивості:

- **property** Align: TAlign – визначає вирівнювання смуги (див. п. 15.3.2);
- **property** AutoSize: Boolean – при значенні True висота компонента автоматично узгоджується з висотою смуг;
- **property** Bitmap: TBitmap – визначає зображення, використовуване всіма смугами, у яких властивість ParentBitmap має значення True;
- **property** FixedOrder: Boolean – при значенні True забороняє переміщення смуг;
- **property** Images: TCustomImageList – містить список зображень, зв'язаних з кожною зі смуг;
- **property** ShowText: Boolean – при значенні True на етапі виконання на смузі відображається текст, записаний у її властивість Text;
- **property** Vertical: Boolean – при значенні True смуги розташовуються по вертикалі уздовж лівої межі компонента.

Для компонента TCoolBar визначена єдина специфічна подія OnChange (інші події успадковуються від його прабатьків):

```
property OnChange: TNotifyEvent;
```

Ця подія виникає при зміні властивостей Break, Index або Width кожної зі смуг.

```
//Приклад 16.2
//Організувати виведення bmp-файлів, що зберігаються в поточ-
//ній папці. Імітувати показ зображення, закритого шторою. При
//кожному відкритті штори здійснювати зміну зображення
//з виведенням повідомлення при завершенні списку bmp-файлів.
```

Створимо форму, помістивши на неї компонент CoolBar та звичайну панель і розмістивши на останній компонент Image. Описаним вище способом додамо в панель CoolBar дві смуги. Виберемо в Дереві Об'єктів першу смугу й клацнемо мишкою в Інспекторі Об'єктів праворуч від її властивості Control, з вибором компонента Panel1 за кліком над кнопкою  праворуч від віконця, що відкрилося. На завершення проектування захопимо мишкою другу смугу й перетягнемо її в рядок панелі CoolBar, у якому розташовувалася перша смуга, і підтягнемо другу смугу максимально можливо вліво.

У секції **implementation** модуля помістимо опис

```
var
  FRec: TSearchRec;
```

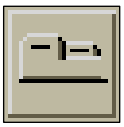
Для розв'язання задачі можна скористатися такими опрацьовувачем події OnCreate форми та опрацьовувачем події OnChange панелі зі смугами:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
                                          //Усуваємо мерехтіння
  Form1.Panel1.DoubleBuffered := True;
                        //Установлюємо розміри панелі зі смугами
  CoolBar1.Align := alClient;
  Coolbar1.Align := alCustom;
  CoolBar1.Height := CoolBar1.Height;
  CoolBar1.Width := CoolBar1.ClientHeight;
                             //Змінюємо розміри звичайної панелі
  Form1.Panel1.Width := CoolBar1.ClientHeight;
  Form1.Panel1.Height := CoolBar1.ClientWidth;
                                          //та компонента Image1
  Image1.Width := Panel1.ClientWidth;
  Image1.Height := ClientHeight;
end;

procedure TForm1.CoolBar1Change(Sender: TObject);
var
  Code: Integer;
begin
  if Tag = 0 then Code := FindFirst('*.bmp', faAnyFile, FRec)
  else Code := FindNext(FRec);
  if Code = 0 then Image1.Picture.LoadFromFile(FRec.Name)
  else begin
    Image1.Picture := nil;           //Більше зображення немає
    Panel1.Caption := 'Більше немає bmp-файлів';//Повідомлення
    if Tag = 1 then FindClose(FRec);
  end;
  Tag := 1;
end;
```

У процедурі виконується пошук BMP-файлів у поточній папці, причому якщо властивість `Tag` форми дорівнює `0` (значення за умовчанням), то шукається перший такий файл за допомогою функції `FindFirst`, а далі пошук проводиться за допомогою функції `FindNext`. Перед виходом із цієї процедури у властивість `Tag` записується значення `1` для забезпечення виклику `FindNext`.

Якщо `FindFirst` або `FindNext` не знаходить BMP-файл, то в змінну `Code` записується значення, відмінне від нуля, після чого очищається властивість `Picture` компонента `Image1` і підготовляється повідомлення про відсутність шуканих файлів.

16.8. Набір закладок – компонент TTabControl



Компонент `TTabControl`, піктографічна кнопка якого розташовується на вкладці Win32 палітри компонентів, є спадкоємцем другого рівня компонента `TWinControl` (`TWinControl – TCustomTabControl – TTabControl`), що визначений у модулі `ComCtrls`.

Від класу `TCustomTabControl` даний компонент успадковує такі специфічні властивості, переносячи їх у секцію **`published`** свого визначення:

- **`property`** `HotTrack: Boolean` – при значенні `True` назва закладки автоматично виділяється кольором, якщо над нею розміщається курсор миші, а при кліку мишею над назвою закладки вона обводиться рамкою;
- **`property`** `MultiLine: Boolean` – якщо закладки не вміщаються в межах компонента, то при значенні `True` вони розташовуються в кілька рядів, а при значенні `False` закладки розміщаються в один ряд, а в зону закладок вставляються кнопки прокручування;
- **`property`** `RaggedRight: Boolean` – при значенні `False` ряди заголовків розтягуються на всю ширину компонента, а при значенні `True` – ні;
- **`property`** `ScrollOpposite: Boolean` – при значенні `True` дозволяється автоматичне перекидання неактивних рядів заголовків на протилежну сторону компонента, а при значенні `False` перекидання заборонене;
- **`property`** `Style: TTabStyle` – стиль закладок (див. нижче);

- **property** TabHeight: SmallInt – висота закладки в пікселях (при TabHeight = 0 висота закладки визначається автоматично за висотою шрифту);
- **property** TabIndex: Integer – номер активної закладки з відліком від 0 (якщо закладки відсутні, то TabIndex = -1);
- **property** TabPosition: TTabPosition – положення закладок щодо робочої області (див. нижче);
- **property** Tabs: TStrings – написи на закладках (кількість написів визначає кількість закладок);
- **property** TabWidth: SmallInt – ширина закладок у пікселях (при TabWidth = 0 ширина кожної закладки визначається автоматично залежно від напису на ній).

Стиль закладок установлює тип

```
type
  TTabStyle = (tsTabs, tsButtons, tsFlatButtons);
```

Цей тип визначають такі константи

- ✓ tsTabs – стандартні закладки;
- ✓ tsButtons – закладки у вигляді кнопок;
- ✓ tsFlatButtons – закладки у вигляді плоских кнопок.

Значення типу

```
type
  TTabPosition = (tpTop, tpBottom, tpLeft, tpRight);
```

визначають, що закладки розташовані над (tpTop) чи під (tpBottom) робочою областю компонента або ліворуч (tpLeft) чи праворуч (tpRight) від неї.

Крім згаданих властивостей, від класу TCustomTabControl успадковуються дві властивості тільки для читання:

- **property** Canvas: TCanvas – визначає канву для малювання;
- **property** DisplayRect: TRect – прямокутник клієнтської області.

Характерними для даного компонента є події

```
property OnChange: TNotifyEvent;
```

та

```
property OnChanging: TTabChangingEvent;
```

Перша з них виникає при виборі нової закладки, а друга – після вибору.

Тип `TTabChangingEvent` події OnChanging визначений так:

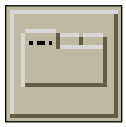
```
type
  TTabChangingEvent = procedure (Sender: TObject;
                var AllowChange: Boolean) of object;
```

Присвоївши усередині опрацьовувача події OnChanging параметру `AllowChange` значення `False`, можна заборонити вибір закладки.

На етапі проектування для створення закладок у компоненті TabControl потрібно в Інспекторі Об'єктів вибрати властивість Tabs і клацнути мишкою в другому стовпчику над кнопкою  праворуч від імені властивості, у відповідь на що відкриється Редактор Списку Рядків (String List Editor), у якому необхідно набрати імена закладок. Після закриття цього редактора за допомогою його кнопки OK закладки будуть створені, причому їх кількість буде дорівнює кількості набраних рядків.

Особливістю компонента є те, що він являє собою одну сторінку з багатьма закладками, звертаючись до яких можна виконувати різні дії над компонентами, розміщеними в цьому контейнері.

16.9. Набір сторінок із закладками – компонент TPageControl



Компонент `TPageControl`, як і `TTabControl`, є спадкоємцем `TCustomTabControl` і визначений у модулі `ComCtrls`. Його піктографічна кнопка розміщена на вкладці Win32 палітри компонентів.

На відміну від `TTabControl`, цей компонент складається з декількох сторінок, кожна з яких має свою закладку і є окремим контейнером – компонентом `TTabSheet` (набір закладок).

Набір закладок `TTabSheet` можна використовувати і самостійно поза компонентом `TTabControl`. У цьому випадку він може бути розміщений на формі як на етапі проектування вибором його піктограми  , розташованої на вкладці Win 3.1 палітри компонентів, так і при програмній модифікації форми під час виконання програми.

Клас `TPageControl` має ті ж самі згадані вище **published**-властивості, що й клас `TTabControl` (за винятком властивості `Tabs`, область видимості якої до **published** не підвищується).

Специфічною **published**-властивістю цього класу є властивість

```
property ActivePage: TTabSheet;
```

яка містить посилання на активну сторінку даного компонента. Зміна цієї властивості під час виконання програми розташовує відповідну сторінку поверх інших. Клік мишкою над закладкою сторінки забезпечує автоматичну зміну властивості ActivePage.

Кількість і список наявних сторінок установлюють дві специфічні властивості тільки для читання:

```
property PageCount: Integer;
property Pages[Index: Integer]: TTabSheet;
```

Остання властивість дозволяє звернутися до кожної зі сторінок за її номером (відлік від 0) і зробити її активною.

Наприклад, якщо k – ціла змінна, а PageControl1 – компонент типу TPageControl, то можна записати такий оператор:

```
with  PageControl1 do begin
  if k < PageCount then ActivePage := Pages[k];
end;
```

```
//Приклад 16.3
//Створити записну книжку зі сторінками, назви яких
//визначаються літерами латинського алфавіту. Передбачити
//завантаження вмісту сторінок з файлів та їхнє збереження.
//Імена файлів A.txt, B.txt, ...,Z.txt.
```

Створимо форму, у нижній частині якої розмістимо панель висотою приблизно у 40 пікселів з очищеною властивістю Caption. На форму помістимо дві кнопки Button1 і Button2, у властивості Caption яких запишемо відповідно тексти Завантажити і Вийти.

У секції **interface** підключимо модуль ComCtrls, а в секцію **implementation** модуля форми внесемо опис

```
var
  PageControl1: TPageControl;
```

Створимо також опрацьовувачі події OnClick для компонентів Button1 і Button2:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  C: Char;       //Літера для назви сторінки та імені її файлу
  Page: TTabSheet;
  Memo: TMemo;
begin  //Створюємо записник
  PageControl1 := TPageControl.Create(Self);
  PageControl1.Parent := Self;                 //Батько – форма
  PageControl1.Align := alClient;  //На всю клієнтську область
  for C:='A' to 'Z' do begin //Цикл за створюваними сторінками
    Page := TTabSheet.Create(Self);       //Створюємо сторінку
```

```
    with Page do begin                  //Для створеної сторінки
      PageControl := PageControl1;          //батько - записник
      Name := 'ts' + C;        //Ім'я сторінки - від TabSheet з
                                                        //літерою
      Caption := C;                   //Назва сторінки - літера
                                       //Створюємо Memo: влас-
      Memo := TMemo.Create(PageControl1);  //ник PageControl1,
      Memo.Parent := Page;                          //а батько Page
      Memo.Align := alClient;       //На всю клієнтську область
      Memo.ScrollBars := ssBoth;          //Дві смуги скролінга
                            //Завантажуємо тільки існуючий файл
      if FileExists(C + '.txt') then
         Memo.Lines.LoadFromFile(C + '.txt');
      Button1.Enabled := False;        //Button1 деактивуємо, а
      Button2.Enabled := True;              //Button2 активуємо
    end;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
begin                      //Нижче - цикл за сторінками записника
  for i := 0 to PageControl1.PageCount - 1 do
                //Перетворюємо до TMemo, бо на сторінці є тільки
          //Memo-поле, і перевіряємо наявність змін на сторінці
    if (PageControl1.Components[i] as TMemo).Modified then
                   //Якщо зміни були, зберігаємо рядки Memo-поля
              //у файлі з ім'ям, зумовленим заголовком сторінки
      (PageControl1.Components[i] as TMemo).Lines.SaveToFile(
                      PageControl1.Pages[i].Caption + '.txt');
  Close;
end;
```

Зазначимо, що текст `TForm1.Button2Click` буде коректним тільки за відсутності в компонента `PageControl1` інших компонентів, крім `TMemo`. У противному разі конструкцію

```
    if (PageControl1.Components[i] as TMemo).Modified
```

потрібно буде замінити конструкцією

```
    if (PageControl1.Components[i] is TMemo)
      and (TMemo(PageControl1.Components[i])).Modified
```

Крім того, у програмі не передбачене збереження змін при виході з неї за допомогою системного меню.

У компонента `TPageControl` є два досить часто використовуваних методи – `FindNextPage` та `SelectNextPage`.

Метод

```
function FindNextPage (CurPage: TTabSheet;
   GoForward, CheckTabVisible: Boolean): TTabSheet;
```

дозволяє відшукати наступну сторінку відносно сторінки, заданої першим параметром звертання. Зміст його параметрів: `CurPage` – поточна сторінка, `GoForward` – напрямок переходу (`True` – наступна сторінка, `False` – попередня), `CheckTabVisible` – ознака селекції сторінок (`True` – врахування тільки сторінок зі значенням властивості `TabVisible` рівним `True`; `False` – розгляд усіх сторінок).

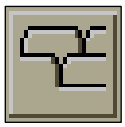
Якщо як `CurPage` зазначити неіснуючу сторінку, то повертається остання або перша сторінка (залежно від напрямку переходу). Наступною щодо останньої сторінки є перша сторінка, а попередньою щодо першої – остання.

Метод

```
procedure SelectNextPage(GoForward: Boolean);
```

робить активною наступну видиму (якщо `GoForward = True`) або попередню видиму (якщо `GoForward = False`) сторінку (щодо активної в даний момент).

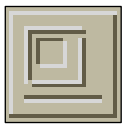
16.10. Блокнот із закладками – компонент TNotebook



Багаторядковий контейнер `TNotebook` визначений у модулі `ExtCtrls` як спадкоємець `TCustomControl`. Його піктографічна кнопка розміщена на вкладці Win 3.1 палітри компонентів. Усі його можливості реалізовані в компоненті `TPageControl`.

Оскільки `TPageControl` є більш досконалим, рекомендується використовувати саме його, а не `TNotebook` (за винятком випадків роботи під Windows 3.1, що на сьогодні малоймовірне).

16.11. Крайка – компонент TBevel



Компонент `TBevel` фактично не є контейнером і має лише оформлювальний характер. Однак зорово він подібний до панелі й служить для візуального виділення та об'єднання в групу інших компонентів або їх розділення. Компонент визначений у модулі `ExtCtrls` як спадкоємець класу `TGraphicControl`, а його піктограма розміщена на вкладці Additional палітри компонентів.

Компонент `TBevel` може створювати на екрані прямокутник, рамку або горизонтальну чи вертикальну лінію залежно від значення його властивості `Shape`, визначеної в такий спосіб:

```
type
  TBevelShape = (bsBox, bsFrame, bsTopLine,
  bsBottomLine, bsLeftLine, bsRightLine, bsSpacer);
property Shape: TBevelShape;
```

Можливими значеннями цієї властивості є:

- ✓ `bsBox` – втиснений або піднятий прямокутник (значення за умовчанням);
- ✓ `bsFrame` – втиснена або піднята рамка;
- ✓ `bsTopLine` – втиснена або піднята верхня лінія;
- ✓ `bsBottomLine` – втиснена або піднята нижня лінія;
- ✓ `bsLeftLine` – втиснена або піднята ліва лінія;
- ✓ `bsRightLine` – втиснена або піднята права лінія;
- ✓ `bsSpacer` – відсутність візуального прояву (порожнє місце).

Стиль компонента `TBevel` визначається властивістю `Style` з можливими значеннями `bsLowered` (втиснений) і `bsRaised` (піднятий):

```
type
  TBevelStyle = (bsLowered, bsRaised);
property Style: TBevelStyle;
```

Запитання для контролю і самоконтролю

1. Який клас служить для створення форми?
2. Як створюється форма на етапі конструювання?
3. Як на етапі проектування форми забезпечити частковий або повний вихід компонента за її розміри?
4. Чи завжди компонент, що виходить за розміри форми, видний на екрані (природно, при значенні `True` у властивості `Visible` компонента)?
5. Як забезпечити доступ до компонента, що повністю виходить за межі форми?
6. Що таке головна форма?
7. Які стилі можуть бути у форм і чим вони відрізняються?
8. Що таке модальне вікно?
9. Що таке немодальне вікно?
10. Чим визначається модальність вікна?
11. Яке призначення властивості `ModalResult`?

12. Назвіть та охарактеризуйте основні властивості форми.
13. Наведіть основні методи форми.
14. На які події реагує форма?
15. Що таке шаблон?
16. Яким способом фрейм підключається до проекту?
17. Як зареєструвати шаблон у палітрі компонентів?
18. Яке призначення панелі?
19. Чим панель групування відрізняється від звичайної панелі?
20. Дайте характеристику різним видам панелей.
21. Як на етапі проектування можна розмістити на панелі зі скролінгом компоненти, що виходять за її розміри?
22. Для чого служить набір закладок?
23. Дайте характеристику блокнота із закладками.
24. Чи є контейнером компонент `TBevel`?

Завдання для практичного відпрацьовування матеріалу

1. На етапі проектування провести експерименти з властивостями форми.
2. Створіть додаток, що передбачає програмну зміну різних властивостей форми (положення, розмір, колірні характеристики та ін.) за командами від кнопки. Проведіть експерименти.
3. Виконайте завдання, аналогічні завданням 1 та 2, для інших контейнерів.
4. Проведіть експерименти з формою, що містить панель зі скролінгом, помістивши на панель кілька компонентів (наприклад, кнопок) таким чином, щоб вони перебували за межами панелі.
5. Створити фрейм, що містить однорядковий редактор з міткою, звичайну кнопку та компонент BitBtn із властивістю Kind, що має значення bkClose. Зареєструвати фрейм як шаблон і використати його при проектуванні додатка для розв'язання якої-небудь найпростішої задачі, наприклад задачі, пов'язаної з перевіркою натурального числа на те, що воно є досконалим. *Досконалим* називають таке натуральне число, що дорівнює сумі всіх своїх дільників, менших самого числа ($6 = 1 + 2 + 3$).
6. Створити та зареєструвати фрейм, що містить фрейм, розроблений у завданні 5, та однорядковий редактор зі своєю міткою. Використати цей фрейм при розв'язанні такої задачі.

   Дано цілі числа p і q. Знайти всі дільники числа p, взаємно прості із числом q. Два цілих числа називаються *взаємно простими*, якщо вони не мають ніяких спільних дільників, крім 1 та –1.

17. КОМПОНЕНТИ ДЛЯ ВВЕДЕННЯ, ВИВЕДЕННЯ ТА ВІДОБРАЖЕННЯ ІНФОРМАЦІЇ

17.1. Компоненти для виведення тексту

17.1.1. Мітка виведення – компонент TLabel

Даний компонент, що досить часто називають ярликом, є графічним компонентом, який служить для відображення невеликих текстових повідомлень і: `TGraphicControl` – `TCustomLabel` – `TLabel`. Піктограма компонента знаходиться на вкладці Standard палітри компонентів, а сам компонент визначений у модулі `StdCtrls`.

Власне відображення тексту здійснюється за допомогою властивості `Caption`, успадкованої від `TControl` (див. п. 15.3.4) з підвищенням рівня видимості до **`published`**. Також успадкована від `TControl` властивість `Color` визначає колір видимої частини мітки (колір фону), а властивість `TFont` – характеристики шрифту, яким виводиться текст (колір, розмір і накреслення).

Описувані нижче властивості успадковані класом `TLabel` від свого безпосереднього прабатька `TCustomLabel`. Зокрема, властивість

```
property AutoSize: Boolean;
```

при її значенні `True` указує, що мітка буде змінювати свої розміри залежно від довжини тексту, вміщеного у властивість `Caption`, а при значенні `False` – забороняє автоматична зміну розмірів.

У мітці можна задавати не тільки однорядковий, але й багаторядковий напис, для чого служить властивість

```
property WordWrap: Boolean;
```

яка дозволяє (`True`) або заборонне (`False`) розрив рядка на межі слова. При цьому висота мітки повинна дозволяти розміщення в ній декількох рядків.

Керування горизонтальним і вертикальним вирівнюванням розташовуваного в мітці тексту виконують відповідно властивості

```
property Alignment: TAlignment;
```

та

```
property Layout: TTextLayout;
```

Тип

```
type
  TAlignment = (taLeftJustify, taRightJustify,
                taCenter);
```

визначає такі можливі варіанти вирівнювання тексту по горизонталі:

- ✓ `taLeftJustify` – текст притискається до лівої межі компонента;
- ✓ `taRightJustify` – текст притискається до правої межі компонента;
- ✓ `taCenter` – текст центрується по горизонталі.

Аналогічно тип

```
type TTextLayout = (tlTop, tlCenter, tlBottom);
```

визначає можливі варіанти вертикального вирівнювання тексту:

- ✓ `tlTop` – текст притискається до верхньої межі компонента;
- ✓ `tlCenter` – текст центрується по вертикалі;
- ✓ `tlBottom` – текст притискається до нижньої межі компонента.

Мітка може бути визначена як прозора, для чого служить властивість

```
property Transparent: Boolean;
```

При `Transparent = True` як колір фону мітки незалежно від значення її властивості `Color` використовується колір батьківського компонента, і мітка, будучи прозорою, не закриває розташовані під нею інші компоненти. Якщо ж `Transparent = False`, то колір мітки визначається її власною властивістю `Color`.

Один із символів, записаних у властивість `Caption` мітки, може бути визначений як символ-акселератор. Його наявністю керує властивість

```
property ShowAccelChar: Boolean;
```

Символ-акселератор – це символ, що визначає клавішу швидкого виконання деякої дії (клавішу швидкого вибору, гарячу клавішу). При

натисканні на таку клавішу в сполученні із клавішею <Alt> (<Alt>+<символ>) виконується деяка заздалегідь визначена дія.

Символ-акселератор активний навіть у випадку, коли компонент, у якому він визначений, є невидимим (але обов'язково зі значенням `True` у властивості `Enabled`).

Якщо властивість `ShowAccelChar` містить значення `True`, то символ `&` у властивості `Caption` мітки вказує, що наступний за ним символ є символом-акселератором, який забезпечує передачу фокуса введення деякому віконному компоненту. При цьому символ-акселератор виводиться з підкресленням. Якщо ж `ShowAccelChar = False`, то символ `&` розглядається як звичайний символ.

Віконний компонент, якому передається фокус введення за допомогою символу-акселератора, визначається властивістю

```
property FocusControl: TWinControl;
```

Так, якщо на формі є мітка Label1 та однорядковий редактор Edit1, то наступні оператори визначають символ `1` як символ-акселератор, що забезпечує передачу фокуса введення компоненту Edit1:

```
Label1.ShowAccelChar := True;
Label1.FocusControl := Edit1;
Label1.Caption :=' Alt+&1 – активізує редактор Edit1';
```

Мітка успадковує всі події класу `TControl`, а також реагує на дві події, успадковані від `TCustomLabel`:

- **property** `OnMouseEnter: TNotifyEvent` – відбувається, коли вказівник миші переміщається над компонентом (міткою);
- **property** `OnMouseLeave: TNotifyEvent` – відбувається, коли вказівник миші залишає поверхню компонента (мітки).

Прикладом застосування події OnMouseEnter є зміна кольору мітки при потраплянні на неї вказівника миші. Як реакції на подію OnMouseLeave може бути, наприклад, скасування дій, виконаних за подієюOnMouseEnter.

17.1.2. Текстова мітка – компонент TStaticText

Компонент `TStaticText` (текстова мітка, статичний текст) служить для виведення однорядкових текстових повідомлень. Він визначений у модулі `StdCtrls`, перебуваючи на нижньому рівні такого ланцюжка віконних компонентів: `TWinControl` – `TCustomStaticText` – `TStaticText`. Піктограма компонента розміщується на вкладці Additional палітри компонентів.

Будучи віконним компонентом, текстова мітка має Windows-вікно, чим вона у першу чергу відрізняється від звичайної мітки. Компонент успадковує властивості, методи й події від класу `TWinControl`, успадкувавши додатково від `TCustomStaticText` властивості `Alignment`, `AutoSize`, `FocusControl` і `ShowAccelChar`, які ідентичні відповідним властивостям мітки `TLabel`. На відміну від `TLabel`, даний компонент може бути обведений рамкою, вигляд якої визначається успадкованим від класу `TCustomStaticText` властивістю

```
property BorderStyle: TStaticBorderStyle;
```

Значення переліченого типу

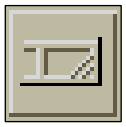
```
type
  TStaticBorderStyle = (sbsNone, sbsSingle, sbsSunken);
```

характеризують можливий вигляд рамки:

- ✓ `sbsNone` – відсутність рамки;
- ✓ `sbsSingle` – наявність рамки завтовшки 1 піксель;
- ✓ `sbsSunken` – піднята межа з подвійною рамкою, що створює ефект вдавлення поверхні (ілюзію рельєфності).

У цілому ж цей компонент аналогічний компоненту `TLabel`, якщо не враховувати те, що в нього відсутні властивості `WordWrap` і `Layout`.

17.1.3. Панель стану – компонент TStatusBar



Панель стану `TStatusBar` (інформаційна панель, рядок стану, рядок статусу) є контейнером рядків, у яких звичайно відбивається інформація про працюючу програму. Компонент `TStatusBar` визначений у модулі `ComCtrls` і є віконним компонентом – спадкоємцем другого рівня компонента `TWinControl`: `TWinControl` – `TCustomStaticText` – `TStaticText`. У палітрі компонентів його піктограма міститься на вкладці Win32.

Виведені в панелі стану рядки розташовуються на окремих панелях усередині панелі стану, яку звичайно розміщають у нижній частині форми, присвоюючи успадкованій від `TControl` властивості `Align` значення `alBottom` (панель стану можна розмістити й усередині іншого контейнера).

У панелі стану є властивість тільки для читання

```
property Canvas: TCanvas;
```

Її можна використовувати при реалізації опрацьовувача події OnDrawPanel, визначеної для даного компонента.

Властивість

```
property SizeGrip: Boolean;
```

дозволяє (`True`) або забороняє (`False`) вставку в панель стану кнопки зміни розміру вікна (значення властивості враховується не завжди).

За допомогою властивості

```
property SimplePanel: Boolean;
```

можна дозволити (`False`) або заборонити (`True`) створення декількох секцій (вкладених панелей) усередині панелі стану.

Фактично властивість `SimplePanel` виконує переключення інформаційної панелі з однопанельного режиму в багатопанельний і назад, тобто навіть при `SimplePanel = True` може існувати кілька секцій, але вони при цьому будуть невидимими, а зробити їх видимими можна за допомогою переключення властивості `SimplePanel` у стан `False`.

Самі вкладені панелі містяться у властивості

```
property Panels: TStatusPanels;
```

Ця властивість є списком, що нумерується від `0` і містить об'єкти-панелі, кількість яких визначається значенням властивості `Panels.Count`.

Доступ до елементів списку `Panels` забезпечується за допомогою його властивості `Items`, що є властивістю-масивом з елементами класового типу `TStatusPanel`. Наприклад, якщо на формі є рядок статусу `StatusBar1`, що містить більше однієї секції, то наступний оператор виведе в другу секцію рядка статусу кількість його секцій:

```
StatusBar1.Panels.Items[1].Text :=
                    IntToStr(StatusBar1.Panels.Count);
```

Щоб створити в рядку статусу вкладену панель, потрібно двічі клацнути мишкою над зображенням рядка статусу на формі чи в Дереві Об'єктів або одноразово клікнути мишкою над кнопкою з трьома крапками біля властивості Panels у вікні Інспектора Об'єктів, результатом чого є відкриття спеціального редактора панелей. Робота в цьому редакторі зводиться до кліків мишкою над кнопками додавання й видалення панелей, а також кнопками переміщення по них (може бути також використане контекстне меню, що відкривається кліком правою кнопкою мишки).

При `SimplePanel = True` у панелі стану відображається текст, записаний у її властивості

```
property SimpleText: string;
```

Якщо ж `SimplePanel = False`, то в кожній із секцій відображається текст, записаний у її властивості

```
property Text: string;
```

яку має кожна із секцій, що містяться в панелі стану.

Одне із застосувань компонента `TStatusBar` – відображення в ньому інформації про те, над яким з компонентів у даний момент перебуває вказівник миші. Для цього використовується властивість

```
property AutoHint: Boolean;
```

Якщо ця властивість має значення `True`, то при розміщенні вказівника миші над яким-небудь компонентом у панелі стану автоматично буде відображатися текст, записаний у властивості `Hint` цього компонента (за умови, що цей текст міститься у властивості `Hint`, а компонент є активним і видимим). Якщо панель стану, у свою чергу, містить кілька вкладених панелей та `SimplePanel = False`, то текст виводиться в першій з них.

Кожна із вкладених панелей (нагадуємо, що вони мають тип `TStatusPanel`), крім згаданої вище властивості `Text`, має такі властивості (перелічуються не всі):

- **property** `Alignment: TAlignment` – визначає вирівнювання тексту (ідентична аналогічній властивості компонента `TLabel`, див. п. 17.1.1);
- **property** `Bevel: TStatusPanelBevel` – визначає стиль рамки секції (див. нижче);
- **property** `Style: TStatusPanelStyle` – визначає спосіб виведення вмісту секції (див. нижче);
- **property** `Width: Integer` – ширина секції в пікселях.

Тип

```
type
  TStatusPanelBevel = (pbNone, pbLowered, pbRaised);
```

визначає такі стилі рамки:

- ✓ `pbNone` – відсутність рамки;
- ✓ `pbLowered` – втиснена рамка;
- ✓ `sbsSunken` – опукла рамка.

Тип `TStatusPanelStyle` є переліченим типом:

```
type
  TStatusPanelStyle = (psText, psOwnerDraw);
```

Якщо у властивість `Style` записане значення `psText` (значення за умовчанням), то вміст секції виводиться автоматично. При установці ж

значення `psOwnerDraw` промальовування здійснюється під час виконання опрацьовувача події OnDrawPanel

Відзначимо, що на компоненті `TStatusBar` можна формувати графічні зображення, оскільки він має канву (див. підрозд. 14.1).

17.2. Компоненти для введення та виведення тексту

17.2.1. Однорядковий редактор тексту – компонент TEdit

Однорядковий редактор тексту `TEdit` (редагований рядок, редаговане текстове поле) є віконним компонентом, призначеним для введення та відображення текстових рядків. Клас `TEdit` визначений у модулі `StdCtrls` і є спадкоємцем компонента `TWinControl`: `TWinControl – TCustomEdit – TEdit`.

Піктограма даного компонента розміщена на вкладці Standard палітри компонентів.

У даного компонента відсутні свої специфічні властивості, методи і події – він успадковує їх або безпосередньо від класу `TCustomEdit`, або через нього від прабатьків вищого рівня.

Основною властивістю цього компонента є властивість `Text` (див. п. 15.3.4):

```
property Text: TCaption;
```

Значення цієї властивості можна змінювати як на етапі проектування, так і на етапі виконання програми. При виконанні додатка властивість `Text` можливо змінювати як програмно (наприклад, виконанням оператора присвоювання), так і безпосередньою зміною вмісту редакторського вікна.

Дані за допомогою компонента `TEdit` вводяться шляхом клавіатурної зміни вмісту редакторського вікна з наступним використанням значення властивості `Text`. Виведення ж даних забезпечується записом у цю властивість значення, що виводиться, у вигляді одного текстового рядка. При цьому нові символи завжди вставляються за символом, що безпосередньо передує миготливому курсору, положенням якого можна керувати.

Компонент не постачений ні смугою скролінга, ні кнопками прокручування. Якщо текст не уміщується в межі вікна, доступ до невидимої частини тексту забезпечується переміщенням миготливого курсору по рядку, оскільки курсор завжди відображається у вікні редактора.

При розв'язанні багатьох з раніше розглянутих задач (починаючи із прикладу 2.2) однорядковий редактор тексту застосовувався для введення даних.

Як при введенні, так і при виведенні даних варто пам'ятати про можливості введення/виведення за допомогою компонента `TEdit` тільки значень типу **`string`** і, отже, про необхідність виконання відповідних перетворень.

Оскільки багато з властивостей, які характерні для компонента `TEdit`, успадковані від класів `TControl` і `TWinControl`, зупинимося тільки на тих з них, що специфічні для даного компонента:

- **`property`** `AutoSelect: Boolean` – при значенні `True` текст, що міститься у вікні, буде автоматично виділятися при одержанні компонентом фокуса введення;
- **`property`** `AutoSize: Boolean` – при значенні `True` встановлює, що висота компонента автоматично змінюється при збільшенні розміру шрифту, яким виводиться текст (ураховується тільки за наявності в компонента рамки);
- **`property`** `BorderStyle: TBorderStyle` – є аналогічною однойменній властивості компонента `TPanel`, розглянутій раніше в підрозд. 16.3;
- **`property`** `CanUndo: Boolean` – властивість тільки для читання, значення `True` якої свідчить про те, що зроблені в тексті зміни можуть бути скасовані методом `Undo` (див. нижче);
- **`property`** `CharCase: TEditCharCase` – визначає автоматичне перетворення всіх літер тексту (у тому числі й символів кирилиці) до малих або великих літер (див. нижче);
- **`property`** `HideSelection: Boolean` – при значенні `False` визначає, що виділення тексту зберігається і при втраті компонентом фокуса введення (значення `True` за умовчанням);
- **`property`** `MaxLength: Integer` – максимальна довжина рядка (при значенні `0` обмеження на довжину рядка знімається);
- **`property`** `Modified: Boolean` – у властивість автоматично записується значення `True`, якщо відбулася зміна тексту;
- **`property`** `PasswordChar: Char` – служить для забезпечення режиму таємності виведення: якщо у властивості записаний будь-який символ, відмінний від `#0`, то цей символ, заміняє кожен із символів при відображенні тексту у вікні (сам текст при цьому не змінюється);

- **property** ReadOnly: Boolean – при значенні True редагування (зміна) тексту неможливе.

Тип

```
type
  TEditCharCase = (ecNormal, ecUpperCase, ecLowerCase);
```

визначають такі значення:

- ✓ ecNormal – відсутність перетворення;
- ✓ ecUpperCase – текст перетворюється до великих літер;
- ✓ ecLowerCase – текст перетворюється до малих літер.

Весь текст або його частину у вікні виведення можна виділити за допомогою лівої кнопки миші або клавіш переміщення курсору при натиснутій клавіші <Shift> (або програмним шляхом). Наявність виділення забезпечує те, що при вставці символів у редакторське вікно відбувається видалення виділеної частини тексту із вставкою символів, що вводяться (виводяться), у те місце, де розташовувалися вилучені символи. Для зняття виділення необхідно або клацнути лівою кнопкою миші над текстом, або натиснути одну з клавіш керування курсором. Три наступні властивості характеризують виділену частину тексту:

- **property** SelLength: Integer – довжина виділеної частини тексту;
- **property** SelStart: Integer – номер останнього символу тексту, за яким зроблене виділення (якщо виділення відсутнє, то в нього записується номер символу, за яким розташований миготливий курсор, а за відсутності тексту – значення 0);
- **property** SelText: **string** – містить виділений текст.

За допомогою зміни властивості SelStart можна виконувати програмне керування положенням миготливого курсору у вікні редактора (нагадаємо, що текст вставляється за курсором). Наприклад, наступний оператор розташує миготливий курсор за другим символом записаного у вікні редактора рядка:

```
Edit1.SelStart := 2;
```

Програмна зміна значення SelStart скасовує виділення тексту, записуючи значення 0 у SelLength та очищаючи SelText. Програмне задання нового значення SelText приводить до заміни цим значенням раніше виділеного тексту. Виділення, якщо воно було, при цьому знімається, а властивість SelText очищується (властивість SelLength дорівнюватиме нулю). Вставка символу у відображуваний у вікні редактора текст приводить до того, що у властивість SelStart автоматично записується номер цього символу, відлічуваний від початку тексту. При видаленні ж

виділеної частини тексту у властивість SelStart записується номер останнього символу, розташованого перед вилученою ділянкою. При вставці символу миготливий курсор автоматично розміщається за вставленим символом, а при присвоюванні нового значення властивості Text курсор установлюється перед першим символом тексту.

Крім методів, успадкованих від своїх прабатьків, компонент TEdit має такі найчастіше використовувані методи:

- **procedure** Clear – видаляє весь текст (записує у властивість Text порожній рядок);
- **procedure** ClearSelection – видаляє виділену частину тексту;
- **procedure** ClearUndo – очищає буфер методу Undo, унеможливлюючи наступне відновлення тексту;
- **procedure** CopyToClipboard – копіює виділений текст у буфер обміну;
- **procedure** CopyToClipboard – копіює виділений текст у буфер обміну і видаляє його з властивості Text;
- **function** GetSelTextBuf(Buffer: PChar; BufSize: Integer): Integer – копіює не більш як BufSize-1 символів виділеного тексту в буфер Buffer і повертає кількість скопійованих символів;
- **procedure** PasteFromClipboard – копіює вміст буфера обміну, заміняючи ним виділений текст або вставляючи його в позицію курсору за відсутності виділення;
- **procedure** SelectAll – виділяє весь текст;
- **procedure** SetSelTextBuf(Buffer: PChar) – копіює вміст буфера Buffer, заміняючи ним виділений текст або вставляючи його в позицію курсору за відсутності виділення;
- **property** Undo – скасовує всі зміни властивості Text, починаючи з останнього звертання до методу ClearUndo, або відновлює початковий текст, якщо звертання до ClearUndo не було.

Основною подією компонента TEdit є подія

```
property OnChange: TNotifyEvent;
```

Подія OnChange відбувається завжди, коли текст, що редагується, можливо, змінився. Опрацьовувач цієї події, якщо він визначений, дозволяє здійснювати відгук на кожну модифікацію тексту компонента.

Однорядковий редактор тексту, природно, може реагувати і на події, визначені в його прабатьках TWinControl і TControl, такі як OnEnter, OnDblClick, OnExit, OnClick та ін.

```
//Приклад 17.1
//Для наявного на формі однорядкового редактора Edit1
//забезпечити можливість введення за допомогою його тільки
//шістнадцяткових цифр без обмеження на довжину рядка.
procedure TForm1.Edit1Change(Sender: TObject);
var
  PosIns: Integer;
  TmpText: string;
begin
  PosIns := Edit1.SelStart;      //Позиція вставленого символу
  if (PosIns <> 0) and not (Edit1.Text[PosIns] in ['A'..'F'] +
                   ['a'..'f'] + ['0'..'9'])
  then begin
    TmpText := Edit1.Text;              //Запам'ятовуємо текст
    Delete(TmpText, PosIns, 1);  //Видаляємо вставлений символ
    Edit1.Text := TmpText;                 //Відновлюємо текст
    Edit1.SelStart := PosIns - 1; //Повертаємо курсор введення
    MessageBeep(MB_OK);                    //Звуковий сигнал
  end;
end;
```

Звичайно на формі над однорядковим текстовим редактором поміщають мітку виведення, що служить для відображення підказки про призначення редактора. У наступному пункті описано більш зручний компонент, що може бути використаний замість `TEdit`.

17.2.2. Редагований рядок з міткою – компонент TLabeledEdit

Редагований рядок з міткою `TLabeledEdit` (однорядковий редактор тексту з міткою), як й `TEdit`, є віконним компонентом, призначеним для введення та відображення текстових рядків. Він розташовується на нижньому рівні такого ланцюжка компонентів: `TWinControl – TCustomEdit – TCustomLabeledEdit – TEdit`. Клас `TLabeledEdit` визначено у модулі `ExtCtrls`, а піктограма компонента розміщена на вкладці Additional палітри компонентів.

Даний компонент є комбінацією компонентів `TEdit` та `TLabel` і позбавляє програміста від необхідності спеціального комбінування цих компонентів з розміщенням їх у спеціальному контейнері.

Будучи спадкоємцем класу `TCustomEdit`, розглянутий компонент має всі властивості, методи та події, які є в `TEdit`, успадковуючи при цьому додаткові характеристики від класу `TCustomLabeledEdit`. До них, насамперед, належать такі властивості:

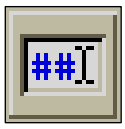
- **property** EditLabel: TBoundLabel – вказівник на мітку редактора (класовий тип TBoundLabel є спадкоємцем класу TCustomLabel і визначає розглянуті в п. 17.1.1 властивості, методи та події, які властиві і класу TLabel);
- **property** LabelPosition: TLabelPosition – положення мітки стосовно редакторського вікна (див. нижче);
- **property** LabelSpacing: Integer – відстань у пікселях від мітки до редакторського вікна.

Тип

```
type
  TLabelPosition = (lpAbove,lpBelow,lpLeft,lpRight);
```

установлює, що мітка може знаходитися вище (lpAbove), нижче (lpBelow), ліворуч (lpLeft) або праворуч (lpAbove) від вікна редактора.

17.2.3. Редагований рядок з маскою форматування – компонент TMaskEdit



Компонент TMaskEdit (поле з маскою введення) служить для введення рядків, що відповідають деякому специфічному формату (шаблону). Коректність даних, що вводяться, перевіряється за допомогою маски, яка кодує дозволені формати (наприклад, дата, час, телефонний номер, ціле число тощо). Даний компонент є віконним компонентом (TWinControl – TCustomEdit – TCustomMaskEdit – TMaskEdit), визначеним у модулі Mask. Його піктограма розміщена на вкладці Additional палітри компонентів.

Результат введення, як і в TEdit, записується у властивість Text. При цьому використовується шаблон, що визначає формат введення:

```
property EditMask: TEditMask;
```

де тип TEditMask є типом **string**.

При порожньому рядку EditMask компонент TMaskEdit є аналогічним компоненту TEdit.

Перевірити те, що маска введення визначена, можна за допомогою звертання до властивості тільки для читання

```
property IsMasked: Boolean;
```

Якщо рядок EditMask визначений, то він складається із трьох полів, розділених символом «крапка з комою». Перша частина цього рядка безпосередньо задає маску введення. Друге поле складається тільки з одного

символу (`0` або `1`) і визначає, що має бути записане у властивість `Text` – результат накладення маски на введений рядок (при `1`) або весь початковий текст (при `0`). Третя частина `EditMask` також містить тільки один символ, що є символом-заповнювачем у позиціях поля маски, призначених для безпосереднього введення (за умовчанням – це символ підкреслення, хоча досить часто вживають велику або малу літеру `X`).

Таблиця 17.1 – Призначення символів маскування

Характеристика позиції введення		Спеціальні символи маски	
Символ	Призначення в масці	Символ	Призначення в масці
`L`	**Повинна** містити літеру	`!`	Придушення всіх початкових пробілів
`l`	**Може** містити літеру	`>`	Всі наступні літери перетворюються до великих
`A`	**Повинна** містити літеру або цифру	`<`	Всі наступні літери перетворюються до малих
`a`	**Може** містити літеру або цифру	`<>`	Скасування перетворення літер
`C`	**Повинна** містити будь-який символ	`\`	Наступний символ вставляється в текст без перетворення
`c`	**Може** містити будь-який символ	`:`	Вставка роздільника Windows для годин, хвилин і секунд
`0`	**Повинна** містити цифру	`/`	Вставка роздільника Windows для складових частин дати
`9`	**Може** містити цифру	`;`	Роздільник полів шаблона
`#`	**Може** містити цифру, а також знаки + або –	`_`	Автоматично вставляється як символ заповнення

Текст, уведений користувачем, разом із символами, явно заданими в масці введення, спочатку записується у властивість

```
property EditText: string;
```

значення якого після його форматування за допомогою маски, що міститься в першому полі `EditMask`, відображається в редакторському вікні (за умови, що властивість `PasswordChar` не встановлює режим таємності введення). Далі виконується запис значення у властивість `Text`: якщо в другому полі маски шаблона введення `EditMask` міститься `0`, то у властивість `Text` переписуються тільки символи, введені користувачем, у противному ж випадку – вміст властивості `EditText`.

Маска введення на етапі проектування може бути введена безпосередньо в Інспекторі Об'єктів. Можна також скористатися редактором коду маски, що відкривається кліком лівою кнопкою миші над кнопкою  в

Інспекторі Об'єктів у правій частині рядка для властивості `EditMask` або вибором команди Input Mask Editor (Редактор Шаблонів) у контекстному меню, що відкривається кліком правою кнопкою мишки над зображенням компонента на формі (рис. 17.1).

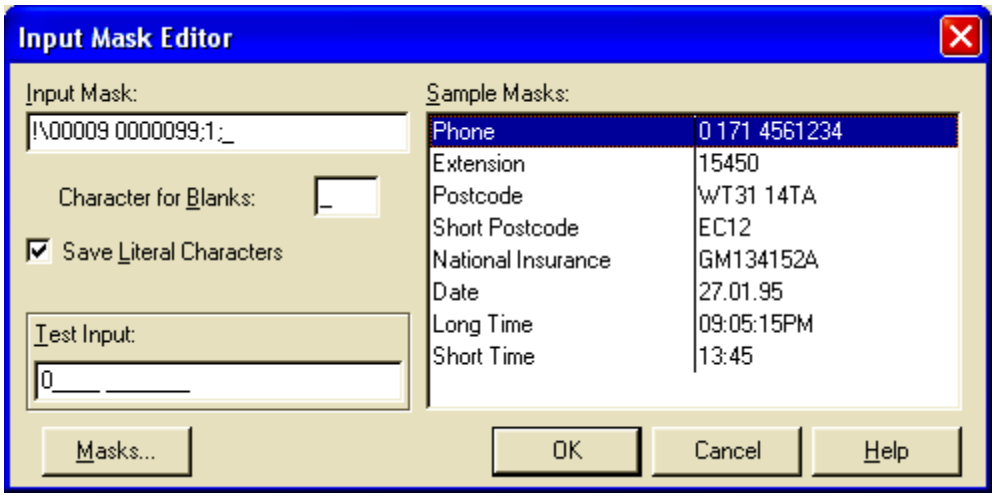


Рис. 17.1. Редагування маски

У Редакторі Шаблонів можна вибрати один зі стандартних шаблонів (вікно Sample Masks), виконати безпосередній набір маски (вікно Input Mask), а також визначити символ заповнення (вікно Character for Blanks). За допомогою ж кнопки Masks можна перейти до вибору стандартних для різних країн шаблонів. Символ заповнення (за умовчанням – це символ підкреслення) автоматично виводиться в редакторському вікні для зазначення позицій, які необхідно заповнити користувачеві.

На етапі проектування можна також визначити початкове значення для властивості `Text`. Найчастіше для цього використовують спеціальний редактор (див. рис. 17.2), що викликається кліком лівою кнопкою миші над кнопкою  праворуч від віконця для значення властивості `Text` в Інспекторі Об'єктів. Цей редактор відкривається також кліком правою кнопкою миші над вікном редактора на формі з вибором у контекстному меню пункту Masked Text Editor (Редактор Тексту Маски).

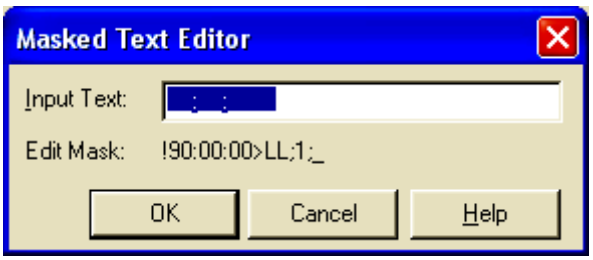


Рис. 17.2. Редактор тексту маски

Будучи текстовим редактором, компонент `TMaskEdit` може бути застосований не тільки для введення рядків, але й для їх виведення.

У прикладі 17.5 компонент `TMaskEdit` використовується для забезпечення введення цілих невід'ємних чисел, що містять не більше чотирьох цифр.

17.2.4. Багаторядковий текстовий редактор – компонент TMemo

Багаторядковийй текстовий редактор `TMemo` (багаторядкове текстове поле) служить для введення, відображення та редагування тексту, що може містити кілька рядків. Клас `TMemo` визначений у модулі `StdCtrls` і розташований на нижньому рівні такого ланцюжка віконних компонентів: `TWinControl` – `TCustomEdit` – `TCustomMemo` – `TMemo`. Піктограма компонента розміщена на вкладці Standard палітри компонентів.

У даному компоненті відображається масив пронумерованих від нуля рядків, що є значенням властивості

```
property Lines: TStrings;
```

На етапі проектування у властивість `Lines` можна записати текстові рядки, що є початковими даними при запуску програми (можна також очистити цю властивість, якщо такі дані відсутні, тим більше, що при розміщенні багаторядкового редактора на формі в цю властивість автоматично заноситься ім'я компонента). Для зміни властивості `Lines` на етапі проектування потрібно або виконати подвійний клік мишкою в Інспекторі Об'єктів над віконцем поруч із ім'ям цієї властивості або клацнути лівою кнопкою мишки над кнопкою  праворуч від цього вікна, у результаті чого відкриється Редактор Списку Рядків (String List Editor), спеціально призначений для редагування властивості `Lines`.

При роботі з компонентом `TMemo` можуть бути використані властивості й методи класу `TStrings`, розглянуті в підрозд. 12.2, такі як `Add`, `Append`, `Clear`, `Count`, `Delete`, `LoadFromFile`, `SaveToFile`, `Strings`, `Text` і т. д.

Оскільки компонент `TMemo` є спадкоємцем компонента `TControl`, він має власну властивість `Text`, і тому звичайно користуються безпосередньо цією властивістю, а не властивістю `Lines.Text`. Аналогічно замість того, щоб застосовувати пронумерований список рядків `Lines.Strings`, користуються тим, що властивість `Lines` сама по собі є

пронумерованим списком рядків, кількість яких визначає властивість `Lines.Count`.

Таким чином, якщо на формі є компонент `Memo1`, а `s` – змінна типу **`string`**, то два наступні оператори дадуть один результат – запис у рядок `s` останнього рядка тексту, що міститься в редакторському вікні:

```
s := Memo1.Lines[Memo1.Lines.Count - 1];
s := Memo1.Lines.Strings[Memo1.Lines.Count - 1];
```

Нагадаємо, що в цьому випадку властивість `Text` інтерпретує весь текст, що міститься у вікні редактора, як один довгий рядок, у якому окремі рядки вихідного тексту відокремлюються стандартною ознакою кінця рядка, визначеною послідовністю з двох символів `#13#10` (див. підрозд. 12.2).

Успадковані від `TCustomEdit` властивості `SelLength`, `SelStart`, `SelText` і методи `ClearSelection`, `SetSelTextBuf` забезпечують можливість роботи з виділеною частиною тексту. При цьому властивості `SelLength`, `SelStart`, `SelText` зберігають значення, отримані в результаті обробки властивості `Text`.

У прикладі 16.3 розглядалася методика завантаження вмісту текстового файлу в багаторядковий текстовий редактор, а в прикладі 6.2 завантажувався текстовий файл у компонент `TMemo` і зберігався в текстовому файлі вміст редакторського вікна.

У компонента є такі специфічні властивості:

- **`property`** `CaretPos: TPoint` – визначає координати миготливого текстового курсору у вікні редактора;
- **`property`** `ScrollBars: TScrollStyle` – визначає наявність у редакторському вікні смуг скролінга (прокручування) тексту (див. нижче);
- **`property`** `WantReturns: Boolean` – при значенні `True` натискання клавіші <Enter> зумовлює перехід до нового рядка у вікні редактора, а при значенні `False` – обробляється операційною системою (для переходу до нового рядка при цьому використовується <Ctrl+Enter>);
- **`property`** `WantTabs: Boolean` – є аналогічною властивості `WantReturns`, але керується клавішею <Tab>;
- **`property`** `WordWrap: Boolean` – якщо відсутня смуга горизонтального скролінга, то при значенні `True` забезпечується розривання рядків у вікні редактора на межах слів, якщо вони не поміщаються в цьому вікні (самі рядки не змінюються, оскільки цей ефект носить тільки оформлювальний характер).

Можливі значення переліченого типу

```
type
  TScrollStyle=(ssNone,ssHorizontal,ssVertical,ssBoth);
```

указують на те, які смуги скролінга вставляються у вікно:

- ✓ `ssNone` – смуги скролінга відсутні;
- ✓ `ssHorizontal` – тільки горизонтальна смуга скролінга;
- ✓ `ssVertical` – тільки вертикальна смуга скролінга;
- ✓ `ssBoth` – обидві смуги скролінга.

```
//Приклад 17.2
//Для наявного на формі редактора Memo1 виконати зчеплення
//всіх рядків, хоча б частково охоплених виділенням.
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
  SelSt, SelLen: Integer;
begin
  s := Memo1.SelText;          //Запам'ятовуємо виділений текст
  SelSt := Memo1.SelStart;          //Позиція початку виділення
  SelLen := Memo1.SelLength;//Довжина виділеної частини тексту
  while Pos(#13#10, s) <> 0 do    //Шукаємо ознаки кінця рядка
    Delete(s, Pos(#13#10, s), 2);              //і видаляємо їх
  Memo1.Text := Copy(Memo1.Text, 1, SelSt)  //Обновляємо текст
                + s + Copy(Memo1.Text, SelSt + 1
                + SelLen, Length(Memo1.Text));
end;
```

У цьому випадку з рядка `s`, у який перший оператор процедури `TForm1.Button1Click` записує виділений текст, видаляються всі підрядки `#13#10`, після чого формується нове значення властивості `Text` (останній оператор). Оскільки значення властивостей `Text` й `Lines` взаємозалежні, при цьому, природно, відбувається відновлення й властивості `Lines`.

17.2.5. Багаторядковий редактор RTF-тексту – компонент TRichEdit

Компонент `TRichEdit` служить для введення, відображення та редагування тексту, поданого в розширеному («багатому») текстовому форматі RTF (Rich Text Format), який дозволяє форматувати як абзаци (вирівнювання, відступи, табуляція, нумерація), так і символи (колір, вид шрифту, накреслення). Клас `TRichEdit` визначений у модулі `ComCtrls` і розміщений на нижньому рівні такого ланцюжка віконних

компонентів: `TWinControl` – `TCustomEdit` – `TCustomMemo` – `TCustomRichEdit` – `TRichEdit`. Піктограма компонента розташовується на вкладці Win32 палітри компонентів.

Даний редактор має тільки вікно і не має жодних компонентів, що забезпечують користувальницький інтерфейс. З цією метою створюються необхідні іетерфейсні компоненти на формі. Клас `TRichEdit` не визначає ніяких нових властивостей, методів і подій, успадковуючи їх від `TCustomRichEdit`.

Приклад програми, у якій застосовано багаторядковий редактор RTF-тексту, наводиться в папці Demos\RichEdit (файл RichEdit.dpr), яка розміщена там же, де і Delphi. На рис. 17.3 показано вигляд форми для цього додатка, а на рис. 17.4 – вигляд працюючого вікна із завантаженим у нього файлом overview.rtf, що зберігається в згаданій папці.

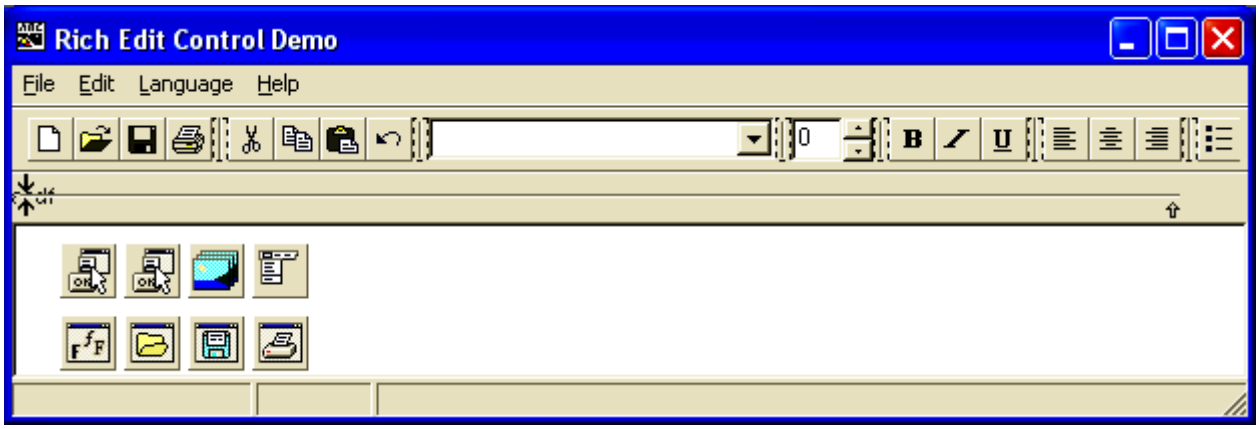


Рис. 17.3. Вигляд форми для проекту Demos\RichEdit\RichEdit.dpr

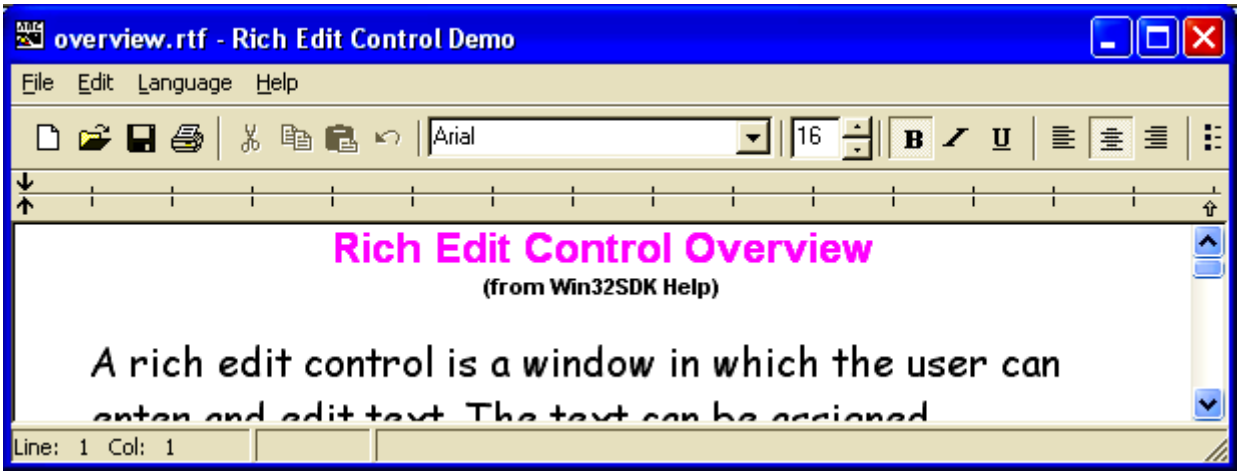


Рис. 17.4. Вікно працюючої програми Demos\RichEdit\RichEdit.exe із завантаженим файлом overview.rtf з папки Demos\RichEdit

Для зміни атрибутів (характеристик) шрифту використовуються як звичайні шрифтові властивості, характерні для класу `TFont`, так і додат-

кові властивості. Атрибути шрифту зберігаються за допомогою об'єктів класу `TTextAttributes`. При цьому атрибути застосовуються як до всього тексту, так і до виділених його частин. Атрибути окремих абзаців зберігаються в спеціально створюваних для кожного абзацу об'єктах класу `TParaAttributes`.

Специфічні методи компонента дозволяють організовувати пошук рядків у тексті з настроюванням на пошук окремих слів і можливістю врахування регістра літер, копіювати виділений текст у буфер, а також установлювати відповідність між розширенням файлу та конвертором тексту.

Компонент може виконувати друк тексту на принтері, визначеним за умовчанням, з форматуванням виведеного тексту в межах аркуша. Для цього використовується метод `Print`, параметр `Caption` якого визначає заголовок друку:

```
procedure Print(const Caption: string);
```

Детально розглядати компонента `TRichEdit` ми не будемо.

17.3. Компоненти для виведення графічної інформації

17.3.1. Зображення – компонент TImage

Компонент `TImage` створює невидимий контейнер для одного окремого зображення – бітового зображення, піктограми, метафайлу, JPEG-зображення (див. п. 14.3.1). Він є спадкоємцем класу `TGraphicControl` і визначений у модулі `ExtCtrls`. Піктограма цього компонента розташована на вкладці Additional палітри компонентів.

Кожне зі згаданих вище зображень може бути завантажене в **published**-властивість

```
property Picture: TPicture;
```

Завантаження зображення в компонент `TImage` можна виконати на етапі проектування програми, для чого в Інспекторі Об'єктів треба клацнути мишкою над кнопкою  праворуч від імені властивості `Picture` і вибрати завантажуваний графічний файл за допомогою Редактора Картинки (Picture Editor), що відкривається в цьому випадку. При цьому зображення буде виводитися навіть тоді, коли додаток ще не виконується. Мало того, завантажене таким способом зображення не просто відображається компо-

нентом, але і зберігається в ньому, що дозволяє передавати додаток користувачу без окремого графічного файлу.

Під час виконання програми зображення в компонент TImage (точніше, у його властивість Picture) завантажується звичайним образом.

```
//Приклад 17.3
//Вивести зображення в компоненті TImage.
```

Помістимо в правій частині форми панель з очищеною властивістю Caption і значенням alRight у властивості Align. Розташуємо на формі вікно відкриття зображення OpenPictureDialog1 (див. підрозд. 21.4), а на панелі кнопку Button1. Розмістимо також на вільному місці форми компонент Image з ім'ям Image1, увівши значення alClient у його властивість Align.

Для виведення зображення з попереднім вибором файлу за допомогою вікна відкриття зображення можна скористатися таким опрацьовувачем події OnClick кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

Наведена вище процедура у випадку, коли розміри зображення більші за розміри компонента Image1, буде обрізати частини зображення, які не поміщаються в розміри компонента. Для забезпечення розтягування або стиснення зображення так, щоб воно повністю займало клієнтську область компонента TImage, використовується властивість

```
property Stretch: Boolean;
```

Якщо присвоїти цій властивості значення True, то при виведенні зображення буде масштабуватися відповідно до розмірів компонента TImage (за умовчанням воно має значення False). Зазначимо, що ця властивість не впливає при виведення піктограм.

Таким чином, наведену вище процедуру TForm1.Button1Click можна дещо модифікувати:

```
procedure TForm1.Button1Click(Sender: TObject);
  if OpenPictureDialog1.Execute then begin
    Image1.Stretch := True;          //Масштабування зображення
    Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
  end;
end;
```

У компонента TImage є ще одна властивість, що керує зміною розмірів зображення:

```
property Proportional: Boolean;
```

Якщо в цю властивість записати значення `True`, то зображення пропорційно зменшується таким чином, щоб його більший розмір міг бути виведений у межах клієнтської області компонента. Занадто малі зображення при цьому не змінюються, тобто зміна властивості `Proportional` може привести **до зменшення** зображення, але не до його збільшення. Як і властивість `Stretch`, дана властивість не впливає на виведення піктограм.

За умовчанням розміри компонента `TImage` є незмінними поза залежністю від розміру зображення, що завантажується в нього. Щоб компонент автоматично змінював свої розміри при зміні розмірів зображення, необхідно встановить у стан `True` таку його властивість:

```
property AutoSize: Boolean;
```

Значенням цієї властивості за умовчанням є значення `False`, і тому зображення може не поміститися в компонент або, навпаки, площа компонента може бути значно більшою від площі картинки.

Якщо не виконувати жодних додаткових дій, то зображення виводиться таким чином, щоб його верхній лівий кут збігся з лівим верхнім кутом компонента. Щоб зображення центрувалося в межах компонента, потрібно присвоїти значення `True` властивості

```
property Center: Boolean;
```

Якщо `Stretch = True` і зображення не є піктограмою, а також якщо `AutoSize = True`, ця властивість ігнорується.

Якщо зображення є бітовим образом, то фон зображення може бути зроблений прозорим, для чого слід записати значення `True` у властивість

```
property Transparent: Boolean;
```

Промальовування зображення виконується з використанням канви, яку визначає властивість

```
property Canvas: TCanvas;
```

Ця властивість є властивістю тільки для читання, причому читання можливе тільки у випадку, коли зображення являє собою бітовий образ. Спроба читання властивості `Canvas`, коли виведений інший тип зображення, приводить до того, що компонент `TImage` збуджує виняток `EInvalidOperation`.

17.3.2. Область малювання – компонент TPaintBox

Компонент TPaintBox створює вікно, що має канву, яка може бути використана для малювання. Це прямий спадкоємець класу TgraphicControl, що визначений у модулі ExtCtrls. Його піктограма розміщена на вкладці System палітри компонентів.

Якщо компонент TImage служить, насамперед, для відображення файлу, що містить зображення, то даний компонент призначений для малювання безпосередньо на канві, яка визначається властивістю Canvas, що успадкована від TGraphicControl. При цьому застосовуються наявні в канви графічні інструменти TPen, TBrush і TFont з усіма їхніми властивостями. Малювання виконує опрацьовувач події OnPaint, яка виникає щоразу при зміні характеристик компонента (наприклад, його розміру). Подія OnPaint визначена в такий спосіб:

```
property OnPaint: TNotifyEvent;
```

```
//Приклад 17.4
//Кліками над кнопкою реалізувати переміщення скругленого
//прямокутника на 1 піксель уліво (вправо) зі зміною
//напрямку руху на протилежний при досягненні
//лівої (правої) межі клієнтської області.
```

Розмістимо на формі кнопку Button1 і вікно для малювання PaintBox1. Для керування переміщенням вікна малювання створимо опрацьовувач події OnClick кнопки Button1, що, використовуючи властивість Tag як перемикач напрямку руху (0 – уліво, 1 – вправо), буде відслідковувати досягнення лівої та правої меж клієнтської області (у даному випадку форми):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 with PaintBox1 do begin
   Left := Left + (2 * Tag - 1);
   if (Left <= 0) and (Tag = 0) then Tag := 1
   else
     if (Left+Width>=Parent.ClientWidth)and(Tag=1)then Tag:=0;
 end;
end;
```

В опрацьовувачі події OnPaint вікна малювання будемо тільки виводити скруглений прямокутник:

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
  with PaintBox1 do begin
   Canvas.RoundRect(0, 0, Width, Height,
                    Width div 10, Height div 10);
```

```
  end;
end;
```

17.3.3. Стандартна фігура – компонент TShape

Компонент `TShape` може використовуватися для відображення графічної інформації за допомогою найпростіших геометричних фігур – прямокутника, квадрата, скругленого прямокутника, скругленого квадрата, еліпса або кола. Його можна застосовувати як в оформлювальних цілях, так і безпосередньо для відображення інформації. Цей компонент визначений у модулі `ExtCtrls` як спадкоємець класу `TGraphicControl`, а його піктограма розміщена на вкладці Additional палітри компонентів.

Вигляд відображуваної фігури визначається властивістю `Shape`:

```
type TShapeType = (stRectangle, stSquare,
  stRoundRect, stRoundSquare, stEllipse, stCircle);
property Shape: TShapeType;
```

Можливими значеннями цієї властивості є:

- ✓ `stRectangle` – прямокутник (значення за умовчанням);
- ✓ `stSquare` – квадрат;
- ✓ `stRoundRect` – скруглений прямокутник;
- ✓ `stRoundSquare` – скруглений квадрат;
- ✓ `stEllipse` – еліпс;
- ✓ `stCircle` – коло.

Компонент має ще дві специфічні властивості:

```
property Brush: TBrush;
property Pen: TPen;
```

Ці властивості відповідно визначають колір і стиль заливки фігури, а також колір, стиль, товщину та режим малювання границь фігур (див. пп. 14.2.2 та 14.2.1).

17.4. Таблиці

17.4.1. Графічна таблиця – компонент TDrawGrid

Компонент `TDrawGrid` служить для графічного відображення масивів даних довільного типу в табличному форматі. Він є візуальним компонентом, що знаходиться на нижньому рівні такого ланцюжка спадкування: `TWinControl` – `TCustomControl` –

TCustomGrid – TCustomDrawGrid – TDrawGrid. Компонент визначений у модулі Grids, а його піктограма розташована на вкладці Additional палітри компонентів.

У графічної таблиці є канва, що визначена властивістю Canvas, з усіма можливостями, надаваними класом TCanvas (див. підрозд. 14.1).

Компонент TDrawGrid має досить багато властивостей, методів і подій, що керують створюваною з його допомогою двовимірною таблицею (сіткою) і використовують табличну організацію відображуваних даних.

Рядки й колонки (стовпці) таблиці нумеруються від 0, причому з рядками зв'язують вісь Y, а зі стовпцями – вісь X. Кількість колонок і рядків таблиці міститься відповідно в наступних її властивостях:

```
property ColCount: LongInt;
property RowCount: LongInt;
```

За умовчанням створювана таблиця має 5 рядків та 5 колонок. Змінюючи значення властивостей RowCount і ColCount, які доступні як при проектуванні, так і при виконанні додатка, можна змінювати розміри таблиці (кількість рядків і колонок). При цьому рядки і колонки видаляються, починаючи від кінця їх списків. Аналогічно проводиться додавання рядків і колонок.

За умовчанням усі рядки таблиці мають однакову висоту в пікселях, що визначається значенням властивості

```
property DefaultRowHeight: Integer;
```

Остання доступна і на етапі проектування, і на етапі виконання програми. Зміна записаного в неї значення (за умовчанням воно дорівнює 24) зумовлює зміну висоти рядків як створюваної, так і вже створеної таблиці.

Шириною колонок таблиці керує властивість

```
property DefaultColWidth: Integer;
```

За умовчанням вона містить значення 64.

У таблиці не обов'язково всі рядки повинні мати однакову висоту. Для визначення висоти кожного з рядків використовується масив RowHeights:

```
property RowHeights[Index: LongInt]: Integer;
```

Цей масив є **public**-властивістю й містить RowCount елементів. При проектуванні форми висота рядків може бути змінена за допомогою мишки; при виконанні ж додатка висота рядка змінюється або програмно за допомогою звертання до елементів масиву RowHeights із вказівкою індексу рядка, або за допомогою мишки переміщенням меж рядків у

фіксованій області таблиці. Якщо висота рядка з номером `Index` не була визначена явно шляхом її зміни за допомогою мишки або задання нового значення `RowHeights[Index]`, то вона визначається значенням властивості `DefaultRowHeight`.

Усе сказане вище стосується і до колонок, ширина кожної з яких визначається **`public`**-властивістю

```
property ColWidths[Index: LongInt]: Integer;
```

Дві наступні властивості, що є **`public`**-властивостями тільки для читання, містять відповідно значення висоти й ширини таблиці:

```
property GridHeight: Integer;
property GridWidth: Integer;
```

Якщо який-небудь із розмірів таблиці занадто великий для його відображення в клієнтській області компонента `TDrawGrid`, то автоматично додаються смуги скролінга (за умови відповідної установки властивості `ScrollBars`, що є аналогічною однойменній властивості компонента `TMemo` – див. п. 17.2.4). Якщо розміри таблиці малі, то частина клієнтської області компонента залишається незаповненою.

Кількість рядків і колонок, що є повністю видимими у зоні прокручування таблиці, містяться відповідно у властивостях

```
property VisibleColCount: Integer;
property VisibleColCount: Integer;
```

У **`public`**-властивостях

```
property LeftCol: Integer;
property TopRow: Integer;
```

містяться відповідно номер першої колонки та номер першого рядка, які відображаються в області прокручування таблиці. Змінюючи їх, можна виконувати програмне прокручування таблиці.

Таблиця може бути розбита на дві частини – фіксовану й робочу. Робоча область служить безпосередньо для відображення інформації, а фіксована – для показу заголовків рядків і колонок. Ручна зміна висоти рядків і ширини колонок, про що згадувалося раніше, можлива тільки при роботі з фіксованою областю таблиці. За умовчанням фіксована область таблиці включає її першу колонку і перший рядок (їх номери дорівнюють нулю). Кількість фіксованих рядків і колонок може бути змінена (як при проектуванні додатка, так і програмно при його виконанні) за допомогою зміни значень відповідно таких **`published`**-властивостей:

```
property FixedCols: Integer;
property FixedRows: Integer;
```

За умови нульового значення якої-небудь із цих властивостей за відповідним виміром таблиця не містить фіксованої області.

За умовчанням фіксована область виділяється кольором `clBtnFace`. Колір фіксованої зони може бути змінений за допомогою зміни значення властивості

```
property FixedColor: TColor;
```

Одна з комірок таблиці є обраною (за умовчанням – це комірка, розташована в лівому верхньому куті робочої області). Якщо таблиця одержує фокус введення, то він автоматично передається саме обраній комірці. При виконанні програми обрану комірку користувач змінює або за допомогою мишки, або за допомогою клавіш зі стрілками. Номери рядка й колонки таблиці, на перетинанні яких знаходиться обрана комірка, містяться відповідно в таких двох **public**-властивостях:

```
property Col: LongInt;
property Row: LongInt;
```

Здійснюючи програмну зміну цих властивостей, можна забезпечувати передачу фокуса введення конкретним коміркам таблиці.

Використовуючи клавішу <Shift>, користувач за допомогою мишки або клавіатури при виконанні програми може виділити групу комірок, доступ до яких забезпечує властивість `Selection`, що визначена в такий спосіб:

```
type
  TGridCoord = record
    X: LongInt;
    Y: LongInt;
  end;
type
  TGridRect = record
    case Integer of
      0: (Left, Top, Right, Bottom: LongInt);
      1: (TopLeft, BottomRight: TGridCoord);
  end;
property Selection: TGridRect;
```

Властивість `Selection` визначає лівий верхній (`Left` та `Top` або `TopLeft`) і правий нижній (`Right` та `Bottom` або `BottomRight`) кути виділеної області усередині таблиці. Фокус введення при цьому одержує комірка, розташована в правому нижньому куті виділеної зони, тобто для `Col` та `Row` автоматично будуть установлені значення `Selection.Right` та `Selection.Top` відповідно.

Товщину ліній (у пікселях), що обмежують комірки таблиці, визначає властивість

```
property GridLineWidth: Integer;
```

Якщо `GridLineWidth = 0`, то лінії відсутні.

Властивість `GridLineWidth` не враховується, якщо властивість `Options` (див. нижче) не включає хоча б одне зі значень `goFixedHorzLine`, `goFixedVertLine`, `goHorzLine` або `goVertLine`.

Властивість `BorderStyle`, що є ідентичною однойменній властивості компонента `TPanel` (див. підрозд. 16.3), визначає рамку компонента `TDrawGrid` (але не таблиці!).

Властивість

```
property EditorMode: Boolean;
```

дозволяє (при значенні `True`) або забороняє (`False`) редагування поточної комірки під час виконання програми. Це властивість не впливає, якщо властивість `Options` (див. нижче) не включає `goEditing` або включає `goAlwaysShowEditor`. Редагування починається після натискання клавіші <F2> або кліку мишею над коміркою й завершується після натискання клавіші <Enter> або кліку мишкою над іншою коміркою. При цьому застосовується спеціалізований нащадок компонента `TMaskEdit`.

За умовчанням при зміні елементів таблиці відбувається їх автоматичне перемальовування. Його можна заборонити, присвоївши значення `False` властивості

```
property DefaultDrawing: Boolean;
```

Якщо в таблиці дозволено здійснення перехід від однієї комірки до іншої за допомогою клавіші <Tab> (для чого потрібно включити значення `goTabs` у властивість `Options`), то до деяких з комірок такий доступ можна заборонити. Кожній колонці таблиці поставлено у відповідність елемент **public**-властивості `TabStops`, що є булівським масивом, у якому значення `True` відповідає колонкам, доступним обходу за допомогою клавіші <Tab>, а значення `False` відповідає недоступним колонкам:

```
property TabStops[Index: LongInt]: Boolean;
```

При обході комірок таблиці за допомогою клавіші <Tab> колонки, яким відповідає елемент масиву `TabStops` зі значенням `False`, будуть пропускатися (для мишки і клавіш зі стрілками такі колонки залишаються доступними).

Властивість Options, що є властивістю-множиною, містить параметри таблиці, які визначають її вигляд і поводження:

```
type
  TGridOption = (goFixedVertLine, goFixedHorzLine,
            goVertLine, goHorzLine, goRangeSelect,
            goDrawFocusSelected, goRowSizing,
            goColSizing, goRowMoving, goColMoving,
            goEditing, goTabs, goRowSelect,
            goAlwaysShowEditor, goThumbTracking);
  TGridOptions = set of TGridOption;
property Options: TGridOptions;
```

Значення, що включаються у властивість Options, мають такий зміст:

- ✓ goFixedVertLine – колонки фіксованої області розділяються вертикальними лініями;
- ✓ goFixedHorzLine – рядки фіксованої області розділяються горизонтальними лініями;
- ✓ goVertLine – колонки робочої області розділяються вертикальними лініями;
- ✓ goHorzLine – рядки робочої області розділяються горизонтальними лініями;
- ✓ goRangeSelect – дозволене виділення групи комірок (ігнорується при включенні в множину Options значення goEditing);
- ✓ goDrawFocusSelected – комірка з фокусом введення виділяється кольором (при множинному виділенні колір комірки з фокусом введення опиниться тим же, що й інших виділених комірок);
- ✓ goRowSizing – дозволена зміна висоти рядків за допомогою мишки при виконанні програми;
- ✓ goColSizing – дозволена зміна ширини колонок за допомогою мишки при виконанні програми;
- ✓ goRowMoving – при виконанні програми дозволене переміщення рядків за допомогою мишки шляхом натискання її лівої кнопки над коміркою фіксованої області з наступним перетаскуванням рядка без відпускання натиснутої кнопки мишки;
- ✓ goColMoving – те ж саме для колонок;
- ✓ goEditing – дозволене редагування вмісту комірок (якщо додатково включене значення goRowSelect ігнорується);

- ✓ `goTabs` – дозволене використання клавіш <Tab> й <Shift+Tab> для обходу комірок;
- ✓ `goRowSelect` – дозволяє виділяти тільки повні рядки із забороною редагування;
- ✓ `goAlwaysShowEditor` – дозволяє редагування комірки, яка має фокус введення (не впливає, якщо в множину `Options` не включене значення `goEditing` або включене значення `goRowSelect`);
- ✓ `goThumbTracking` – дозволяє оновлення при виконанні прокручування (скролінга); якщо це значення не встановлене, оновлення таблиці відбувається тільки після закінчення прокручування (відпусканні повзунка смуги скролінга).

Якщо не враховувати методи `Create` і `Destroy`, що мають стандартне призначення, то специфічними методами компонента `TDrawGrid` є три методи:

- ➢ **function** `MouseCoord(X, Y: Integer): TGridCoord` – повертає табличні координати точки екрана, заданої як параметри (звичайно – це координати точки, на яку вказує миша);
- ➢ **procedure** `MouseToCell(X, Y: Integer; var ACol, ARow: LongInt)` – процедурний аналог методу `MouseCoord`;
- ➢ **function** `CellRect(ACol, ARow: LongInt): TRect` – повертає екранні координати прямокутника комірки, розташованої в колонці `ACol` і рядку `ARow` таблиці.

Центральною подією, на яку реагує компонент `TDrawGrid`, можна вважати подію OnDrawCel, визначену в такий спосіб:

```
type
  TDrawCellEvent = procedure (Sender: TObject;
    ACol, ARow: LongInt; Rect: TRect;
    State: TGridDrawState) of object;
property OnDrawCell: TDrawCellEvent;
```

Ця подія виникає щоразу, коли повинна бути промальована комірка таблиці. Якщо опрацьовувач цієї події не буде визначений, то компонент не зможе заповнити комірки таблиці яким-небудь зображенням або текстом (або тим й іншим одночасно). Промальовування комірки здійснюється з використанням методів канви.

Параметри опрацьовувача події OnDrawCel мають такий зміст:

`ACol`, `ARow` – табличні координати комірки, що промальовується;

`Rect` – прямокутник, що промальовується (місце розташування комірки на полотні);

State – стан комірки, що є множиною, яка має тип

```
type
  TGridDrawState = set of (gdSelected, gdFocused,
                           gdFixed);
```

де значення gdSelected позначає, що комірка виділена, gdFocused – комірка має фокус введення, gdFixed – комірка належить фіксованій області таблиці.

```
//Приклад 17.5
//У комірках ігрового поля розміру 4x4 розміщені неповторювані
//цілі числа від 1 до 15, причому в правому нижньому куті
//знаходиться порожня комірка. За допомогою мишки розташувати
//числа так, щоб вони зростали зліва направо зверху вниз. Кін-
//цеве положення порожньої комірки повинне збігтися з почат-
//ковим. Дозволене переміщення чисел тільки вліво – вправо та
//вгору – вниз у розташовану поруч порожню комірку.
```

Розмістимо на формі графічну таблицю DrawGrid1 для відображення в ній ігрового поля та панель із очищеною властивістю Caption і значенням alBottom у властивості Align. На панелі помістимо кнопку Button1 (запишемо в її властивість Caption текст Гра) і редагований рядок з маскою MaskEdit, над яким розташуємо мітку виведення з текстом Кількість зсувів у її властивості Caption. Для забезпечення можливості введення за допомогою компонента MaskEdit тільки цілих невід'ємних чисел, що мають у своєму записі не більш чотирьох цифр, запишемо в три поля маски форматування (властивість EditMask) відповідно значення 9999, 0 і символ «пробіл» (див. п. 17.2.3).

У секцію **private** опису форми включимо такі поля:

```
List: TStringList;                          //Список "чисел"
ColEmpty,RowEmpty: Integer;//Координати порожньої комірки
```

В опрацьовувачі події OnCreate форми перевизначимо деякі характеристики таблиці, створимо згаданий вище список List і занесемо в нього рядкові подання чисел від 1 до 15, а також порожній рядок, що відповідає порожній комірці ігрового поля, і відобразимо ці значення в таблиці відповідно до кінцевого її вмісту, який повинен бути досягнутий:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  k: Byte;
begin
  DrawGrid1.ColCount := 4;      //Задаємо кількість колонок та
  DrawGrid1.RowCount := 4;                    //рядків таблиці
 //Установлюємо розміри таблиці з урахуванням меж і роздільних
  DrawGrid1.Width := DrawGrid1.DefaultColWidth * 4 + 7;//ліній
```

```
  DrawGrid1.Height := DrawGrid1.DefaultRowHeight * 4 + 7;
  List := TStringList.Create;              //Створюємо список
  for k := 1 to 15 do  //Заповнення списку числами від 1 до 15
    List.Add(IntToStr(k));                 //Поповнення списку
  List.Add('');            //Останній елемент – порожній рядок
  ColEmpty := 3;                  //Порожня комірка – у правому
  RowEmpty := 3;                        //нижньому куті таблиці
  DrawGrid1.Col := ColEmpty;                //Ця комірка буде
  DrawGrid1.Row := RowEmpty;                   //сфокусованою
  DrawGrid1.Enabled := False;    //Спочатку таблиця недоступна
  MaskEdit1.TabOrder := 0;        //Фокус введення у MaskEdit1
end;
```

Для формування вмісту таблиці, при якому буде починатися гра, скористаємося опрацьовувачем події OnClick кнопки Button1. Процес формування початкового вмісту таблиці організуємо шляхом багаторазового переміщення у вільну комірку вмісту однієї із сусідніх по вертикалі або горизонталі комірок. Для вказівки кількості таких переміщень скористаємося редактором MaskEdit (це значення одночасно є максимальною кількістю ходів, за допомогою яких можна одержати розв'язок).

Опрацьовувач події OnClick кнопки Button1 може мати такий текст:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  k: Integer;
  R, C, dR, dC: Integer;
  PreviousCol, PreviousRow: Integer;
begin
  Randomize;
 //Запам'ятовуємо координати порожньої комірки, щоб заборонити
        //її переміщення в положення, яке вона раніше займала
  PreviousCol := DrawGrid1.Col;
  PreviousRow := DrawGrid1.Row;
            //Цикл для багаторазового зсуву порожньої комірки
  for k := 1 to StrToInt('0' + MaskEdit1.Text) do begin
    repeat
      dR := 0; dC := 0;   //Порожня комірка поки не зміщається
        //У якому напрямку будемо намагатися здійснювати зсув?
      if Random(2) = 1 then dR := Random(2)*2-1 //По вертикалі
      else dC := Random(2) * 2 - 1;             //По горизонталі
      R := DrawGrid1.Row + dR;       //Можливі нові координати
      C := DrawGrid1.Col + dC;              //порожньої комірки
    until (R>=0)and(R<DrawGrid1.RowCount)and  //Не можна вихо-
       (C>=0) and (C<DrawGrid1.ColCount) and  //дити з таблиці
            //Не можна також повертатися в попереднє положення
       ((PreviousRow <> R) or (PreviousCol <> C));
                      //Міняємо елементи списку відповідно до
                                 //комірки переміщення порожньою
```

```
    List.Exchange(DrawGrid1.Row * DrawGrid1.ColCount +
                  DrawGrid1.Col, R * DrawGrid1.ColCount + C);
    PreviousCol := DrawGrid1.Col;   //Запам'ятовуємо попереднє
    PreviousRow := DrawGrid1.Row;//положення порожньої комірки
    DrawGrid1.Col := C;    //Передаємо фокус введення порожній
    DrawGrid1.Row := R;                                 //комірки
    ColEmpty := C;       //Запам'ятовуємо координати порожньої
    RowEmpty := R;                                      //комірки
  end;                  //Кінець циклу зсуву порожньої комірки
  DrawGrid1.Enabled := True;          //Тепер таблиця доступна
  MaskEdit1.Visible := False;     //Все зайве робимо невидимим
  Label1.Visible := False;
  Button1.Visible := False;
end;
```

Підкреслимо, що в наведеній вище процедурі в заголовку циклу **for** до значення властивості `MaskEdit1.Text` ліворуч приписується символ `'0'`. Це зроблено для того, щоб уникнути помилки перетворення, яка виникне, якщо рядок `MaskEdit1.Text` виявиться порожнім.

Напишемо також наступні опрацьовувачі подій OnDrawCell й OnMouseDown для графічної таблиці DrawGrid1:

```
procedure TForm1.DrawGrid1DrawCell(Sender: TObject;
  ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);
var
  k: Integer;
begin
                 //Обчислюємо позицію в списку по ACol та ARow
 k := ARow * 4 + ACol;
 with DrawGrid1 do
  Canvas.TextOut(
   (DefaultColWidth+1) * ACol +  //Пропускаємо ACol колонок та
                  //центруємо у колонці текст, що  виводиться
   (DefaultColWidth - Canvas.TextWidth(List[k])) div 2,   //Це
                              //координата X на канві таблиці
   (DefaultRowHeight + 1) * ARow +   //Пропускаємо ARow рядків
                   //і центруємо у рядку текст, що виводиться
   (DefaultRowHeight - Canvas.TextHeight(List[k])) div 2, //Це
                              //координата Y на канві таблиці
    List[k]                  //Текст, що виводиться у комірку
   );
end;

procedure TForm1.DrawGrid1MouseDown(Sender: TObject;
  Button: TMouseButton;  Shift: TShiftState; X, Y: Integer);
var
  ACol, ARow: Integer;                          //Обрана комірка
  k, kEmpty: Integer;    //Індекси обраної й порожньої комірок
```

```
begin
  //Визначаємо координати комірки за координатами курсору миші
  DrawGrid1.MouseToCell(X, Y, ACol, ARow);
                                  //Чи можна переміщати комірку?
  if((ACol + 1 = ColEmpty) or (ACol - 1 = ColEmpty)) and
     (ARow = RowEmpty) or ((ARow + 1 = RowEmpty) or
     (ARow - 1 = RowEmpty)) and (ACol = ColEmpty)
  then begin              //Якщо так, те здійснюємо переміщення
                                   //Обчислюємо позицію в списку
    k := ARow * 4 + ACol;                          //переміщуваної
    kEmpty := RowEmpty * 4 + ColEmpty;   //і порожньої комірок
    List[kEmpty] := List[k];       //Обмінюємо вміст елементів
    List[k] := '';                                 //списку
    with DrawGrid1 do begin
      Canvas.TextOut(      //Виведення тексту в комірку kEmpty
        (DefaultColWidth + 1) * ColEmpty + (DefaultColWidth
        - Canvas.TextWidth(List[kEmpty])) div 2,
        (DefaultRowHeight + 1) * RowEmpty + (DefaultRowHeight
        - Canvas.TextHeight(List[kEmpty])) div 2,
        List[kEmpty]);
                                   //Виводимо порожній рядок ''
      Canvas.TextOut((DefaultColWidth + 1) * ACol
        + (DefaultColWidth - Canvas.TextWidth(List[k])) div 2,
        (DefaultRowHeight + 1) * ARow +
        (DefaultRowHeight - Canvas.TextHeight(List[k])) div 2,
        List[k]);
    end;
    ColEmpty := ACol;  //Фіксуємо координати порожньої комірки
    RowEmpty := ARow;
  end;
end;
```

У програмі поле `List` форми служить для зберігання списку перетворених до рядкового подання чисел, а також порожнього рядка для заповнення порожньої комірки. Цей список одержує початкове наповнення при створенні форми, а його вміст за допомогою опрацьовувача події OnDrawCell таблиці виводиться в її комірки. Для запуску гри використовується кнопка Button1, в опрацьовувачі події OnClick якої таблиця заповнюється таким чином, щоб задача мала розв'язок.

При кліку мишкою над таблицею опрацьовувач події OnMouseDown останньої за допомогою методу `DrawGrid1.MouseToCell` ідентифікує комірку, над якою виконано клік, після чого, якщо обрана комірка може бути переставлена, змінює вмісту списку `List` і перемальовує дві комірки таблиці.

Зазначимо, що в наведених вище текстах підпрограм не передбачено автоматичного контролю того, що розв'язок уже отриманий.

У комірках таблиці можна відображати будь-яку інформацію, що може бути подана в графічному вигляді, у тому числі й зображення, які можуть як зчитуватися з файлів, так і формуватися безпосередньо на канві.

```
//Приклад 17.6
//Малюнок розміром 240x180 пікселів розрізаний на фрагменти
//розміром 30x30 пікселів, що зберігаються в окремих
//BMP-файлах. Вивести ці фрагменти у вигляді таблиці, що
//містить 8 колонок та 6 рядків і за допомогою перестановки
//фрагментів малюнка отримати початкову картинку.
```

Розмістимо на формі графічну таблицю DrawGrid1 і включимо в секцію **private** опису форми такі поля:

```
bmList: TStringList;                    //Список малюнків
Col0, Row0: Integer;  //Координати першої обраної комірки
```

При розв'язанні задачі будемо виходити з вимоги реалізації обміну вмісту двох комірок таблиці після кліків мишкою над цими комірками.

Вважаючи, що файли, у яких зберігаються фрагменти малюнка, мають імена 00.BMP, 01.bmp, ..., 07.bmp, 10.bmp, ..., 57.bmp, напишемо також опрацьовувач події OnCreate для форми й опрацьовувачі подій OnDrawCell та OnMouseDown для графічної таблиці DrawGrid1:

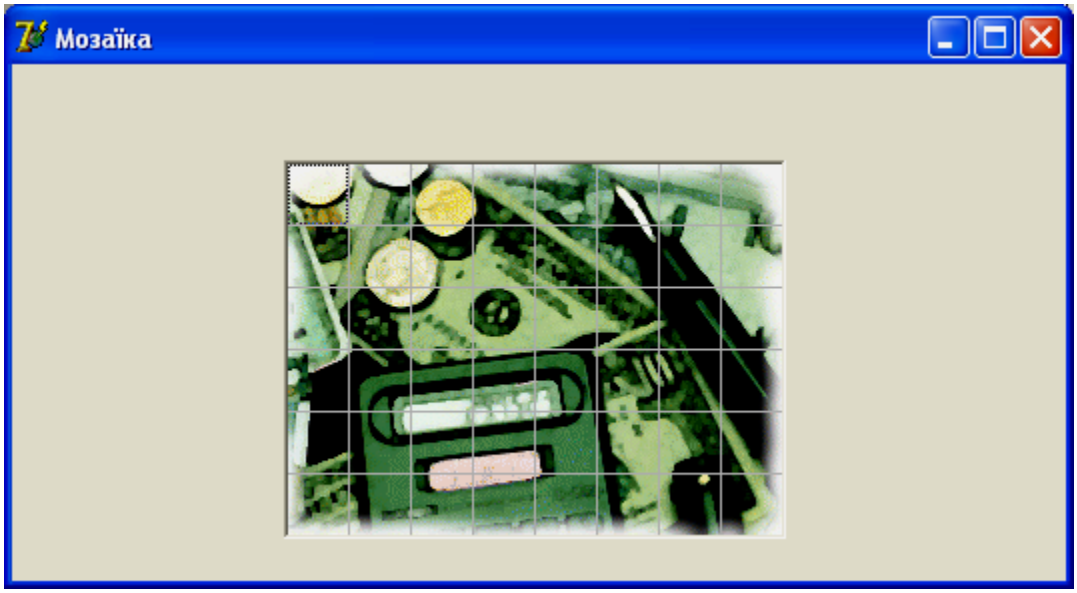
```
procedure TForm1.FormCreate(Sender: TObject);
var
  i, j, k: Integer;
  s: set of Byte;
begin
  with DrawGrid1 do begin
                        //Установлюємо розміри комірок таблиці
    DefaultColWidth := 30;
    DefaultRowHeight := 30;
       //Установлюємо розміри таблиці з урахуванням меж і роз-
    Width := (DefaultColWidth + 1) * ColCount + 3;   //дільних
    Height := (DefaultRowHeight + 1) * RowCount + 3;   //ліній
         //Створюємо список для майбутнього заповнення таблиці
    bmList := TStringList.Create;
    bmList.Capacity := RowCount * ColCount;
    for k := 0 to RowCount * ColCount - 1  do       //Спочатку
      bmList.Add('');        //заносимо в список порожні рядки
    Randomize;
    s := [];     //Множина номерів заповнених елементів списку
    for i := 0 to RowCount - 1 do  //Подвійний цикл заповнення
      for j := 0 to ColCount - 1 do begin   //списку й таблиці
        repeat     //Цикл одержання неповторюваних імен файлів
          k := Random(48);             //Номер елемента списку
        until not (k in s);
        Include(s, k);                  //Запам'ятовуємо номер
```

```
                                  //Заносимо в список ім'я файлу
        bmList.Strings[k] := IntToStr(i)+IntToStr(j) + '.bmp';
        bmList.Objects[k] := TBitMap.Create;       //Створюємо
                       //об'єкт, пов'язаний з елементом списку,
                                 //і завантажуємо в нього малюнок
        (bmList.Objects[k]as TBitMap).LoadFromFile(bmList[k]);
      end;
  end;
end;

procedure TForm1.DrawGrid1DrawCell(Sender: TObject;
   ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);
var
  k: Integer;    //Позиція в списку, що відповідає ACol і ARow
begin
  k := ARow * DrawGrid1.ColCount + ACol;
  DrawGrid1.Canvas.StretchDraw(Rect,
                              (bmList.Objects[k] as TBitMap));
end;

procedure TForm1.DrawGrid1MouseDown(Sender: TObject;
   Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  with DrawGrid1 do begin
    if Tag = 0 then begin            //Якщо це перший вибір, то
      Tag := Tag + 1;
      Col0 := Col;             //запам'ятовуємо номер колонки та
      Row0 := Row;                 //номер рядка обраної комірки
    end
    else begin                              //Обрана друга комірка
       //Обмін елементів списку, що відповідають двом коміркам
      bmList.Exchange(Row0*ColCount+Col0, Row*ColCount+Col);
         //Передаємо фокус введення першій із двох комірок, що
      DrawGrid1.Col := Col0; //вибиралися, для забезпечення її
      DrawGrid1.Row := Row0;  //автоматичного перемальовування
      Tag := 0;
    end;
  end;
end;
```

Після вибору двох комірок таблиці, що будуть переставлятися, і перестановки за допомогою методу `Exchange` відповідних їм елементів списку `bmList` друга обрана комірка перемальовується автоматично, оскільки вона має фокус введення. З метою забезпечення гарантії перемальовування першої з обраних комірок в опрацьовувачі події OnMouseDown наприкінці його роботи здійснюється передача фокуса введення цій комірці, що приводить до генерування події OnDrawCell.

На рис. 17.5, *а* та 17.5, *б* наведено приклади початкового вмісту графічної таблиці й остаточний результат перестановки її комірок (як базовий малюнок узято зображення, що зберігається у файлі C:\Program Files\Common Files\Borland Shared\Images\Splash\256Color\FINANCE.BMP).

а



б

Рис. 17.5. Початкове й результуюче зображення для прикладу 17.6

При редагуванні тексту в комірці з координатами (`ACol`, `ARow`) виникає подія OnGetEditText:

```
type
  TGetEditEvent = procedure (Sender: TObject;
```

```
    ACol, ARow: LongInt; var Value: string) of object;
  property OnGetEditText: TGetEditEvent;
```

За допомогою параметра `Value` опрацьовувач даної події передає текст внутрішньому `TMaskEdit`-редактору.

Крім події OnGetEditText, при редагуванні тексту в комірці з координатами (`ACol`, `ARow`) виникає подія OnGetEditMask:

```
property OnGetEditMask: TGetEditEvent;
```

За допомогою параметра `Value` опрацьовувач цієї події передає маску внутрішньому `TMaskEdit`-редактору. Якщо комірка не має маски редагування, цей параметр повинен передавати порожній рядок.

Щоразу по завершенні редагування тексту в якій-небудь комірці виникає подія OnSetEditText:

```
type
  TSetEditEvent = procedure (Sender: TObject;
  ACol, ARow: LongInt; const Value: string) of object;
property OnSetEditText: TSetEditEvent;
```

Через параметр `Value` опрацьовувач цієї події дістає результат введення або редагування тексту.

Останні три події не виникають, якщо параметр `Options` не містить значення `goEditing`.

Наступні дві події виникають при переміщенні колонки (рядка) таблиці з індексом `FromIndex` у положення, що визначається індексом `ToIndex`:

```
type
  TMovedEvent = procedure (Sender: TObject;
  FromIndex, ToIndex: LongInt) of object;
property OnColumnMoved: TMovedEvent;
property OnRowMoved: TMovedEvent;
```

Ці події не виникають, якщо параметр `Options` не містить значення `goColMoving` (`goRowMoving`).

При спробі виділення комірки з координатами (`ACol`, `ARow`) виникає подія OnSelectCell:

```
type
  TSelectCellEvent = procedure (Sender: TObject;
    ACol, ARow: LongInt; var CanSelect: Boolean)
    of object;
property OnSelectCell: TSelectCellEvent;
```

У параметрі CanSelect опрацьовувач цієї події повідомляє про можливості (True) або неможливості (False) виділення комірки. Установивши цей параметр усередині опрацьовувача даної події в стан False, можна заборонити вибір комірки.

Відразу ж після зміни однієї із властивостей TopRow та LeftCol виникає подія OnTopLeftChanged:

```
property OnTopLeftChanged: TNotifyEvent;
```

Опрацьовувач цієї події пишуть, якщо буде потреба у виконанні спеціальних дій при прокручуванні комірок.

17.4.2. Таблиця рядків – компонент TStringGrid

Компонент TStringGrid служить для створення таблиці для відображення й редагування двовимірних рядкових масивів і є прямим спадкоємцем компонента TDrawGrid. Компонент визначений у модулі Grids, а його піктограма розташована на вкладці Additional палітри компонентів.

Будучи спадкоємцем TDrawGrid, даний компонент має всі властивості, методи й події, які є в TDrawGrid. Специфічними властивостями даного компонента є:

- **property** Cells[ACol, ARow: Integer]: **string** – визначає вміст комірки на перетинанні колонки з індексом ACol і рядка з індексом ARow, забезпечуючи доступ до цієї комірки;
- **property** Cols[Index: Integer]: TStrings – містить всі елементи (кожен з яких є рядком) колонки таблиці з номером Index;
- **property** Rows[Index: Integer]: TStrings – містить всі елементи (кожен з яких є рядком) рядка таблиці з номером Index;
- **property** Objects [ACol, ARow: Integer]: TObject – забезпечує доступ до об'єктів, зв'язаних з комірками таблиці.

Специфічні властивості компонента TStringGrid (насамперед, властивості Cells та Objects) значно спрощують доступ до комірок створюваної таблиці, оскільки в цьому випадку, на відміну від компонента TDrawGrid, доступ до комірок організується за індексами колонки й рядка, як це має місце при роботі зі звичайними двовимірними масивами. Властивості ж Cols і Rows дозволяють трактувати колонки й рядки таблиці як одновимірні масиви.

```
//Приклад 17.7
//Організувати наочне введення двовимірного числового масиву
//із записом сформованого масиву в текстовий файл F.TXT.
```

Розмістимо на формі таблицю рядків StringGrid1 для введення двовимірного масиву дійсних чисел, а також кнопку Button1 для організації виведення вмісту масиву у файл. У властивість `Options` таблиці включимо значення `goEditing` з метою забезпечення можливості редагування її вмісту. В опрацьовувачі події OnCreate форми будемо створювати таблицю потрібного розміру із проставлянням у фіксованій області номерів її рядків і колонок, а за допомогою опрацьовувачів подій OnSetEditText та OnGetEditText таблиці організуємо її заповнення з контролем коректності даних, що вводяться, який ґрунтується на винятках `EConvertError`, `EOverflow` та `EUnderflow`.

Передбачимо також можливість перетаскування рядків і колонок таблиці, для чого включимо у властивість `Options` таблиці значення `goRowMoving` й `goColMoving`, а також створимо для таблиці опрацьовувачі подій OnRowMoved та OnColumnMoved, за допомогою яких буде виконуватися відновлення вмісту фіксованої області таблиці (нумерації рядків і колонок), яка змінюється при перетаскуванні.

Опрацьовувачі зазначених подій можуть мати таку реалізацію:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i, j: Integer;
begin
  DecimalSeparator := '.';
  with StringGrid1 do begin
    repeat //Введення кількості рядків з контролем коректності
      try             //Визначаємо кількість рядків у таблиці
        RowCount := StrToInt(
                  InputBox('Введення розмірів масиву',
                  'Кількість рядків', '1')) + 1;
      except
        on EConvertError do RowCount := 0;
      end;
    until RowCount > 1;
         //Введення кількості колонок з контролем коректності
    repeat
      try            //Визначаємо кількість колонок у таблиці
       ColCount := StrToInt(
                   InputBox('Введення розмірів масиву',
                   'Кількість колонок', '1')) + 1;
     except
       on EConvertError do ColCount := 0;
     end;
```

```
    until ColCount > 1;
    FixedCols := 1;                        //Кількість колонок та
    FixedRows := 1;      //рядків у фіксованій області таблиці
                              //У фіксованій області запишемо
    for j := 1 to RowCount do                              //рядків
      Cells[0, j] := IntToStr(j);
    for i := 1 to ColCount do                          //і колонок
      Cells[i, 0] := IntToStr(i);
  end;
end;

procedure TForm1.StringGrid1SetEditText(Sender: TObject;
    ACol, ARow: Integer; const Value: string);
begin
  //Комірка не повинна бути порожньою, попереду може бути знак
  if not ((Value = '') or (Value = '+') or (Value = '-')) then
    try
      StrToFloat(Value);         //При перетворенні можуть бути
    except    //помилки. При їхнім виникненні очищаємо комірку
      on EConvertError do StringGrid1.Cells[ACol, ARow] := '';
      on EOverflow do StringGrid1.Cells[ACol, ARow] := '';
      on EUnderflow do StringGrid1.Cells[ACol, ARow] := '';
    end;
end;


procedure TForm1.StringGrid1GetEditText(Sender: TObject;
    ACol, ARow: Integer; var Value: string);
begin
  Value := Trim(StringGrid1.Cells[ACol, ARow]);
end;

        //Відновлення нумерації рядків після їх перетаскування
procedure TForm1.StringGrid1RowMoved(Sender: TObject;
    FromIndex, ToIndex: Integer);
var
  i: Integer;
begin              //Рядок може перетаскуватися вгору або вниз
  if ToIndex < FromIndex then begin     //Перетаскування вгору
    for i := ToIndex + 2 to FromIndex do
      StringGrid1.Cells[0, i - 1] := StringGrid1.Cells[0, i];
  end
  else                                    //Перетаскування униз
    for i := ToIndex - 2 downto FromIndex do
      StringGrid1.Cells[0, i + 1] := StringGrid1.Cells[0, i];
  StringGrid1.Cells[0, ToIndex] := IntToStr(ToIndex);
  StringGrid1.Cells[0, FromIndex] := IntToStr(FromIndex);
end;
```

```
      //Відновлення нумерації колонок після їх перетаскування
procedure TForm1.StringGrid1ColumnMoved(Sender: TObject;
    FromIndex, ToIndex: Integer);
var
  i: Integer;
begin          //Колонка може перетаскуватися вліво або вправо
  if ToIndex < FromIndex then begin     //Перетаскування вліво
    for i := ToIndex + 2 to FromIndex do
      StringGrid1.Cells[i - 1, 0] := StringGrid1.Cells[i, 0];
  end
  else                                  //Перетаскування вправо
    for i := ToIndex - 2 downto FromIndex do
      StringGrid1.Cells[i + 1, 0] := StringGrid1.Cells[i, 0];
  StringGrid1.Cells[ToIndex, 0] := IntToStr(ToIndex);
  StringGrid1.Cells[FromIndex, 0] := IntToStr(FromIndex);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i, j: Integer;
  st: string;
  f: TextFile;
begin
       //Спочатку перевіряємо комірки таблиці щодо їх порожні
  for i := 1 to StringGrid1.RowCount - 1 do begin
    for j := 1 to StringGrid1.ColCount - 1 do begin
      if StringGrid1.Cells[j,i]='' then begin //Якщо порожньо,
                           //передаємо фокус введення таблиці
        StringGrid1.SetFocus;
                              // і фокусуємо порожню комірку,
        StringGrid1.Col := j;
        StringGrid1.Row := i;
        Exit;              //після чого виходимо з підпрограми
      end;
    end;
  end;
  AssignFile(f, 'F.TXT');
  Rewrite(f);
      //Пишемо масив у файл за рядками з розділенням елементів
//символом табуляції без порожнього рядка наприкінці виведення
  for i := 1 to StringGrid1.RowCount - 1 do begin
    st := '';
    for j := 1 to StringGrid1.ColCount - 1 do //Формуємо рядок
      st := st + Trim(StringGrid1.Cells[j, i]) + #9;
    Delete(st, Length(st),1);//Видалення останнього символу #9
    Write(f, st);
    if i < StringGrid1.RowCount - 1 then WriteLn(f);
  end;
```

```
  CloseFile(f);
end;
```

Зазначимо, що при виникненні винятків `EConvertError`, `EOverflow` та `EUnderflow` середовище Delphi перехоплює їх, що утруднює виконання запущеної із середовища програми через появу діагностичних повідомлень про помилки. Тому перед запуском даної програми з середовища Delphi варто виконати переналаштовування середовища (див. зауваження наприкінці підрозд. 13.2).

Запитання для контролю і самоконтролю

1. Яке призначення компонента `TLabel`? Стисло охарактеризуйте його.
2. У чому відмінність текстової мітки від звичайної мітки виведення?
3. Для чого служить панель стану й у чому її особливості?
4. Охарактеризуйте компонент `TEdit`.
5. У чому відмінність компонентів `TEdit` і `TLabeledEdit`?
6. Які особливості редактора `TMaskEdit`?
7. Охарактеризуйте компонент `TMemo`.
8. Яке призначення компонента `TImage`? Назвіть його основні властивості.
9. Яке призначення компонента `TPaintBox`?
10. Дайте характеристику компоненту «графічна таблиця».
11. Чим відрізняється таблиця рядків від графічної таблиці? Дайте розгорнуте пояснення.
12. Як виконується виведення графічних зображень у графічну таблицю та у таблицю рядків?

Завдання для практичного відпрацьовування матеріалу

1. За допомогою компонента Memo реалізувати виведення всіх простих чисел, що не перевищують задане натуральне число *n*. Скористатися способом, називаним «Решето Ератосфена».

   Суть способу в наступному. Нехай дана послідовність натуральних чисел 2, 3, …, *n*... Перше просте число 2. Викреслимо його і всі кратні йому числа. Перше із чисел, що залишилися, знову є простим, і можна виконати ті ж самі дії над цим числом і т. д.
2. Розташувати на формі компонент Memo і кнопку для подачі команди на збереження вмісту редакторського вікна у файлі. У нижній частині фор-

ми помістити панель стану для відображення інформації про компонент, що має фокус введення. При натисканні кнопки в рядку панелі стану повинен відображатися текст Збереження. Якщо фокус введення має компонент Memo, то в трьох розділах панелі стану відображати текст Редагування і координати курсору у вікні редактора (номер рядка і номер колонки).

3. За допомогою компонента MaskEdit організувати введення дійсних чисел, записуваних у форматі з фіксованою точкою без порядку. Кількість цифр у цілій частині – не більше чотирьох, а в дробової – не більше шести. Передбачити можливість наявності знака в числа. Як десятковий роздільник використовувати встановлений у Windows роздільник без його перевизначення.
4. Параметричним поданням кривої на площині називаються дві функції $x = x(t)$ і $y = y(t)$, що визначають декартові координати x та y залежно від значення параметра t. Побудувати спіраль із n витками та зовнішнім радіусом r (рис. 17.6). Початковий напрямок спіралі утворює із віссю OX кут α (на рис. 17.6 $\alpha = 0$). Параметричне подання спирали: $x = r\cos t$, $y = r\sin t$, $r = t/2$, $\alpha \le t \le 2n\pi$.

   Примітка. Задати значення r не менше 100 пікселів.

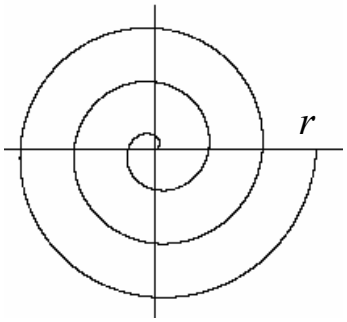


Рис 17.6. Спіраль з трьома витками

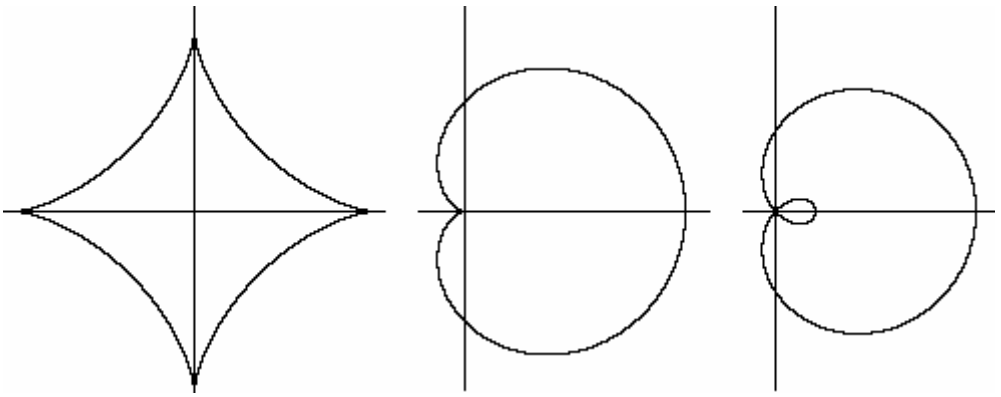
5. Для $t \in [0, 2\pi)$ побудувати криві за заданим параметричним поданням.

   а) Астроїда: $x = a\cos^3 t$, $y = a\sin^3 t$ (див. рис. 17.7, *а*).

   б) Кардіоїда: $x = a\cos t\,(1+\cos t)$, $y = a\sin t\,(1+\cos t)$, $a > 0$ (див. рис. 17.7, *б*).

   в) Равлик Паскаля: $x = a\cos^2 t + b\cos t$, $y = a\cos t\sin t + b\sin t$, $a > 0$, $b > 0$. Розглянути випадки, коли $a > b$ (вигляд кривої наведено на рис. 17.7, *в*), $b \ge 2a$ та $a < b < 2a$.

   г) Еліпс із півосями, що дорівнюють r_1, r_2 і розташовані паралельно осям координат: $x = r_1\cos t$, $y = r_2\sin t$.



а *б* *в*

Рис. 17.7. Астроїда (*а*), кардіоїда (*б*) і варіант равлика Паскаля (*в*)

6. Накреслити візерунок, показаний на рис. 17.8. Візерунок утворений квадратами, сторони самого зовнішнього з яких дорівнюють 200 пікселів і паралельні сторонам компонента Image. Вершини інших квадратів розташовуються в точках на сторонах попереднього квадрата й ділять їх у відношенні $\mu = 0{,}9$ (див. завдання 1 до розд. 14).
7. На компоненті PaintBox побудувати візерунок з трикутників, аналогічний візерунку з завдання 6.
8. Побудувати візерунок з повторюваних у вигляді таблиці 3×3 візерунків з завдання 6 з довжиною сторони першого квадрата, що дорівнює 50.

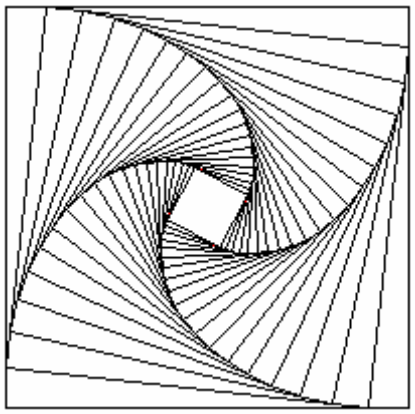


Рис. 17.8. Візерунок

9. Реалізувати завдання з прикладу 17.6 з використанням події OnSelectCell.
10. Реалізувати завдання з прикладу 17.6 за допомогою компонента StringGrid.
11. Дано двовимірний масив цілих чисел розміру $m \times n$, де m – кількість рядків, n – кількість колонок. За допомогою мишки переставити його дві колонки, виконавши їх візуальний вибір.

18. КОМПОНЕНТИ ДЛЯ ПОДАЧІ КОМАНД КЕРУВАННЯ

18.1. Кнопка – компонент TButton

Цей компонент є одним з найбільш часто використовуваних для керування програмами компонентів. Компонент `TButton` визначений у модулі `StdCtrls` і є віконним компонентом: `TWinControl` – `TButtonControl` – `TButton`. У палітрі компонентів його піктограма розміщена на вкладці Standard.

Компонент створює на формі прямокутну кнопку, що має записаний у властивості `Caption` напис (за умовчанням – це ім'я компонента). У працюючій програмі при кліку мишкою над цією кнопкою генерується подія OnClick, на яку програма реагує виконанням деякого алгоритму керування, реалізованого в опрацьовувачі цієї події.

Кнопка може бути оголошена *кнопкою за умовчанням*, для чого необхідно записати значення `True` у її властивість

```
property Default: Boolean;
```

Така кнопка реагує на натискання клавіші <Enter> запуском опрацьовувача події OnClick.

Можливе також оголошення кнопки *кнопкою переривання*, що забезпечується записом значення `True` у властивість

```
property Cancel: Boolean;
```

Кнопка переривання генерує подію OnClick при натисканні клавіші <Esc>.

Ці дві властивості безпосередньо визначені в класі `TButton`, а не успадковані від його прабатьків.

Ще однією специфічною властивістю цього класу є властивість

```
property ModalResult: TModalResult;
```

Як про це говорилося вище в підрозд. 16.1, якщо в кнопки визначена властивість ModalResult і вона міститься в модальному вікні, то клік мишею на ній зумовлює закриття цього вікна з передачею в програму значення ModalResult як результату діалогу, організованого в модальному вікні (див. приклад 16.1).

Будучи компонентом Windows, кнопка завжди має системний колір clBtnFace і не має властивості Color. Вона також не реагує на зміну кольору шрифту, реагуючи в той же час на зміну його стилю й розміру. Крім того, стандартні кнопки не можуть відображати багаторядкові написи. У той же час, звернувшись до API-функцій Windows, можна забезпечити можливість відображення в кнопці багаторядкового напису з автоматичною розбивкою його на декілька рядків. Висота кнопки при цьому повинна бути встановлена таким чином, щоб багаторядковий напис міг бути відображений.

Не вдаючись особливо в особливості використовуваних нижче API-функцій Windows GetWindowLong та SetWindowLong, а також у зміст констант GWL_STYLE й BS_MULTILINE, наведемо текст опрацьовувача події OnCreate форми, що забезпечує можливість відображення в кнопці Button1 багаторядкового напису:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  BtnStyle: LongInt;
begin
  BtnStyle := GetWindowLong(Button1.Handle,GWL_STYLE);
  BtnStyle := BtnStyle or BS_MULTILINE;
  SetWindowLong(Button1.Handle, GWL_STYLE, BtnStyle);
end;
```

Приклади застосування кнопок у програмі наводилися неодноразово в попередніх розділах.

18.2. Графічна кнопка – компонент TBitBtn

Графічна кнопка (кнопка із зображенням) є модифікованою стандартною кнопкою – спадкоємцем компонента TButton, визначеним у модулі Buttons. Піктограма компонента TBitBtn розташована на вкладці Additional палітри компонентів.

Будучи спадкоємцем класу TButton, компонент TBitBtn має всі його властивості, методи та події, у тому числі властивості Default, Cancel і ModalResult, реагуючи, на відміну від TButton, на зміну

кольору шрифту. При цьому за рахунок додаткових властивостей компонент `TBitBtn` дістав нові можливості.

На поверхні графічної кнопки може бути намальоване растрове зображення, причому як це зображення програміст може взяти власну картинку, створену за допомогою якого-небудь графічного процесора. Можна також використати один із графічних образів, що спеціально розроблені для їх розміщення на таких кнопках і зберігаються в папці Program Files\Common Files\Borland Share\Images\Buttons. Виведене зображення визначається властивістю

```
property Glyph: TBitmap;
```

Щоб визначити властивість `Glyph`, необхідно в Інспекторі Об'єктів клацнути мишкою над кнопкою  поруч з ім'ям цієї властивості, відкривши тим самим спеціальний Редактор Картинки (Picture Editor).

Саме зображення формується таким чином, щоб воно являло собою горизонтальну смугу, що містить від одного до чотирьох малюнків розміру 16×16 пікселів. Перший з малюнків відповідає відпущеній кнопці або кнопці, що отримала фокус введення (в останньому випадку кнопка обводиться рамкою). Другий малюнок (якщо він є) виводиться у випадку, коли кнопка не може бути обрана (недоступна). Третій малюнок виводиться при кліку мишкою над кнопкою, а четвертий – над натиснутою та утримуваною кнопкою. Виведені зображення є піктограмами, один з кольорів яких є прозорим. Такими кольором є колір пікселя, розташованого в лівому нижньому куті растрового зображення. На кнопці виводиться тільки один з малюнків, причому загальна їхня кількість установлюється автоматично й записується у властивість

```
property NumGlyphs: TNumGlyphs;
```

де тип `TNumGlyphs` визначений у такий спосіб:

```
type
  TNumGlyphs = 1..4;
```

Дві наступні властивості «керують» відповідно відстанню у пікселях від краю кнопки до піктограми та відстанню від піктограми до напису на кнопці:

```
property Margin: Integer;
property Spacing: Integer;
```

Край кнопки, до якого притискається піктограма, визначається властивістю

```
property Layout: TButtonLayout;
```

Тип `TButtonLayout` є переліченим типом

```
type TButtonLayout = (blGlyphLeft, blGlyphRight,
                      blGlyphTop, blGlyphBottom);
```

з такими можливими значеннями:

- ✓ `blGlyphLeft` – зображення виводиться біля лівої межі кнопки;
- ✓ `blGlyphRight` – зображення виводиться біля правої межі кнопки;
- ✓ `blGlyphTop` – зображення виводиться біля верхньої межі кнопки;
- ✓ `blGlyphBottom` – зображення виводиться в нижній частині кнопки.

Визначено 11 стандартизованих графічних кнопок, що мають написи й відповідні графічні зображення. Деякі з цих кнопок мають символи-акселератори, що викликають їхні опрацьовувачі події OnClick. Різновид стандартизованої кнопки встановлює її властивість

```
property Kind: TBitBtnKind;
```

Тип `TBitBtnKind` є таким переліченим типом:

```
type
  TBitBtnKind = (bkCustom, bkOK, bkCancel, bkHelp,
  bkYes,bkNo,bkClose,bkAbort,bkRetry,bkIgnore,bkAll);
```

Смисл можливих значень типу `TBitBtnKind` і відповідні їм кнопки наведені в таблиці 18.1.

Таблиця 18.1 – Різновиди стандартизованих графічних кнопок

Значення властивості `Kind`	Вигляд кнопки	Особливості
`bkCustom`		Працює як звичайна кнопка, але забезпечується малюнком відповідно до властивості `Glyph`. За умовчанням `ModalResult = mrNone`
`bkOK`		За умовчанням установлені такі значення властивостей: `Default = True`, `ModalResult = mrOk`
`bkCancel`		За умовчанням установлені такі значення властивостей: `Cancel = True`, `ModalResult = mrCancel`
`bkHelp`		За умовчанням `ModalResult = mrNone`. При виборі такої кнопки з'являється вікно з файлом допомоги, вміст якого визначається значенням властивості `HelpContext`

Продовження табл. 18.1

Значення властивості Kind	Вигляд кнопки	Особливості
bkYes		За умовчанням установлені такі значення властивостей: Default = True, ModalResult = mrYes. При кліку мишкою над такою кнопкою всі зміни, зроблені в діалоговому вікні, приймаються
bkNo		За умовчанням установлені такі значення властивостей: Cancel = True, ModalResult = mrNo При кліку мишкою над такою кнопкою всі зміни, зроблені в діалоговому вікні, скасовуються
bkClose		За кліком мишкою кнопка закриває форму, на якій вона розташована
bkAbort		Кнопка аварійного припинення роботи. За умовчанням ModalResult = mrNone
bkRetry		Кнопка повторення. За умовчанням для властивості ModalResult установлене значення mrRetry
bkIgnore		Використовують для ігнорування помилки, що відбулася. За умовчанням ModalResult = mrRetry
bkAll		Використовується для передачі повідомлення «Так для всіх». За умовчанням ModalResult = mrAll

18.3. Кнопка швидкого виклику – компонент TSpeedButton

Компонент TSpeedButton (кнопка для інструментальних панелей) є графічним компонентом, що створює кнопку, яка за умовчанням не має напису. Клас TSpeedButton визначений у модулі Buttons і є безпосереднім спадкоємцем TGraphicControl. У палітрі компонентів його піктограма знаходиться на вкладці Additional.

Кнопки швидкого виклику звичайно розміщають на спеціальних панелях (панелях інструментів) для забезпечення швидкого виклику певних команд меню або установки режимів.

Компонент TSpeedButton в основному повторює властивості й методи компонента TBitBtn, не маючи при цьому вікна. Крім того, цей компонент не створює кнопки за умовчанням і кнопки переривання (у нього відсутні властивості Default і Cancel) та не має властивості ModalResult, що дозволяє передати результат діалогу при закритті модального вікна. За умовчанням напис і зображення на такій кнопці відсутні.

Особливістю таких кнопок є можливість їх об'єднання в групи. Для визначення номера групи, до якої належить дана кнопка, служить властивість

```
property GroupIndex: Integer;
```

За умовчанням значенням цієї властивості є число 0, яке вказує, що кнопка не входить у жодну з груп (яких може не бути взагалі). Щоб включити кнопку в яку-небудь групу, потрібно занести номер групи (який може бути і від'ємним) у властивість GroupIndex.

Одна з кнопок, що входять у групу, може залишатися в натиснутому стані, поки не буде виконано повторного кліку над нею або клік над іншою кнопкою з цієї ж групи. Керує цим процесом властивість

```
property AllowAllUp: Boolean;
```

Якщо AllowAllUp = True, то всі кнопки, що входять в одну групу, можуть залишатися у відпущеному стані. При цьому повторний клік над уже натиснутою кнопкою повертає її у відпущений стан (так само, як і клік над іншою кнопкою групи). Якщо ж AllowAllUp = False, то одна із кнопок групи обов'язково повинна залишатися натиснутою, у зв'язку з чим повторний клік над кнопкою не повертає її у відпущений стан – потрібне клацання над якою-небудь іншою кнопкою групи. Зміна значення властивості для однієї кнопки групи, установлює його нове значення для всіх кнопок, що входять у ту ж саму групу. При GroupIndex = 0 значення властивості AllowAllUp не враховується.

Поточний стан кнопки (натиснута – True, відтиснута – False) визначається значенням властивості

```
property Down: Boolean;
```

Кнопка, що не входить у групу (з GroupIndex=0), не може залишитися в натиснутому стані після клацання над нею. Тому при виході з її опрацьовувача події OnClick у властивість Down такої кнопки автоматично заноситься значення False.

У кнопки є також властивості Caption (за умовчанням порожня), Glyph, Layout (за умовчанням порожня), Margin, NumGlyphs, Spacing, що є ідентичними однойменним властивостям компонента TBitBtn. Є також властивість Transparent, що повторює властивість компонента TLabel.

Специфічною властивістю кнопки швидкого виклику є властивість

```
property Flat: Boolean;
```

При значенні `True` цієї властивості кнопка не має меж, що забезпечують її «випинання», і зображення кнопки у відпущеному стані відсутнє на формі при працюючій програмі (текст і зображення, виведені на поверхні кнопки, при цьому залишаються видимими). Ці межі з'являються тільки при потраплянні курсору миші на поверхню кнопки.

Як зазначалося вище, кнопки швидкого виклику звичайно розміщають на інструментальних панелях, хоча для цих цілей існує спеціальний їх різновид – кнопки панелі інструментів (компоненти `TToolButton`). Особливістю компонента `TToolButton` є те, що він відсутній у палітрі компонентів і розміщається за допомогою контекстного меню на компоненті `TToolBar` (див. далі підрозд. 18.7).

18.4. Головне меню – компонент TMainMenu

Компонент `TMainMenu` створює панель, що містить розділи головного меню форми з розкривним списком команд. Компонент визначений у модулі `Menus` і є невізуальним компонентом – спадкоємцем класу `TMenu`, який, у свою чергу, є безпосереднім спадкоємцем `TComponent`. У палітрі компонентів його піктограма розміщена на вкладці Standard.

Подвійним кліком над зображенням компонента на формі або в Дереві Об'єктів, забезпечується виклик Дизайнера, що служить для конструювання меню. Можливе також використання контекстного меню, викликуваного кліком правої кнопки миші над зображенням компонента на формі з вибором команди Menu Designer (Дизайнер Меню). Цього ж можна домогтися в Інспекторі Об'єктів при обраному компоненті `TMainMenu` кліком над кнопкою  поруч із віконцем для властивості `Items` (див. нижче), яка служить для зберігання посилань на окремі опції (пункти) головного меню.

При розміщенні компонента `TMainMenu` на формі він автоматично одержує ім'я за умовчанням, що формується звичайним для Delphi способом (перший такий компонент отримує ім'я `MainMenu1`) і записується в його властивість

```
property Name: TComponentName;
```

Значення цієї властивості, що фактично має тип **`string`**, можна змінити, не забуваючи при цьому про правила побудови ідентифікаторів.

При завантаженні Дизайнера Меню на екрані з'являється вікно, вигляд якого наведений на рис. 18.1. Прямокутник, що обведений пунктиром, служить для відображення назви створюваної опції (пункту) меню, яке заноситься (зберігається) у властивості `Caption` опції (для формування

опції меню цей прямокутник повинен бути виділений). Якщо введена назва опції задовольняє правилам побудови ідентифікаторів, то в її властивість `Name` заноситься цей рядок з додаванням суфікса, що позначає номер опції (наприклад, `File1`). У противному випадку (наприклад, при використанні символів кирилиці) у властивість `Name` заноситься латинська літера `N` знову ж із числовим суфіксом (наприклад, `N1`). При формуванні назви опції (вмісту властивості `Caption`) може бути визначений символ-акселератор (див. п. 17.1.1). Щоб перейти до формування наступної опції потрібно або натиснути клавішу <Enter> по закінченні введення назви опції, або клацнути мишкою праворуч від прямокутника.

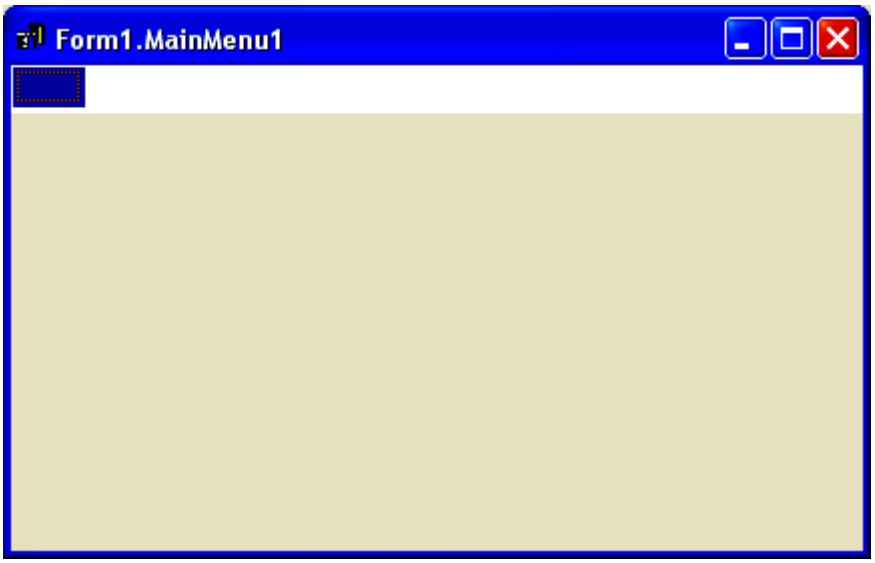


Рис. 18.1. Початковий етап проектування головного меню

За допомогою Дизайнера Меню можна також видаляти будь-які опції та вставляти нові опції. Для видалення опції потрібно виділити її і натиснути клавішу <Del> або скористатися контекстним меню з вибором команди Delete (Видалити). Вставляється нова опція ліворуч від виділеної опції за допомогою клавіші <Ins> або виконанням команди Insert (Вставити) у контекстному меню.

З кожним пунктом меню можна зв'язати кілька підпунктів (підопцій). Для цього потрібно в Дизайнері Меню вибрати пункт меню, для якого формується підменю, і перейти в прямокутник, що при цьому відкриється нижче цього пункту. Далі процес формування підменю принципово нічим не відрізняється від описаної вище методики формування меню. При цьому нові підпункти додаються в кінець їх списку, а вставка виконується вище виділеного підпункту. Підменю може бути розділене на групи за допомогою горизонтальної риси, що вводиться заданням символу «мінус» як назви підпункту.

Кожен з підпунктів може бути зв'язаний з розкривним списком підменю нового рівня. Для створення такого підменю необхідно в Дизайнері Меню при виділеній підопції натиснути клавішу <Ctrl+→> або в контекстному меню вибрати команду SubMenu (Створити Підменю). Підопції, що містять вкладене підменю, при відображенні їх на екрані в правій частині свого рядка містять символ ► (див. рис. 18.2).

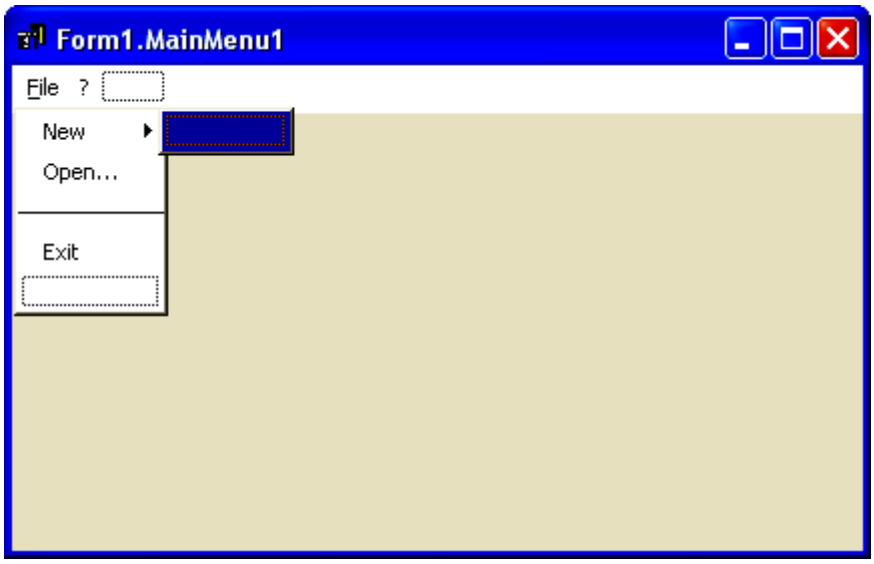


Рис. 18.2. Вікно Дизайнера Меню в процесі проектування меню

Ліворуч від назви підопції може виводитися деяке зображення, що пояснює її дію. Для визначення цього малюнка слід в Інспекторі Об'єктів або в Дереві Об'єктів вибрати потрібну підопцію й визначити значення властивості

```
property Bitmap: TBitmap;
```

Визначення (або корекцію) властивості `Bitmap` найчастіше виконують за допомогою Редактора Картинки (Image Editor), що викликається кліком над кнопкою ... в правій частині рядка для `Bitmap`.

За допомогою Дизайнера Меню можна переміщати елементи меню на нове місце. Для цього потрібно виділити елемент меню лівою кнопкою миші й, не відпускаючи та, перемістити елемент.

Як про це вже говорилося раніше, посилання на елементи меню (так само як і підменю) зберігаються у властивості `Items` компонента `TMainMenu`.

Властивість `Items` є списком об'єктів класу `TMenuItem` – дочірнього компонента класу `TComponent`:

```
property Items: TMenuItem;
```

Кількість опцій у підпорядкованому меню (кількість елементів у масиві `Items`) визначає властивість тільки для читання

```
property Count: Integer;
```

За допомогою властивості `Items` можна звернутися до будь-якого пункту підпорядкованому меню за його індексом.

У об'єктів класу `TMenuItem` є досить велика кількість властивостей, що їх характеризують (зокрема, розглянута вище властивість `Bitmap`).

Так, якщо потрібно, щоб поруч з ім'ям опції з'являлася галочка, то необхідно присвоїти значення `True` властивості

```
property Checked: Boolean;
```

Булівські властивості `Visible` та `Enabled` керують відповідно видимістю та доступністю пунктів меню.

Створення і корекцію головного меню можна здійснювати і динамічно на етапі виконання програми.

Наступний приклад демонструє додавання в головне меню нової опції під час виконання додатка.

```
//Приклад 18.1
//Для наявних на формі головного меню MainMenu1 і кнопки
//Button1 створити опрацьовувач події OnClick компонента
//Button1, що додає до меню недоступний пункт Help.
procedure TForm1.Button1Click(Sender: TObject);
var
  NewItem: TMenuItem;                                  //Опція меню
begin
                                                //Створюємо нову опцію
  NewItem := TMenuItem.Create(Self);
                                             //Додаємо її в список опцій
  MainMenu1.Items.Add(NewItem);
                                           //Визначаємо назву опції
  MainMenu1.Items[MainMenu1.Items.Count - 1].Caption := 'Help';
                                         //Установлюємо недоступність
  MainMenu1.Items[MainMenu1.Items.Count - 1].Enabled := False;
end;
```

Дія, виконувана при виборі пункту меню, визначається його опрацьовувачем події OnClick. Для його створення необхідно клацнути мишкою над опцією в Дизайнері Меню, або виконати подвійний клік над зображенням опції в Дереві Об'єктів.

Нижче розв'язується найпростіша задача, в якій використовуються опрацьовувачі події OnClick для пунктів меню.

```
//Приклад 18.2
//Сконструювати меню з єдиним пунктом Example, що містить
//опції Run та Exit. Опція Run організує підрахунок кількості
//звертань до неї і відображення отримуваного значення в
//заголовку форми, а опція Exit закриває форму.
procedure TForm1.Run1Click(Sender: TObject);
begin
        //Тут Tag і Caption – для форми. Використовуємо Tag як
                  //лічильник кількості звертань до пункту Run
  Tag := Tag + 1;
  Caption := IntToStr(Tag);          //За умовчанням початкове
                                   //значення Tag дорівнює нулю
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;                                          //Закриття форми
end;
```

Природно, для кожного пункту й кожного підпункту меню повинні бути реалізовані опрацьовувачі події OnClick.

18.5. Контекстне меню – компонент TPopupMenu

Компонент `TPopupMenu` Вживається для створення контекстного (спливаючого) меню, що з'являється при кліку над компонентом правою кнопкою миші. Компонент `TPopupMenu` визначений у модулі `Menus` і є безпосереднім спадкоємцем класу `TMenu`, як і компонент `TMainMenu`.

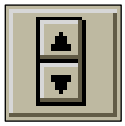
У палітрі компонентів піктограма компонента `TPopupMenu` знаходиться на вкладці Standard.

Контекстне меню можна створити для будь-якого візуального компонента. Для цього слід у властивість `PopupMenu` потрібного компонента помістити ім'я контекстного меню.

У компонента `TPopupMenu` є властивість `AutoPopup`, що має тип `Boolean` і визначає можливість виклику контекстного меню кліком правою кнопкою миші над елементом керування при виконанні додатка. Якщо ця властивість містить значення `True`, то виклик контекстного меню можна виконати за кліком правої кнопки мишки. При значенні ж `False` контекстне меню викликається програмно.

Процес створення контекстного меню за допомогою Дизайнера Меню нічим не відрізняється від аналогічного процесу для головного меню.

18.6. Спарені кнопки – компонент TSpinButton



Даний компонент є безпосереднім спадкоємцем `TWinControl` і визначений у модулі `Spin`. Його піктограма в палітрі компонентів розміщена на вкладці Samples. У довідковій системі Delphi відсутні відомості про цей компонент, у зв'язку з чим для одержання більш-менш повної інформації про його властивості та методи потрібно розібратися у початковому тексті файлу Spin.pas, що зберігається в папці Source\Samples каталогу, який містить Delphi.

Даний компонент не пов'язаний з безпосереднім регулюванням якої-небудь величини і може використовуватися як пара кнопок. Кліки над його верхньою та нижньою кнопками генерують події OnUpClick та OnDownClick відповідно. Для них обов'язково повинні бути написані опрацьовувачі подій.

За умовчанням на поверхні кнопок виводяться зображення стрілок. Виведені зображення можуть бути замінені шляхом зміни властивостей

```
property UpGlip: TBitmap;
property DownGlip: TBitmap;
```

Для визначення малюнків, з якими зв'язані кнопки, звичайно використовують Редактор Картинки (Picture Editor), що викликається з Інспектора Об'єктів кліком над кнопкою  у правій частині рядка для кожної зі згаданих вище властивостей. При цьому малюнки можуть містити від одного до трьох зображень – для кнопки у звичайному, натиснутому та недоступному стані.

Звичайно даний компонент зв'язують з деякими іншими компонентами для регулювання значень яких-небудь величин.

18.7. Панель інструментів та інструментальні кнопки – компоненти TToolBar та TToolButton

Компонент `TToolBar` є контейнером з інструментальними кнопками – компонентами `TToolButton` (загалом кажучи, на такій панелі можна розміщати будь-які елементи керування). Він визначений у модулі `ComCtrls` і є спадкоємцем другого рівня класу `TWinControl` (`TWinControl – TToolWindow – TToolBar`). Його піктограма знаходиться на вкладці Win32 палітри компонентів.

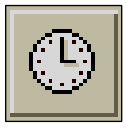
Інструментальні кнопки `TToolButton` багато в чому повторюють можливості кнопок швидкого виклику `TSpeedButton`, будучи, як і останні, дочірніми компонентами класу `TGraphicControl`. Однак, на

відміну від TSpeedButton, компонент TToolButton може бути розміщений тільки усередині панелі TToolBar, у зв'язку з чим він відсутній у палітрі компонентів. Вставка інструментальної кнопки виконується після кліку правою кнопкою миші над зображенням компонента TToolBar на формі або в Дереві Об'єктів з вибором у контекстному меню, що відкриється при цьому, команди New Button (Нова Кнопка). Інші елементи керування розташовують в панелі інструментів звичайним чином. Компоненти, розміщені в панелі інструментів, можуть поділятися на групи відповідно до їх функціонального призначення. Для цього в контекстному меню замість команди New Button необхідно виконати команду New Separator (Новий Роздільник), яка забезпечує вставку роздільника, що є спеціальною кнопкою.

Елементи керування, розміщені в панелі TToolBar, мають такі особливості:

- однакова висота;
- автоматично переміщаються на інший рядок панелі у випадку неможливості їх розташування в поточному рядку (керується булівською властивістю Wrapable компонента TToolBar);
- роздільники втримують кнопки на місці та групують їх візуально і логічно;
- наданням значення True властивості Flat компонента TToolBar усім інструментальним кнопкам може бути забезпечена здатність «випинання» (за аналогією зі TSpeedButton, див. підрозд. 18.3);
- мають властивості, що забезпечують керування причалюванням;
- можуть мати на поверхні напис, а також малюнок.

18.8. Таймер – компонент TTimer



Компонент TTimer служить для відліку часу, що пройшов з моменту його вмикання. Він визначений у модулі ExtCtrls і є безпосереднім спадкоємцем TControl, будучи невидимим компонентом.

Піктограма компонента TTimer розміщена на вкладці System палітри компонентів.

Якщо усі раніше розглянуті в цьому підрозділі компоненти забезпечують подачу команд користувача, то даний компонент формує команди автоматично. Будучи включеним, таймер генерує подію OnTime, що визначається властивістю

```
property OnTimer: TNotifyEvent;
```

Збудження події має періодичність, яка визначається значенням властивості

```
property Interval: Cardinal;
```

Властивість `Interval` за умовчанням дорівнює 100 мілісекунд і може змінюватися як на етапі проектування додатка, так і програмно під час його виконання. Значення цієї властивості, яким би воно не встановлювалося, завжди кратне так званому *тику* – інтервалу часу, з яким спрацьовує системний таймер. Тривалість тику залежить від версії операційної системи. Один тик для Windows NT/2000/XP дорівнює 10 мс, а для Windows 95/98/Me – 55 мс.

Відлік часу таймером починається з його вмикання, що здійснюється заданням значення `True` його властивості

```
property Enabled: Boolean;
```

При присвоєнні цій властивості значення `False` таймер вимикається.

Визначивши опрацьовувач події OnTime, можна забезпечити подачу команд на виконання тих або інших дій з періодичністю, яка установлюється значенням властивості `Interval`, за умови, що властивість `Enabled` має значення `True`. Присвоєнням властивості `Enabled` значення `False` можна відмовитися від подачі команд.

```
//Приклад 18.3
//Створити найпростіший тренажер клавіатури. На екран із
//заданим періодом T1 виводяться латинські літери. До появи
// нової літери користувач повинен встигнути натиснути
//відповідну клавішу.  Тривалість тесту визначається
//значенням T2. По закінченні тесту у вигляді рядків
//"генерована літера - уведена літера" вивести підсумки
//роботи з додатковим повідомленням про кількість
//правильно й неправильно натиснутих клавіш.
```

Створимо форму з однорядковим редактором для відображення літер, які генеруються, Memo-компонентом для виведення результатів, кнопкою Button1 і двома таймерами (Timer1 – для керування процесом тестування, Timer2 – для забезпечення генерування нових літер). Змінимо для компонентів значення деяких з їх властивостей:

- Однорядковий редактор:
  Name — edLetterOutput1
- Memo-компонент:
  Name — mmOutput1
- Таймер Timer1, що призначає час тестування:
  Interval — 5000

- Таймер Timer2, що призначає періодичність появи літер: Interval — 500

Включимо також у секцію **private** опису форми оголошення двох полів:

```
QuantityTrue: Integer;   //Кількість правильних натискань
Letter: Char;                          //Генерована літера
```

В опрацьовувачі події OnCreate форми виконаємо установку початкових значень властивостей компонентів:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  Timer1.Enabled := False;                  //Вимикаємо Timer1
  Timer2.Enabled := False;                  //Вимикаємо Timer2
  mmOutput1.ScrollBars := ssBoth;
  mmOutput1.Clear;
  edLetterOutput1.Clear;
  mmOutput1.Visible := False;
  edLetterOutput1.Visible := True;
  Form1.KeyPreview := True;       //Форма першою отримує події
end;                                              //від клавіатури
```

В опрацьовувачі події OnClick кнопки, призначеної для запуску поточного тесту, виконаємо генерування першої літери і запуск таймерів:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Distance: Byte;    //Відстань між літерами в кодовій таблиці
begin
  mmOutput1.Clear;
  mmOutput1.Visible := False;
  edLetterOutput1.Visible := True;
  //Нижче, якщо Random дасть 1, одержимо Distance=0, у резуль-
    //таті чого буде генерована велика літера, інакше – мала
  Distance := Byte(Random(1) = 0) * (Ord('a') - Ord('A'));
                                              //Генеруємо літеру
  Letter := Char(65 + Distance + Random(26));
                        //Властивість Tag форми використовуємо
  Tag := 1;                           //для підрахунку всіх літер
  edLetterOutput1.Text := Letter;      //Виводимо першу літеру
  QuantityTrue := 0;
  Button1.Enabled := False;
  Timer1.Enabled := True;                   //Запускаємо Timer1
  Timer2.Enabled := True;                   //Запускаємо Timer2
end;
```

Опрацьовувач події OnTime першого таймера виводить результати тестування і вимикає обидва таймери:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Timer1.Enabled := False;                    //Вимикаємо Timer1
  Timer2.Enabled := False;                    //Вимикаємо Timer2
  Button1.Enabled := True;
  mmOutput1.Lines.Add('Правильно натиснуто ' +
    IntToStr(QuantityTrue) + ' клавіш з ' + IntToStr(Tag));
  mmOutput1.Visible := True;
  edLetterOutput1.Visible := False;
end;
```

Опрацьовувач події OnTime другого таймера генерує нову літеру з її виведенням і збільшенням значення лічильника Tag:

```
procedure TForm1.Timer2Timer(Sender: TObject);
var
  Distance: Byte;    //Відстань між літерами в кодовій таблиці
begin
  mmOutput1.Lines.Add(Letter + ' -');
  Distance := Byte(Random(1) = 0) * (Ord('a') - Ord('A'));
                                              //Генеруємо літеру
  Letter := Char(65 + Distance + Random(26));
  Tag := Tag + 1;       //Збільшуємо значення лічильника літер
  edLetterOutput1.Text := Letter;             //Виводимо літеру
end;
```

Під час виконання програми натискання будь-якої алфавітно-цифрової клавіші приводить до збудження події OnKeyPress. Оскільки при відкритті форми її властивість KeyPreview було встановлено в стан True, форма отримує цю подію до передачі її в елемент із фокусом введення, у результаті чого саме її опрацьовувач цієї події буде виконуватися в першу чергу:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
var
  Distance: Byte;    //Відстань між літерами в кодовій таблиці
begin
         //Нижче, якщо клавіша натиснута правильно, збільшуємо
  if Key = Letter then Inc(QuantityTrue);//значення лічильника
  mmOutput1.Lines.Add(Letter + ' - ' + Key);
  Distance := Byte(Random(1) = 0) * (Ord('a') - Ord('A'));
                                              //Генеруємо літеру
  Letter := Char(65 + Distance + Random(26));
  Tag := Tag + 1;       //Збільшуємо значення лічильника літер
  edLetterOutput1.Text := Letter;             //Виводимо літеру
  Timer2.Enabled := True;              //Перезапускаємо Timer2
    //Підмінюємо введений символ, для заборони його виведення,
  Key := #0; //якщо один з редакторів дістане фокус введення
end;
```

Відзначимо, що **var**-параметр `Key` усередині цієї процедури дістає значення `#0`, результатом чого є те, що в компонент із фокусом введення замість символу, що відповідає клавіші, яка натискалася, буде передаватися символ `#0`, і символ, що вводився, на екрані відображатися не буде.

Запитання для контролю і самоконтролю

1. Яке призначення компонента `TButton`? Дайте йому коротку характеристику.
2. Що таке кнопка за умовчанням?
3. Що таке кнопка переривання?
4. У чому відмінність компонентів `TButton` і `TBitButton`?
5. Які різновиди стандартизованих графічних кнопок є в Delphi?
6. Охарактеризуйте кнопку швидкого виклику.
7. Як формується група кнопок швидкого виклику?
8. Які з кнопок (`TButton`, `TSpeedButton`, `TBitButton`) можна наділити здатністю зникати та випинатися залежно від того, перебуває чи ні курсор миші над її поверхнею? Як це забезпечити?
9. Опрацьовувачі яких подій повинні обов'язково створюватися для головного меню?
10. У чому особливості спарених кнопок? Опрацьовувачі яких подій повинні обов'язково створюватися для цього компонента?
11. Для чого служить компонент `TToolBar`?
12. Де розміщаються і для чого служать кнопки класу `TToolButton`?
13. Опишіть процес створення головного меню при проектуванні додатка.
14. Що таке контекстне меню і як воно створюється?
15. Дайте характеристику компоненту «таймер».

Завдання для практичного відпрацьовування матеріалу

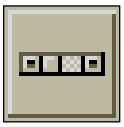
1. Зобразити на формі секундомір. Запуск секундоміра виконувати при натисканні наявної на формі кнопці швидкого запуску SpeedButton, що входить у групу з двох кнопок. При повторному натисканні цієї ж кнопки має припинятися процес візуального відображення відліку часу без відключення таймера та з продовженням відображення при новому натисканні тієї ж кнопки. Натискання другої кнопки швидкого запуску повинне приводити до скидання секундоміра в нульовий стан. Повторне натискання цієї кнопки без попереднього натискання першої

з них не повинне мати фактичної дії. Напис на поверхні першої кнопки має змінюватися, відображаючи один з текстів «Пуск» або «Продовжити» залежно від стану кнопки – натиснута або відпущена. Напис на поверхні другої кнопки – «Скидання».

2. Виконати завдання 1 з використанням замість групи з двох компонентів SpeedButton двох звичайних або графічних кнопок.
3. Виконати завдання 5 з розд. 17, доповнивши його вимогою вибору виду графіків за допомогою головного меню, що реалізує вибір створюваного графіка (тут можна використати змінні процедурного типу).
4. Створити форму з компонентом Memo. Для Memo-компонента побудувати контекстне меню, що дозволяє керувати наявністю смуг скролінга.
5. Створити форму з панеллю в нижній її частині та компонентом Memo, що має задані за умовчанням властивості і розташований поза панеллю. На панель помістити дві спарені кнопки, перша з яких повинна керувати положенням Memo-компонента по горизонталі, а друга – по вертикалі. Передбачити неможливість виходу Memo-компонента за межі клієнтської області форми.
6. До умови завдання 1 додається вимога розміщення на формі інструментальної панелі з двома інструментальними кнопками, що виконують ті ж самі дії, що і дві кнопки швидкого запуску. Забезпечити взаємодію (взаємний вплив) інструментальних кнопок і кнопок швидкого запуску.

19. КОМПОНЕНТИ ДЛЯ ВІДОБРАЖЕННЯ ТА КЕРУВАННЯ ЗНАЧЕННЯМИ ВЕЛИЧИН

19.1. Смуга прокручування – компонент TScrollBar



Даний компонент створює смугу з кнопками для прокручування та бігунком і служить для організації плавної зміни деякої цілочислової величини. Компонент визначений у модулі `StdCtrls` і є віконним компонентом – безпосереднім спадкоємцем `TWinControl`. Його піктограма в палітрі компонентів розміщена на вкладці Standard.

Керована за допомогою компонент `TScrollBar` величина повинна бути зв'язана з властивістю `Position` цього компонента:

```
property Position: Integer;
```

Зміна значення цієї властивості відображається зміною положення бігунка в зображенні компонента на формі. У свою чергу, при зміні положення бігунка змінюється значення властивості `Position` (але не зв'язаної з ним величини – її зміну потрібно виконувати безпосередньо).

Мінімальне і максимальне значення діапазону зміни властивості `Position` визначають відповідно такі властивості:

```
property Min: Integer;
property Max: Integer;
```

Одночасну установку значень усіх трьох перерахованих властивостей виконує метод

```
procedure SetParams(APosition, AMin, AMax: Integer);
```

Наступні дві властивості визначають відповідно «великий» та «малий» зсуви бігунка:

```
property LargeChange: TScrollBarInc;
property SmallChange: TScrollBarInc;
```

Тут тип TScrollBarInc є відрізком:

```
type
  TScrollBarInc = 1..32767;
```

«Великий» зсув бігунка – це зсув, що відбувається при кліку лівою кнопкою миші поруч із кінцевою кнопкою або натисканні клавіш <Page Up> та <Page Down>, а «малий» – при кліку над кінцевою кнопкою або натисканні однієї з клавіш зі стрілками. При цьому зсув зумовлює зміну властивості Position.

За умовчанням компонент має вертикальну орієнтацію, яку можна змінити за допомогою властивості

```
property Kind: TScrollBarKind;
```

Тип TScrollBarKind є переліченим типом

```
type
  TScrollBarKind = (sbHorizontal, sbVertical);
```

що визначається значеннями sbHorizontal (горизонтальна орієнтація) і sbVertical (вертикальна орієнтація).

Компонент реагує на події OnChange та OnScroll:

```
property OnChange: TNotifyEvent;
property OnScroll: TScrollEvent;
```

Тип TScrollEvent визначений так:

```
type
  TScrollEvent=procedure(Sender: TObject; ScrollCode:
      TScrollCode; var ScrollPos: Integer) of object;
```

Опрацьовувачі згаданих вище подій визначають поведінку змінюваної за допомогою даного компонента величини. Подія OnScroll відбувається завжди, коли користувач змінює положення бігунка за допомогою миші або клавіатури (<стрілка>, <Ctrl+стрілка>). Негайно за подією OnScroll завжди генерується подія OnChange. При зміні властивості Position програмним шляхом подія OnScroll не виникає.

Приклад, що наведений нижче, ілюструє застосування компонента TScrollBar для роздільного керування складовими кольору з використанням API-функції RGB:

```
function RGB(r, g, b: Byte): COLORREF;
```

Параметри цієї функції задають відповідно інтенсивність червоної (r), зеленої (g) і синьої (b) складових кольору, а значенням, що повертається (фактично воно має тип LongWord), є номер результуючого кольору в палітрі.

```
//Приклад 19.1
//За допомогою трьох смуг прокручування організувати роздільне
//керування складовими кольору, здійснюючи зафарбовування
//панелі кольором, отриманим зміщуванням трьох складових.
```

Розмістимо на формі три смуги прокручування ScrollBar1, ScrollBar2 та ScrollBar3 для керування інтенсивністю червоної, зеленої й синьої складових кольору відповідно, а також панель Panel1. В опрацьовувачі події OnCreate форми установимо властивості `Position` смуг прокручування в значення, що відповідають максимальним значенням інтенсивності, а також визначимо діапазони зміни значень інтенсивності:

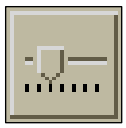
```
procedure TForm1.FormCreate(Sender: TObject);
begin
   with ScrollBar1 do  //Чистий червоний колір, діапазон від 0
     SetParams(clRed, 0, 255); //до 255 (максимально можливий)
   with ScrollBar2 do          //Те ж саме для зеленого кольору
    SetParams(clGreen, 0, 255);
   with ScrollBar3 do           //Те ж саме для синього кольору
     SetParams(clBlue, 0, 255);
end;
```

Для всіх трьох смуг прокручування визначимо один і той самий опрацьовувач події OnScroll:

```
procedure TForm1.ScrollBarScroll(Sender: TObject;
            ScrollCode: TScrollCode; var ScrollPos: Integer);
begin
  Panel1.Color := TColor(RGB(ScrollBar1.Position,
      ScrollBar2.Position, ScrollBar3.Position));
end;
```

Оскільки всі три смуги прокручування мають один і той самий опрацьовувач події OnScroll, він буде викликатися при зсуві бігунка на кожній зі смуг прокручування і змінюватиме кольору панелі.

19.2. Повзунок – компонент TTrackBar



Компонент `TTrackBar` створює шкалу з рисками та повзунком – регулятором поточного положення – та призначений для плавної зміни цілочислової величини з відображенням процесу зміни на шкалі. Як і `TScrollBar`, він є спадкоємцем компонента `TWinControl`. Піктограма компонента розміщена на вкладці Win32 палітри компонентів, а сам він визначений у модулі `ComCtrls`.

Компонент має властивості `Position`, `Min` і `Max`, ідентичні аналогічним властивостям компонента `TScrollBar`.

Керують положенням повзунка дві властивості:

```
property LineSize: Integer;
property PageSize: Integer;
```

Властивість `LineSize` визначає зсув повзунка при натисканні клавіш зі стрілками або мінімальний його зсув при перетаскуванні мишею, а властивість `PageSize` визначає зсув повзунка при кліку мишею праворуч або ліворуч від повзунка або при натисканні клавіш <Page Up> і <Page Down>.

За оформлення компонента відповідають такі його властивості:

- **property** `Frequency: Integer` – частота нанесення рисок на шкалі (одна риска відповідає одній одиниці);
- **property** `Orientation: TTrackBarOrientation` – визначає орієнтацію компонента;
- **property** `SliderVisible: Boolean` – визначає видимість повзунка (при значенні `True` повзунок є видимим);
- **property** `ThumbLength: Integer` – розмір повзунка;
- **property** `TickMarks: TTickMark` – розташування рисок щодо компонента;
- **property** `TickStyle: TTickStyle` – стиль нанесення рисок.

Властивість `Orientation` має тип

```
type
  TTrackBarOrientation = (trHorizontal, trVertical);
```

де значення `trHorizontal` установлює горизонтальну орієнтацію компонента, а `trVertical` – вертикальну.

Можливі значення типу

```
type
  TTickMark = (tmBottomRight, tmTopLeft, tmBoth);
```

мають такий зміст:

- ✓ `tmBottomRight` – риски розміщаються внизу або праворуч;
- ✓ `tmTopLeft` – риски розміщаються вгорі або ліворуч;
- ✓ `tmBoth` – риски розміщаються по обидва боки.

Перелічений тип `TickStyle`

```
type
  TTickStyle = (tsNone, tsAuto, tsManual);
```

визначається за допомогою таких значень:

- ✓ `tsNone` – риски відсутні;

- ✓ `tsAuto` (значення за умовчанням) – риски розставляються автоматично відповідно до значень `Frequency`;
- ✓ `tsManual` – показуються тільки перша й остання риски, а інші наносяться програмно з використанням методу `SetTick`.

Метод `SetTick` має один параметр `Value`, що задає значення, для якого повинна бути нанесена риска:

```
procedure SetTick(Value: Integer);
```

Частина смуги, зображуваної в розглянутому компоненті, може бути виділена, для чого позиції початку та кінця виділення записуються відповідно у властивості `SelStart` і `SelEnd`:

```
property SelStart: Integer;
property SelEnd: Integer;
```

Для даного компонента визначена специфічна подія

```
property OnChange: TNotifyEvent;
```

Опрацьовувач події `OnChange` має бути обов'язково визначений, якщо необхідно, щоб компонент `TTrackBar` реагував на кожну зміну положення повзунка.

З урахуванням того, що компоненти `TTrackBar` і `TScrollBar` дуже схожі, приклад 19.1 дуже просто вирішується із застосуванням повзунка замість смуги прокручування. Для цього в наведеному програмному коді досить зробити найпростішу заміну імен.

Як це зазначалося вище, компоненти `TTrackBar` і `TScrollBar` звичайно зв'язують із якою-небудь цілочисловою величиною. Таке зв'язування не є обов'язковим, оскільки ці компоненти можуть бути використані і як компоненти для виклику команд. Приміром, цілком нормальною з погляду можливості реалізації є наступна конструкція, заснована на застосуванні об'єкта `TrackBar1` класу `TTrackBar`:

```
case TrackBar1.Position of
  1: команда1;
  2: команда2;
  3: команда3;
end;
```

Наприклад, припустимим є такий оператор:

```
case TrackBar1.Position of
  1: Form1.Color := clBlue;
  2: Form1.Color := clYellow;
  5: Form1.Color := clGreen;
end;
```

19.3. Лічильник – компонент TUpDown

Компонент TUpDown створює дві спарені кнопки з направленими вгору й униз стрілками, призначені для покрокового регулювання цілочислової величини. Він визначений у модулі ComCtrls і є віконним компонентом: TWinControl – TCustomUpDown – TUpDown. У палітрі компонентів його піктограма розміщена на вкладці Win32.

Клік над однією з кнопок цього компонента зумовлює зміну значення його властивості

```
property Position: SmallInt;
```

Мінімальне й максимальне можливі значення цієї властивості, а також крок його зміни за допомогою кнопок визначаються відповідно властивостями:

```
property Min: SmallInt;
property Max: SmallInt;
property Increment: Integer;
```

Властивість

```
property Wrap: Boolean;
```

забороняє (True) або дозволяє (False) вихід значення Position за встановлений діапазон (за умовчанням True).

Звичайно компонент TUpDown використовують разом із супровідним компонентом, який називають ***компаньйоном*** і під яким розуміють компонент, що служить для відображення регульованої величини. Найчастіше як компаньйон уживається який-небудь з редакторів, у якому в цьому випадку відображається рядковий еквівалент поточного значення властивості Position.

Для забезпечення зв'язку компонента TUpDown з компаньйоном необхідно занести посилання на нього у властивість

```
property Associate: TWinControl;
```

При цьому сам компонент автоматично переміщається, розташовуючись ліворуч або праворуч від компаньйона залежно від значення властивості

```
property AlignButton: TUDAlignButton;
```

Можливі значення властивості AlignButton визначає тип

```
type
  TUDAlignButton = (udLeft, udRight);
```

де значення udLeft та udRight відповідають розташуванню компонента TUpDown ліворуч та праворуч від компаньйона.

Властивість

```
property Orientation: TUDOrientation;
```

що має тип

```
type
  TUDOrientation = (udHorizontal, udVertical);
```

визначає орієнтацію компонента: udHorizontal – горизонтальна, udVertical – вертикальна.

Властивість

```
property ArrowKeys: Boolean;
```

дозволяє (при значенні True) або забороняє (False) використання клавіш «стрілка вгору» та «стрілка вниз» для керування значенням властивості Position замість клацання кнопкою миші.

Компонент TUpDown може не мати компаньйона, зв'язаного з ним властивістю Associate. Якщо властивість Associate не визначена, то для того, щоб забезпечити належну реакцію на зміну властивості Position, застосовують подію OnClick, що виникає при клацанні мишкою над кнопками лічильника:

```
type
  TUDBtnType = (btNext, btPrev);
  TUDClickEvent = procedure (Sender: TObject;
                             Button: TUDBtnType) of object;
property OnClick: TUDClickEvent;
```

Параметр Button опрацьовувача цієї події визначає, яка кнопка натискалася: btNext – угору або вправо, btPrev – униз або вліво.

Ще одна подія дозволяє забезпечити реакцію на будь-яку зміну регульованої величини:

```
type
  TUDChangingEvent = procedure (Sender: TObject;
                     var AllowChange: Boolean) of object;
property OnChanging: TUDChanginEvent;
```

За допомогою параметра AllowChange опрацьовувач події OnChanging забезпечує передачу повідомлення про можливості зміни регульованої величини.

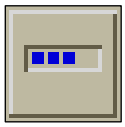
19.4. Редагований рядок з лічильником – компонент TSpinEdit

Компонент `TSpinEdit` являє собою комбінацію однорядкового текстового редактора та лічильника `TSpinButton`. Він визначений у модулі `Spin` і є віконним компонентом – спадкоємцем `TCustomEdit`. Якщо не враховувати наявність спарених кнопок, то можна вважати, що даний компонент відрізняється від компонента `TEdit` тільки тим, що він служить для введення і регулювання цілочислової величини. У палітрі компонентів його піктограма знаходиться на вкладці Samples. Відомості про цей компонент у довідковій системі Delphi відсутні.

Компонент має такі специфічні властивості:

- **property** `EditorEnabled: Boolean` – при значенні `True` дозволяє, а при значенні `False` забороняє ручну зміну вмісту текстового поля;
- **property** `Increment: LongInt` – крок зміни регульованої величини при клацанні мишкою над однією з кнопок;
- **property** `MaxValue: LongInt` – максимальне значення регульованої числової величини;
- **property** `MinValue: LongInt` – мінімальне значення регульованої числової величини;
- **property** `Value: LongInt` – містить поточне значення регульованої числової величини.

19.5. Індикатор процесу – компонент TProgressBar



Компонент `TProgressBar` створює прямокутник, у якому наочно відображається процес виконання деякої досить тривалої процедури. По ходу виконання процедури індикатор поступово забарвлюється зліва направо (за умовчанням) або знизу вгору (при вертикальній орієнтації). Компонент визначений у модулі `ComCtrls` і є дочірнім компонентом класу `TWinControl`. У палітрі компонентів його піктограма знаходиться на вкладці Win32.

Компонент багато в чому схожий на розглянуті компоненти `TScrollBar` і `TTrackBar`, однак він може використовуватися тільки для відображення числової величини без керування нею (у нього у зв'язку з цим відсутні й регулятори зміни значення).

Відображуване значення міститься у властивості `Position`:

```
property Position: Integer;
```

Наступні властивості встановлюють інтервал значень індикатора (Min і Max) та крок (Step) нарощування властивості Position:

```
property Min: Integer;
property Max: Integer;
property Step: Integer;
```

У компоненті TProgressBar властивості Min і Max відображують відсотки його заповнення (Min – 0 % заповнення, Max – 100 % заповнення), а властивість Step – відсоток від закінченого процесу.

Безпосередня зміна властивості Position виконується одним з двох методів:

```
procedure StepIt;
procedure StepBy(Delta: Integer);
```

Метод StepIt змінює властивості Position із застосуванням властивості Step, а метод StepBy використовує для цих цілей значення свого параметра Delta.

Орієнтацією компонента керує властивість

```
type
  TProgressBarOrientation=(pbHorizontal,pbVertical);
property Orientation: TProgressBarOrientation;
```

з можливими значеннями pbHorizontal (горизонтальна орієнтація) і pbVertical (вертикальна орієнтація).

Смуга, що призначена для відображення відсотка заповнювання, може заповнюватися суцільно або сегментовано.

Властивість

```
property Smooth: Boolean;
```

керує способом заповнення смуги індикатора: True (за умовчанням) – сегментована смуга, False – суцільна.

Прикладом використання компонента TProgressBar є типова операція форматування дискети.

Скористаємося (у чисто ілюстративних цілях) для розв'язання сформульованої нижче задачі компонентами TTrackBar, TUpDown, TSpinEdit і TProgressBar (приклад застосування компонента TScrollBar наведено у підрозд. 19.1, цей компонент, природно, може бути вжито і тут).

```
//Приклад 19.2
//Створити секундомір для відліку не більше 60 с. Передбачити
//можливість установки максимального інтервалу часу, на який
//розрахований секундомір, кожним з таких регуляторів:
```

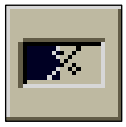
```
//TTrackBar, TUpDown, TSpinEdit. Після закінчення установле-
//ного часу забезпечити видачу звукового сигналу. Передбачити
//відображення у компонентах TTrackBar та TProgressBar часу,
//що минув з моменту запуску секундоміра.
```

Розмістимо на формі кнопку Button1, дві панелі групування GroupBox1 та GroupBox2 і два таймери: Timer1 – для відліку загального часу, Timer2 – для керування процесом відображення ходу часу. У панелі GroupBox1 помістимо компоненти, які будуть забезпечувати установку часу до видачі звукового сигналу, а саме: повзунок TrackBar1, лічильник UpDown1, однорядковий редактор Edit1, а також редактор з лічильником SpinEdit1. Усередині панелі GroupBox2 розташуємо компоненти, які будуть відображати хід часу – повзунок TrackBar2 та індикатор процесу TrackBar2. Запишемо також у властивості Max і Position компонентів TrackBar1 та UpDown1, у властивість Text компонента Edit1, а також у властивості MaxValue і Value компонента SpinEdit1 значення 60 (60 с – відповідно до умови). Крім того, визначимо властивість Association компонента UpDown1, зазначивши як компаньйон компонент Edit1, і сховаємо панель групування GroupBox2. Тоді для розв'язання задачі можуть бути використані такі опрацьовувачі подій:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin   //Кожна зміна у вікні редактора відображається перемі-
  TrackBar1.Position := StrToInt(Edit1.Text);//щенням повзунка
end;                                        //компонента TrackBar1
procedure TForm1.SpinEdit1Change(Sender: TObject);
begin     //Кожна зміна, зумовлена введенням числа у вікні або
  if SpinEdit1.Text <> '' then  //кліками мишкою над кнопками,
    TrackBar1.Position := SpinEdit1.Value;    //також відобра-
end;     // жається переміщенням повзунка компонента TrackBar1
procedure TForm1.TrackBar1Change(Sender: TObject);
begin         //Зворотне відображення зміни положення повзунка
  Edit1.Text := IntToStr(TrackBar1.Position);
  SpinEdit1.Value := TrackBar1.Position;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin   //При кліку над кнопкою панель настановних компонентів
  GroupBox1.Visible := False;   //ховається, а панель відобра-
  GroupBox2.Visible := True;//ження ходу часу робиться видимою
  TrackBar2.Max := TrackBar1.Position;        //Час до сигналу
  ProgressBar1.Max := TrackBar1.Position;
                                             //Установка Timer1
  Timer1.Interval := TrackBar1.Position * 1000;
  Button1.Enabled := False;
                                              //Запуск таймерів
  Timer1.Enabled := True;
  Timer2.Enabled := True;
end;
```

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin                          //При закінченні встановленого часу
  MessageBeep(MB_OK);                  //видається звуковий сигнал
  Timer1.Enabled := False;    //і вимикається секундомір Timer1
end;
procedure TForm1.Timer2Timer(Sender: TObject);
begin     //При спрацьовування таймера хід часу відображається
  ProgressBar1.Position := ProgressBar1.Position + 1; //у двох
  TrackBar2.Position := TrackBar2.Position + 1;  //компонентах
  if not Timer1.Enabled then Timer2.Enabled := False;
end;
```

19.6. Індикатор величини – компонент TGauge



Компонент визначений у модулі `Gauges` як дочірній клас `TGraphicControl`. Його піктограма в палітрі компонентів розміщена на вкладці Samples. Про цей компонент, як і про інші компоненти вкладки Samples палітри компонентів, відсутні відомості в довідковій системі Delphi.

Даний компонент є розвитком компонента `TProgressBar` у плані методики відображення інформації.

Числова величина, що змінюється, і діапазон її зміни містяться в таких властивостях цього компонента:

```
property Progress: LongInt;
property MinValue: LongInt;
property MaxValue: LongInt;
```

Поточне значення властивості `Progress` у відсотках від його діапазону зміни міститься у властивості `PercentDone`, що є властивістю тільки для читання:

```
property PercentDone: LongInt;
```

Індикатор `TGauge` може мати одну з п’яти різних форм, що визначає властивість `Kind`:

```
type
  TGaugeKind = (gkText, gkHorizontalBar,
                gkVerticalBar, gkPie, gkNeedle);
property Kind: TGaugeKind;
```

Можливими значеннями властивості `Kind` є такі:

- ✓ `gkText` – виводиться тільки числове значення;
- ✓ `gkHorizontalBar` – горизонтальний прямокутник із заповненням;

- ✓ `gkVerticalBar` – вертикальний прямокутник із заповненням;
- ✓ `gkPie` – секторна діаграма;
- ✓ `gkNeedle` – половина еліпса з променем, що з'єднує його центр із точкою на дузі.

Змінювати значення властивості `Progress` на цілочислову величину `Value` можна за допомогою звертання до методу

```
procedure AddProgress(Value: LongInt);
```

Усередині індикатора `TGauge` додатково може відображатися саме числове значення. Для цього повинне бути записане значення `True` у властивість

```
property ShowText: Boolean;
```

Використання даного компонента подібне з використанню інших компонентів регулювання й відображення значень величин.

Запитання для контролю і самоконтролю

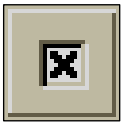
1. Опишіть призначення й основні властивості та методи наступних компонентів: *а*) смуги прокручування; *б*) повзунка; *в*) лічильника.
2. Що таке компаньйон компонента `TUpDown` і як він використовується?
3. Яку подію застосовують для керування значенням числової величини за відсутності компаньйона в компонента `TUpDown`?
4. У чому особливості подій `OnScroll` та `OnChange` компонента `TScrollBar`?
5. Що таке індикатор процесу?
6. Які особливості індикатора величини?

Завдання для практичного відпрацьовування матеріалу

1. За допомогою лічильника `TUpDown` забезпечити можливість зміни допустимої кількості цифр у записі цілого числа, що вводиться (виводиться) за допомогою редактора MaskEdit. Забезпечити автоматичну зміну розміру редакторського вікна залежно від кількості цифр, що вводяться. Виконайте це ж завдання з використанням повзунка та смуги прокручування.
2. Створити додаток для копіювання файлу з відображенням процесу копіювання в компоненті ProgressBar (як варіант розв'язання задачі – копіювання нетипізованого файлу).

20. КОМПОНЕНТИ ВИБОРУ ТА УСТАНОВКИ

20.1. Прапорець – компонент TCheckBox



Цей компонент створює елемент керування з двома або трьома станами (*ні/так* або *ні/так/сумніваюся*), який служить для установки й вибору стану з метою вироблення тієї чи іншої команди керування. Крім того, компонент містить текст, що описує його функціональне призначення. Компонент `TCheckBox` є віконним компонентом (`TWinControl – TButtonControl – TCustomCheckBox – TCheckBox`), визначеним у модулі `StdCtrls`. У палітрі компонентів його піктограма розміщена на вкладці Standard.

Пояснювальний текст міститься у властивості

```
property Caption: TCaption;
```

Цей текст може розташовуватися ліворуч (`taLeftJustify`) або праворуч (`taRightJustify`) від прапорця, що визначається властивістю `Alignment`:

```
type
  TLeftRight = taLeftJustify..taRightJustify;
property Alignment: TLeftRight;
```

За умовчанням компонент `TCheckBox` перевіряє й установлює тільки один з двох можливих станів (*ні/так*). Для забезпечення можливості перевірки й установки третього стану (затіненого «сірого» стану *сумніваюся*) необхідно присвоїти значення `True` властивості

```
property AllowGrayed: Boolean;
```

Стан компонента визначається значенням властивості

```
type
  TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);
property State: TCheckBoxState;
```

Можливими значеннями властивості `State` є:

- ✓ `cbUnchecked` – прапорець не установлений;
- ✓ `cbChecked` – прапорець установлений;
- ✓ `cbGrayed` – прапорець перебуває в «сірому» стані *сумніваюся*.

Для перевірки установки прапорця досить часто використовують властивість

```
property Checked: Boolean;
```

Слід зазначити, що значенню `True` властивості `Checked` відповідають два можливих значення властивості `State` – `cbChecked` і `cbGrayed`. Тому при `AllowGrayed=True` для перевірки стану корисніше може виявитися звертання до властивості `State`. Установка і зняття прапорця користувачем при виконанні програми здійснюється кліком мишкою над компонентом. У випадку установки стану `cbGrayed` прапорець забарвлюється у сірий колір.

Вікно може містити одночасно кілька прапорців. Стан кожного з них не впливає на стан інших прапорців, тобто прапорці є незалежними.

Для перевірки стану компонента використовують умовний оператор або оператор вибору. Наприклад, можливі такі конструкції:

1) `if CheckBox.Checked then оператор1`
   `else оператор2;`

2) `if CheckBox.State = cbUnchecked then оператор;`

3) `case CheckBox.State of`
   `cbChecked : оператор1;`
   `cbUnchecked : оператор2;`
   `cbGrayed : оператор3;`
   `end;`

```
//Приклад 20.1
//Скаче Іван-Царевич по полю й бачить камінь. А на камені на-
//пис: Ліворуч підеш – сам загинеш, але коня збережеш. Право-
//руч підеш – коня втратиш, проте сам живий будеш. Як бути?
```

Розмістимо на формі кнопку Button1, компонент Memo1 і прапорець CheckBox1 з написом V – ліворуч, порожньо – праворуч, затінений – дзвінок другові. Присвоїмо властивості AllowGrayed компонента CheckBox1 значення True. Опишемо також у секції **`private`** опису форми три методи:

```
procedure CommandChecked;
procedure CommandUnchecked;
procedure CommandGrayed;
```

Умовно визначимо ці методи в такий спосіб:

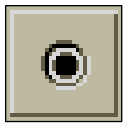
```
procedure TForm1.CommandChecked;
begin
  Memo1.Lines.Add('Що ж. Сам напросився!');
end;

procedure TForm1.CommandUnchecked;
begin
  Memo1.Lines.Add('Друзями не розкидаються!');
end;

procedure TForm1.CommandGrayed;
begin
  Memo1.Lines.Add('Залиш коня тут, а сам іди пішки!');
end;
```

Для кнопки Button1 визначимо такий опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  case CheckBox1.State of
    cbChecked : CommandChecked;
    cbUnchecked : CommandUnchecked;
    cbGrayed : CommandGrayed;
  end;
end;
```

У цьому випадку командами керування є команди, що забезпечують виведення того або іншого повідомлення залежно від варіанта вибору.

20.2. Перемикач – компонент TRadioButton



Перемикач – це елемент керування з двома станами, який має описовий текст, що специфікує його призначення. Піктограма компонента `TRadioButton` розміщена на вкладці Standard палітри компонентів, а сам він визначений у модулі `StdCtrls` і є дочірнім компонентом класу `TButtonControl`, що, у свою чергу, є дочірнім компонентом `TWinControl`.

Цей компонент має деяку подібність із прапорцем. Однак, на відміну від `TCheckBox`, з декількох перемикачів, розташованих в одному контейнері, може бути встановлений тільки один.

Стан компонента визначає його властивість

```
property Checked: Boolean;
```

Якщо в одного перемикача ця властивість має значення `True`, то у всіх інших компонентів `TRadioButton`, розташованих у тому ж контей-

нері, властивість Checked автоматично одержує значення False. Установка перемикача користувачем виконується кліком над ним мишкою. При цьому після того, як один з перемикачів виявиться встановленим, користувач втрачає можливості за допомогою мишки встановити значення False для властивості Checked одночасно всім перемикачам, що входять в одну групу (це зробити можна програмно, наприклад, за допомогою оператора присвоювання).

У класі TRadioButton визначена ще одна специфічна властивість – Alignment, що є аналогічною однойменній властивості компонента TCheckBox. Специфічні події для компонента TRadioButton не визначені.

Зазвичай перемикачі розміщають усередині попередньо встановленого на формі контейнера, причому установка в контейнері тільки одного перемикача практично безглузда. Спеціальним контейнером для розміщення компонентів TRadioButton є компонент TRadioGroup.

```
//Приклад 20.2
//Розмістити на формі перемикачі з іменами Aqua, Cream,
//MoneyGreen та BtnFace для установки кольору форми:
//відповідно clAqua, clCream, clMoneyGreen і clBtnFace.
```

Розмістимо на формі названі перемикачі та кнопку Button1, визначивши для неї наступний опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  cl: Integer;
begin
  if Aqua.Checked then cl := clAqua
  else
    if Cream.Checked then cl := clCream
    else
      if MoneyGreen.Checked then cl := clMoneyGreen
      else cl := clBtnFace;
  Form1.Color := cl;
end;
```

20.3. Група перемикачів – компонент TRadioGroup

Компонент TRadioGroup створює контейнер, що має вигляд прямокутної рамки. Цей контейнер візуально та логічно поєднує групу компонентів TRadioButton. Компонент є віконним: TWinControl – TCustomControl – TCustomGroupBox – TCustomRadioGroup – TRadioGroup. Його піктограма знаходиться на

вкладці Standard палітри компонентів, а сам він визначений у модулі ExtCtrls.

Усі перемикачі, що входять в одну групу, поміщаються в спеціальний список, визначений властивістю

```
property Items: TStrings;
```

Загальну кількість перемикачів у групі визначає властивість Count списку Items (тобто Items.Count).

Номер перемикача, що в групі є установленим, міститься у властивості

```
property ItemIndex: Integer;
```

Щоб зняти установку з усіх перемикачів групи, потрібно присвоїти її властивості ItemIndex значення -1 (саме це значення задається за умовчанням, визначаючи групу невстановлених перемикачів). Для установки ж якого-небудь перемикача його номер (відлічений від 0) записується у властивість ItemIndex:

```
RadioGroup.ItemIndex := 1;     //Установка перемикача 1
```

За допомогою звертання до властивості ItemIndex можна також проаналізувати вибір користувача. Так, якщо група RadioGroup містить три перемикачі, то оператор типу наведеного нижче дозволяє виконати різні дії при установці кожного з перемикачів, а також за відсутності установки:

```
case RadioGroup.ItemIndex of
  0 : оператор1;        //Установлений перший перемикач
  1 : оператор2;        //Установлений другий перемикач
  2 : оператор3;        //Установлений третій перемикач
  else оператор4;       //Усі перемикачі не установлені
end;
```

Перемикачі можуть бути розташовані в декількох стовпцях. Кількість стовпців визначає властивість

```
property Columns: Integer;
```

Для включення перемикачів у групу необхідно клацнути мишкою в Інспекторі Об'єктів над кнопкою в рядку з ім'ям властивості Items, у результаті чого відкривається Редактор Списку Рядків (String List Editor). Занесення тексту в рядок приводить до створення перемикача, біля якого розташується пояснювальний напис, що визначається введеним текстом. Порожньому рядку відповідає перемикач без напису. Видалення рядка приводить до видалення перемикача.

Як і у випадку TGroupBox, у розриві рамки компонента TRadioGroup у верхній її частині можна вивести текст, який пояснює призначення групи. Для цього використовується властивість Caption.

```
//Приклад 20.3
//Організувати обчислення визначеного інтеграла деякої
//функції методом прямокутників або методом трапецій (див.
//завдання 3 до розд. 9).
```

Розмістимо на формі групу RadioGroup1 з двох перемикачів, занісши в їх властивість Items тексти Метод прямокутників і Метод трапецій. Для подачі команди на обчислення інтеграла скористаємося звичайною кнопкою Button1, а для виведення результатів – компонентом Memo1. Опускаючи питання, пов'язані з безпосередньою реалізацією двох методів інтегрування, введенням даних і визначенням інтегрувальної функції, підпрограми для розв'язання даного завдання схематично можуть бути подані в такий спосіб:

```
function MethodOfTrapezes: Real; //В обох функцій повинна бути
begin                                //специфікація параметрів
     //Тут реалізується метод прямокутників
end;

function MethodOfRectangulars: Real;
begin
     //Тут реалізується метод трапецій
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Mess: string;
     //Потрібно ще описати всі змінні
begin
     //Тут потрібно забезпечити введення початкових даних
  case RadioGroup1.ItemIndex of
    0 : Mess := FloatToStr(MethodOfRectangulars);
    1 : Mess := FloatToStr(MethodOfTrapezes);
    else Mess := 'Не вибрано метод інтегрування';
  end;
  Memo1.Lines.Add(Mess);
end;
```

У даному прикладі при виборі одного з перемикачів групи виконується звертання до відповідної функції інтегрування. Якщо при цьому занести імена функцій у масив вказівників на функцію (див. підрозд. 8.7 і приклад 8.14), то відкривається можливість звертання до кожної із цих функцій за поточним значенням властивості ItemIndex компонента

RadioGroup1. Початкові дані для програми можуть бути введені як у процедурі TForm1.Button1Click, так (що більш розумно) і поза нею. Повністю виконати запропоноване завдання пропонується виконати самостійно.

20.4. Простий список – компонент TListBox

Компонент TListBox створює прямокутну область, що є контейнером для зберігання списку рядків, один або декілька з яких користувач може вибрати під час виконання програми з метою обробки їх вмісту або подачі тієї чи іншої команди. Компонент визначений у модулі StdCtrls і є віконним компонентом – спадкоємцем другого рівня компонента TWinControl: TWinControl – TCustomListBox – TListBox. Піктограма компонента знаходиться на вкладці Standard палітри компонентів.

Список TListBox може містити не тільки текстові рядки, але й довільні зображення, що пояснюється тим, що в нього є властивість

```
property Canvas: TCanvas;
```

Як і у компонента TRadioGroup, рядки, що утворюють список, містяться у властивості

```
property Items: TStrings;
```

Номер елемента, який буде обраний під час виконання програми, визначається властивістю

```
property ItemIndex: Integer;
```

Елемент, номер якого занесений у властивість ItemIndex, має фокус введення. Під час виконання програми рядки можна додавати або видаляти, користуючись наявними в класу TStrings методами Add, Insert, Delete і т. д. Значення властивості ItemIndex при цьому не змінюється.

Рядки, що складають список, можуть піддаватися автоматичному сортуванню. Для цього необхідно установити значення True для властивості

```
property Sorted: Boolean;
```

Елементи списку, як і перемикачі в компоненті TRadioGroup, можуть розташовуватися в декількох колонках, кількість яких визначає властивість

```
property Columns: Integer;
```

Висота рядка списку в пікселях визначається значенням властивості

```
property ItemHeight: Integer;
```

У списку одночасно може бути виділено кілька елементів (на етапі виконання це здійснюється кліками мишею над елементами списку при натиснутій клавіші <Ctrl> або <Shift>). Для реалізації можливості одночасного виділення декількох елементів необхідно присвоїти значення `True` властивості

```
property MultiSelect: Boolean;
```

Кількість виділених елементів списку зберігається у властивості тільки для читання

```
property SelCount: Integer;
```

При `MultiSelect = False` властивість `SelCount` завжди містить значення `-1`.

Кожному елементу списку ставиться у відповідність елемент властивості-масиву

```
property Selected[Index: Integer]: Boolean;
```

Якщо елемент списку з номером `Index` обраний (виділений), то відповідний елемент масиву `Selected` має значення `True`; невиділені елементи масиву `Selected` визначені значенням `False`.

Властивість `Selected` дозволяє організувати виконання деякої команди (оператора) для всіх виділених елементів списку:

```
for i := 0 to (ListBox1.Items.Count - 1) do
  if ListBox1.Selected[i] then оператор;
```

Властивість `Style` компонента визначає його модифікацію:

```
type
  TListBoxStyle = (lbStandard, lbOwnerDrawFixed,
  lbOwnerDrawVariable, lbVirtual, lbVirtualOwnerDraw);
property Style: TListBoxStyle;
```

Найчастіше використовуються значення цієї властивості, що визначаються такими константами:

- ✓ `lbStandard` (значення за умовчанням) – створює список, елементи якого мають однакову висоту, зумовлену операційною системою;
- ✓ `lbOwnerDrawFixed` – список аналогічний `lbStandard`-списку, але висота елементів списку визначається значенням властивості

ItemHeight, установлюваним програмою за допомогою опрацьовувача події OnDrawItem;

- ✓ lbOwnerDrawVariable – список аналогічний списку lbOwnerDrawFixed, але його елементи можуть мати різну висоту, що встановлюється опрацьовувачем події OnMeasureItem; кожен з елементів прорисовується опрацьовувачем події OnDrawItem.

```
//Приклад 20.4
//Вивести список імен файлів поточного каталогу, що мають
//розширення .txt. Забезпечити виведення потрібного файлу.
```

Розмістимо на формі компонент ListBox1, надавши його властивості Align значення alLeft, після чого розташуємо на формі компонент Memo1 з двома смугами скролінга і значенням властивості Align, рівним alClient. Тоді запропоновану задачу можна розв'язати, визначивши два опрацьовувачі подій – OnCreate для форми та OnClick для списку:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  SR: TSearchRec;
begin
  if FindFirst('*.txt', faAnyFile, SR) = 0 then begin
    repeat
      if SR.Name[Length(SR.Name) - 3] = '.' then
        ListBox1.Items.Add(Copy(SR.Name,1,Length(SR.Name)-4));
    until FindNext(SR) <> 0;
    FindClose(SR);
  end;
  if ListBox1.Count = 0 then
    Memo1.Lines.Add('У папці немає текстових файлів');
end;

procedure TForm1.ListBox1Click(Sender: TObject);
begin
  if ListBox1.ItemIndex = -1 then Exit;
  Memo1.Clear;
  Memo1.Visible := True;
  Memo1.Lines.LoadFromFile(
        ListBox1.Items.Strings[ListBox1.ItemIndex] + '.txt');
end;
```

Зробимо деякі пояснення до програми. Насамперед, зазначимо, що функція FindFirst буде відшукувати не тільки файли з розширенням .txt, але й файли, розширення яких починається з послідовності символів txt (тобто не обов'язково трисимвольне розширення). Тому перед виведенням імені файлу у вікно компонента перевіряється, чи довжина розширення

дорівнює трьом, після чого виводиться тільки ім'я файлу. Опрацьовувач події OnClick списку виконується при кліках мишкою над списком або при натисканні клавіш керування курсором, забезпечуючи виведення в Memo-поле вмісту файлу, ім'я якого обране в списку (попередньо до імені файлу приєднується розширення .txt).

20.5. Комбінований список – компонент TComboBox

Компонент TComboBox комбінує редагований рядок класу TEdit і список TListBox. Компонент є віконним компонентом (TWinControl – TCustomListControl – TCustomCombo – TCustomComboBox – TComboBox), визначеним у модулі StdCtrls. Його піктограма знаходиться на вкладці Standard палітри компонентів.

Рядки комбінованого списку зберігаються у властивості Items, а для зазначення обраного рядка (елемента, що має фокус введення) використовується властивість ItemIndex. Ці властивості аналогічні однойменним властивостям компонента TListBox.

На екрані завжди відображається редагований рядок, вміст якого визначається його властивістю Text. При виборі рядка, що міститься в списку, він автоматично переписується в цю властивість і відображається в редагованому рядку.

Властивість Style компонента визначає його модифікацію:

```
type
  TComboBoxStyle=(csDropDown,csSimple,csDropDownList,
    csOwnerDrawFixed,csOwnerDrawVariable);
property Style: TComboBoxStyle;
```

Можливими значеннями цієї властивості є такі константи:

- ✓ csDropDown (значення за умовчанням) – створює список, що розкривається за допомогою кнопки, розташованої в правій частині вікна редактора, з можливістю ручного редагування властивості Text (пункти списку мають однакову висоту, що встановлюється операційною системою);
- ✓ csSimple – створює редакторське вікно з розташованим під ним списком, довжина відображуваної частини якого визначається значенням властивості Height (переміщення за списком із забезпеченням можливості доступу до невідображених його пунктів забезпечується за допомогою клавіш зсуву курсору «униз» і «вгору»);

- ✓ `csDropDownList` – список аналогічний `csDropDown`-списку без можливості ручного редагування вмісту вікна редактора;
- ✓ `csOwnerDrawFixed` – список аналогічний `csDropDownList`-списку, але висота пунктів списку визначається значенням властивості `ItemHeight`;
- ✓ `csOwnerDrawVariable` – список аналогічний тому, що створюється для значення `csDropDownList`, але висота його пунктів може бути різною.

Кількість елементів списку, яка не вимагає появи смуги скролінга, визначається властивістю

```
property DropDownCount: Integer;
```

Властивість `DroppedDown` (що є **public**-властивістю) своїм значенням `True` виконує розкриття списку (за умовчанням `False`):

```
property DroppedDown: Boolean;
```

Для компонента `TComboBox` визначена досить велика кількість подій, насамперед, успадкованих від його прабатьків.

Найпростіший випадок використання комбінованого списку наводився в підрозд. 11.2 (приклад 11.2).

20.6. Група прапорців – компонент TCheckListBox

Компонент `TCheckListBox`, як і `TRadioGroup`, створює контейнер, що візуально й логічно об'єднує групу компонентів `TCheckBox` і має вигляд прямокутної рамки. Компонент визначений у модулі `CheckLst` і є віконним компонентом – спадкоємцем другого рівня компонента `TWinControl`: `TWinControl` – `TCustomListBox` – `TCheckListBox`.

Піктограма компонента `TCheckListBox` розміщена на вкладці Additional палітри компонентів.

Даний компонент має властивості `Canvas`, `Columns`, `ItemHeight`, `ItemIndex`, `Items`, `MultiSelect`, `SelCount`, `Selected`, `Sorted`, `Style`, аналогічні однойменним властивостям компонента `TListBox`. Властивості `State` й `AllowGrayed` даного компонента ідентичні однойменним властивостям компонента `TCheckBox` (властивість `AllowGrayed` при цьому стосується відразу всіх прапорців).

Властивість

```
property IntegralHeight: Boolean;
```

даного компонента при її значенні True указує компоненту на необхідність автоматичної зміни висоти таким чином, щоб у ньому повністю розміщалися всі прапорці (за умовчанням воно має значення False).

Визначити, які з прапорців установлені (або програмно установити прапорець з індексом Index чи скасувати його установку), можна за допомогою звертання до властивості

```
property Checked[Index: Integer]: Boolean;
```

Процес створення групи прапорців нічим не відрізняється від процесу створення групи перемикачів.

Наступний приклад ілюструє деякі можливості компонентів TCheckListBox, TListBox й TRadioGroup.

```
//Приклад 20.5
//У текстовому файлі List.txt зберігається список імен
//avi-файлів. Використовуючи групу прапорців, відобразити цей
//список й організувати вибір імен файлів з формуванням
//"гарячої десятки" для наступного перегляду за допомогою
//медіаплеєра. Для вибору фільму застосувати групу перемика-
//чів. Передбачити можливість видалення записів зі списку
//перегляду й можливість його поповнення.
```

Розмістимо на формі групу прапорців (дамо їй ім'я TotalList1), панель з групою перемикачів (ім'я RadioGroup1) і простим списком (ім'я HotList1). Крім того, розташуємо на формі медіаплеєр (компонент TMediaPlayer), а також три кнопки: Button1 (для завантаження списку фільмів), Button2 (для видалення елементів) і Button3 (для запуску медіаплеєра). Створимо також опрацьовувачі події OnClick для згаданих компонентів:

```
procedure TForm1.Button1Click(Sender: TObject);
begin                              //Завантажуємо список фільмів
 if FileExists('List.txt') then                 //Чи є список?
   TotalList1.Items.LoadFromFile('List.txt');
 Button1.Visible := False;
end;

procedure TForm1.TotalList1Click(Sender: TObject);
var
  Ind: Integer;                          //Індекс обраного рядка
  Film: string;                                  //Обраний рядок
begin
  if HotList1.Count = 0 then
    Panel1.Visible := True;
  Ind := TotalList1.ItemIndex;   //Запам'ятовуємо індекс рядка
  Film := TotalList1.Items.Strings[Ind];//Запам'ятовуємо рядок
```

```
  TotalList1.Checked[Ind] := True;    //Установлюємо прапорець
  if HotList1.Items.IndexOf(Film) <> -1   //Такий рядок уже є
    then Exit;
  if HotList1.Items.Count > 9 then begin
            //Список переповнився. Знімаємо установку прапорця
    TotalList1.Checked[TotalList1.ItemIndex] := False;
    Exit;                   //й виходимо з опрацьовувача події
  end;
  RadioGroup1.Height := 15 +
                        (RadioGroup1.Items.Count + 1) * 15;
  HotList1.Height := 5 + (HotList1.Items.Count + 1)
                     * TotalList1.ItemHeight;
  HotList1.Items.Add(Film);           //Додаємо рядок у список
  RadioGroup1.Items.Add(Film);            //Додаємо перемикач
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
   //Спочатку перевіряємо порожність списку й наявність вибору
  if (HotList1.Count = 0) or (HotList1.ItemIndex = -1)
    then Exit;
                //Видаляємо перемикач і зменшуємо висоту групи
  RadioGroup1.Items.Delete(RadioGroup1.ItemIndex);
  RadioGroup1.Height := RadioGroup1.Height - 15;
  RadioGroup1.ItemIndex := -1;         //Тепер вибір відсутній
                    //Знімаємо установку прапорця для фільму,
                                       //вилученого з HotList1
  TotalList1.Checked[TotalList1.Items.IndexOf(
      HotList1.Items.Strings[HotList1.ItemIndex])] := False;
  HotList1.DeleteSelected;         //Видаляємо виділений рядок
                                     //Зменшуємо висоту списку
  HotList1.Height := HotList1.Height - HotList1.ItemHeight;
  //Якщо список порожній, приховуємо панель разом з її вмістом
  if HotList1.Count = 0 then
    Panel1.Visible := False;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    //Спочатку перевіряємо порожність групи й наявність вибору
  if (RadioGroup1.Items.Count=0)or(RadioGroup1.ItemIndex < 0)
    then Exit;
                        //Завантажуємо ім'я файлу в медіаплеєр
  MediaPlayer1.FileName :=
            RadioGroup1.Items.Strings[RadioGroup1.ItemIndex];
  MediaPlayer1.Open;                  //Відкриваємо медіаплеєр
  MediaPlayer1.Play;                   //Починаємо відтворення
end;
```

```
procedure TForm1.HotList1Click(Sender: TObject);
begin          //Тут за вибором у списку встановлюємо перемикач
  RadioGroup1.ItemIndex := HotList1.ItemIndex;
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin                                          //А тут навпаки
   HotList1.ItemIndex := RadioGroup1.ItemIndex;
end;
```

Підкреслимо, що оскільки установка прапорця в компоненті `TCheckListBox` відбувається тільки при кліку мишкою безпосередньо над прапорцем, але не над написом біля нього, у процедурі `TForm1.Button2Click` при виборі рядка в списку здійснюється установка відповідного прапорця за допомогою оператора `TotalList1.Checked[Ind] := True`.

У програмі компоненти `RadioGroup1` і `HotList1` функціонують спільно (обрані елементи в них завжди мають однакові значення індексів). Тому, щоб уникнути збоїв у роботі програми, список `HotList1` не повинен піддаватися автоматичному сортуванню.

У наведеному вище лістингу використовувався віконний компонент `TMediaPlayer`, визначений у модулі `MPlayer`. Піктограма цього компонента, що має вигляд  , розміщена на вкладці System палітри компонентів. У найпростішому випадку звертання до цього компонента не становить жодних труднощів. Для цього достатньо завантажити ім'я AVI-файлу у властивість `FileName` цього компонента, після чого послідовно звернутися до його методів `Open` й `Play`.

20.7. Комбінований список з розширеними можливостями – компонент TComboBoxEx

Компонент `TComboBoxEx` багато в чому є аналогом компонента `TComboBox`, але при цьому має деяке розширення можливостей (`Ex` – скорочення від extended – розширений). Піктограма компонента знаходиться на вкладці Win32 палітри компонентів, а сам він визначений у модулі `StdCtrls` і розташовується на нижньому рівні такого ланцюжка віконних компонентів: `TWinControl` – `TCustomListControl` – `TCustomCombo` – `TCustomComboBoxEx` – `TBiDiComboBoxEx` – `TComboBoxEx`.

У порівнянні з `TComboBox`, елементи списку, створені за допомогою даного компонента, можуть забезпечуватися невеликими зображеннями, а

також регульованим відступом від лівого краю компонента. Більша частина властивостей і методів компонентів `TComboBoxEx` та `TComboBox` збігається.

Спеціально компонент `TComboBoxEx` нами розглядатися не буде.

20.8. Список вибору кольору – компонент TColorBox

Компонент `TColorBox` створює випадний список, призначений для вибору та відображення кольору. Він визначений у модулі `ExtCtrls` і є віконним компонентом (`TWinControl` – `TCustomListControl` – `TCustomCombo` – `TCustomComboBox` – `TCustomColorBox` – `TColorBox`). Піктограма компонента перебуває на вкладці Additional палітри компонентів.

Більшість властивостей, методів і подій даного компонента подібні до однойменних характеристик компонента `TComboBox`. Специфічні ж властивості компонента визначають те, що він використовується саме для вибору кольору, причому склад відображуваних цим компонентом кольорів може змінюватися.

Те, які саме категорії кольору наведені в списку, визначає множина `Style`:

```
type
  TColorBoxStyles = (cbStandardColors,
    cbExtendedColors, cbSystemColors, cbIncludeNone,
    cbIncludeDefault, cbCustomColor, cbPrettyNames);
  TColorBoxStyle = set of TColorBoxStyles;
property Style: TColorBoxStyle;
```

```
property Style: TColorBoxStyle;
```

Значення, що включають у цю множину, позначають таке:

- ✓ `cbStandardColors` – список містить 16 основних RGB-кольорів;
- ✓ `cbExtendedColors` – у список включені додаткові кольори, визначені в модулі `Graphics`;
- ✓ `cbSystemColors` – у список включені системні кольори, установлені в Windows;
- ✓ `cbIncludeNone` – у список включений рядок `clNone` («без кольору»); який саме колір відповідає цьому рядку визначається властивістю `NoneColorColor` (ця опція працює тільки разом з опцією `cbSystemColors`);

- ✓ `cbIncludeDefault` – у список включений рядок clDefault («колір за умовчанням»); який саме колір відповідає цьому рядку визначається версією Windows (ця опція працює тільки разом з опцією `cbSystemColors`);
- ✓ `cbCustomColor` – у список включений рядок Custom..., при виборі якої відкривається стандартне діалогове Windows-вікно для вибору кольору, у якому користувач може визначити нестандартний колір;
- ✓ `cbPrettyNames` – рядки списку позначають кольори, а не їх імена (наприклад, Black, а не clBlack).

Іншими специфічними властивостями компонента `TColorBox` є такі властивості:

- ➢ **`property`** `ColorNames[Index: Integer]:` **`string`** – масив імен кольорів, що містяться в списку (властивість тільки для читання);
- ➢ **`property`** `Colors[Index: Integer]: TColor` – список кольорів (властивість тільки для читання);
- ➢ **`property`** `DefaultColorColor: TColor` – колір, що відповідає рядку `clDefault`;
- ➢ **`property`** `NoneColorColor: TColor` – колір, що відповідає рядку clNone;
- ➢ **`property`** `Selected: TColor` – поточний колір.

20.9. Список зображень – компонент TImageList

Компонент `TImageList` є контейнером для зберігання набору малюнків однакового розміру, які можуть бути растровими малюнками або піктограмами. Він визначений у модулі `Controls` і є невізуальним компонентом: `TControl` – `TCustomImageList` – `TDragImageList` – `TImageList`. Піктограма компонента розміщена на вкладці Win32 палітри компонентів.

Будучи списком, даний компонент забезпечує індексований доступ до зображень, що зберігаються в ньому. Кількість зображень у списку, їхня висота та ширина визначаються відповідно такими властивостями:

```
property Count: Integer;
property Height: Integer;
property Width: Integer;
```

При зміні однієї з властивостей `Height` і `Width` список очищається.

На етапі проектування список може бути заповнений за допомогою спеціального редактора (ImageList Editor), що відкривається подвійним кліком мишкою над зображенням компонента на формі або дереві об'єктів або кліком правою кнопкою мишки над зображенням компонента з наступним вибором у контекстному меню, що при цьому відкриється, команди Редактор ImageList (ImageList Editor).

Малюнок виводиться за допомогою методу `Draw`, причому, оскільки компонент `TImageList` не має власної канви, зображення виводиться на канві іншого компонента:

```
procedure Draw(Canvas: TCanvas; X, Y, Index: Integer;
                  Enabled: Boolean = True);
```

Метод `Draw` малює на канві `Canvas` зображення, що має в списку зображень індекс, який задається параметром `Index`. Координати лівого верхнього кута виведеного зображення визначають параметри `X` та `Y`, а параметр `Enabled` установлює, як повинне виводитися зображення – нормально (`True`) або сірим, як зображуються недоступні компоненти (`False`). Оскільки параметр `Enabled` за умовчанням дорівнює `True`, при виклику методу `Draw` його можна не зазначати.

Для перенесення в графічний об'єкт зображення з індексом `Index` можуть бути використані методи

```
procedure GetBitmap(Index: Integer; Image: TBitmap);
procedure GetIcon(Index: Integer; Image: TIcon);
```

У компонента визначено досить багато методів (`Add`, `AddIcon`, `Clear`, `Delete`, `Insert`, `InsertIcon` й ін.), що дозволяють на етапі виконання програми коректувати список.

Більш докладні відомості про цей компонент можна одержати у вбудованій довідковій системі Delphi.

Запитання для контролю і самоконтролю

1. У чому особливості прапорця?
2. Охарактеризуйте компонент `TRadioButton`.
3. Яке призначення групи перемикачів?
4. Що таке простий список?
5. У чому відмінність комбінованого списку від простого?
6. Яке призначення групи прапорців? Як вона використовується?
7. Опишіть призначення й методику використання списку вибору кольору.

8. Опишіть методику використання списку вибору зображення.

Завдання для практичного відпрацьовування матеріалу

1. Створити додаток, що демонструє роботу компонента `TCheckBox`, а саме, за командою від кнопки виводити повідомлення про установку прапорця, розміщеного на формі.
2. Створити додаток, аналогічний попередньому, який демонструє роботу перемикача `TRadioButton`.
3. Створити аналогічний додаток для компонента `TRadioGroup`.
4. Доповнити завдання 3 вимогою наявності групи перемикачів, що вкладена в інший компонент `TRadioGroup`.
5. Дано натуральне число *R*. Побудувати фігури, зображені на рис. 20.1, *а–г*. Фігури утворені колом радіусом *R* і вісьмома точками, що є вершинами вписаного в це коло правильного багатокутника і з'єднані між собою зазначеним способом. Для вибору фігури, що будується, використовувати компонент `TRadioGroup`.

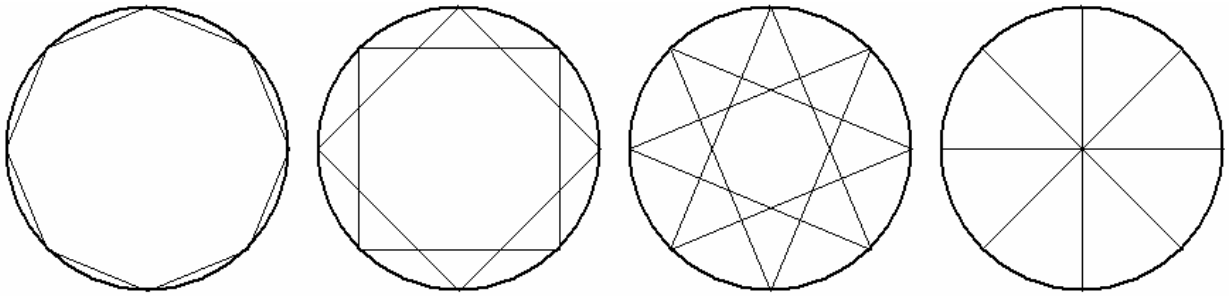


Рис. 20.1. Фігури для побудови

6. Створити клас `TLineSegm`, що реалізує задачу із завдання 1 розд. 14. Від класу `TLineSegm` породити два нащадки – `TSquarePattern` (візерунок з квадратів, див. завдання 6 з розд. 17) і `TTrianglePattern` (аналогічний візерунок з трикутників). Класи-нащадки повинні мати властивість, що визначає кількість рівнів вкладеності (тут рекомендується застосувати проміжний клас в ієрархії спадкування). Для класу `TSquarePattern` визначити клас-нащадок `TCompoundSquarePattern`, що реалізує завдання 8 з розд. 17. Визначити аналогічного нащадка для `TTrianglePattern`. Вибір рисунка виконувати за допомогою груп перемикачів.

21. КОМПОНЕНТИ ДЛЯ ОРГАНІЗАЦІЇ ДІАЛОГІВ

Компоненти для організації діалогів є невізуальними компонентами – спадкоємцями класу `TCommonDialog`, що, у свою чергу, є дочірнім класом класу `TComponent`. Піктограми всіх цих компонентів знаходяться на вкладці Dialogs палітри компонентів, а самі вони визначені в модулі `Dialogs` (за винятком компонентів `TOpenPictureDialog` та `TSavePictureDialog`, визначених у модулі `ExtDlgs`). При виконанні програми здійснюється візуалізація створюваних цими компонентами вікон, але самі компоненти при цьому залишаються невидимими.

21.1. Загальні принципи роботи з компонентами для організації діалогів

Усі ці компоненти можуть бути розміщені на формі, як на етапі проектування, так і при виконанні програми. Діалогове вікно створюється звертанням до методу `Execute` відповідного класу:

```
function Execute: Boolean;
```

Метод `Execute` створює вікно, у якому, крім інших компонентів, містяться дві кнопки – підтвердження вибору (наприклад, Open – Відкрити) і відмови від вибору (Cancel – Скасування). При закритті діалогового вікна кліком над кнопкою підтвердження вибору функція `Execute` повертає значення `True`, у противному випадку повертається значення `False`. Якщо функція `Execute` повернула значення `False`, то це означає, що в діалоговому вікні не проводився вибір, а отже, не відбулася зміна значень властивостей цього компонента. Оскільки за умовчанням ці властивості можуть бути не визначені, їх використання може призвести до появи помилок часу виконання (у випадку клацання мішкою над кнопкою Cancel дії з обробки даних не повинні реалізовуватися – саме для подачі такого сигналу і служить ця кнопка). Таким чином, щоб уникнути можливих

помилок після закриття вікна, потрібно завжди робити перевірку того, яким способом проводилося закриття.

Підсумовуючи усе сказане, можна зробити висновок про те, що відкриття діалогового вікна та обробка даних, отриманих у результаті його функціонування, повинні виконуватися за такою схемою:

```
if екземпляр_діалогового_класу.Execute then
begin                                    //Вибір здійснено
  //Дії з обробки результатів вибору
end;
```

Тут екземпляр_діалогового_класу – екземпляр будь-якого класу – спадкоємця `TCommonDialog` (див. приклади нижче).

Початкові установки діалогових вікон на етапі проектування можна виконати після подвійного кліку мишкою над зображенням компонента на формі чи в інспекторі об’єктів або після кліку над ним правою кнопкою мишки з вибором у контекстному меню команди (Test Dialog) Перевірити Діалог.

21.2. Вікно відкриття файлу – компонент TOpenDialog

Компонент `TOpenDialog` створює та обслуговує стандартне модальне діалогове вікно завантаження файлу. Діалогове вікно, створюване методом `Execute` цього класу, має вигляд, аналогічний тому, що подано на рис. 21.1.

У відкритому діалоговому вікні користувач може вибрати той файл, що його цікавить, користуючись стандартними для Windows методами, у тому числі набрати ім’я файлу вручну у віконці Ім’я об’єкта. Результат вибору у вигляді повного імені файлу записується в **published**-властивість

```
property FileName: TFileName;
```

Тип `TFileName` цієї властивості фактично є типом **string**.

За умовчанням у цю властивість записується порожній рядок, результатом чого може бути те, що при відмові від вибору (закритті вікна кліком над кнопкою Скасування) порожній рядок може надійти на обробку як ім’я відкриваного файлу, що підтверджує необхідність перевірки того, що в діалоговому вікні перед його закриттям вибір був зроблений. При ручному введенні імені файлу користувач може зазначити неіснуючий файл, результатом чого також буде помилка часу виконання, якщо не передбачити дій, спрямованих на усунення наслідків неправильного

введення (обробку відповідного винятку, перевірку існування файлу, що відкривається, перевірку типізованого файлу на наявність атрибута захисту від запису тощо).

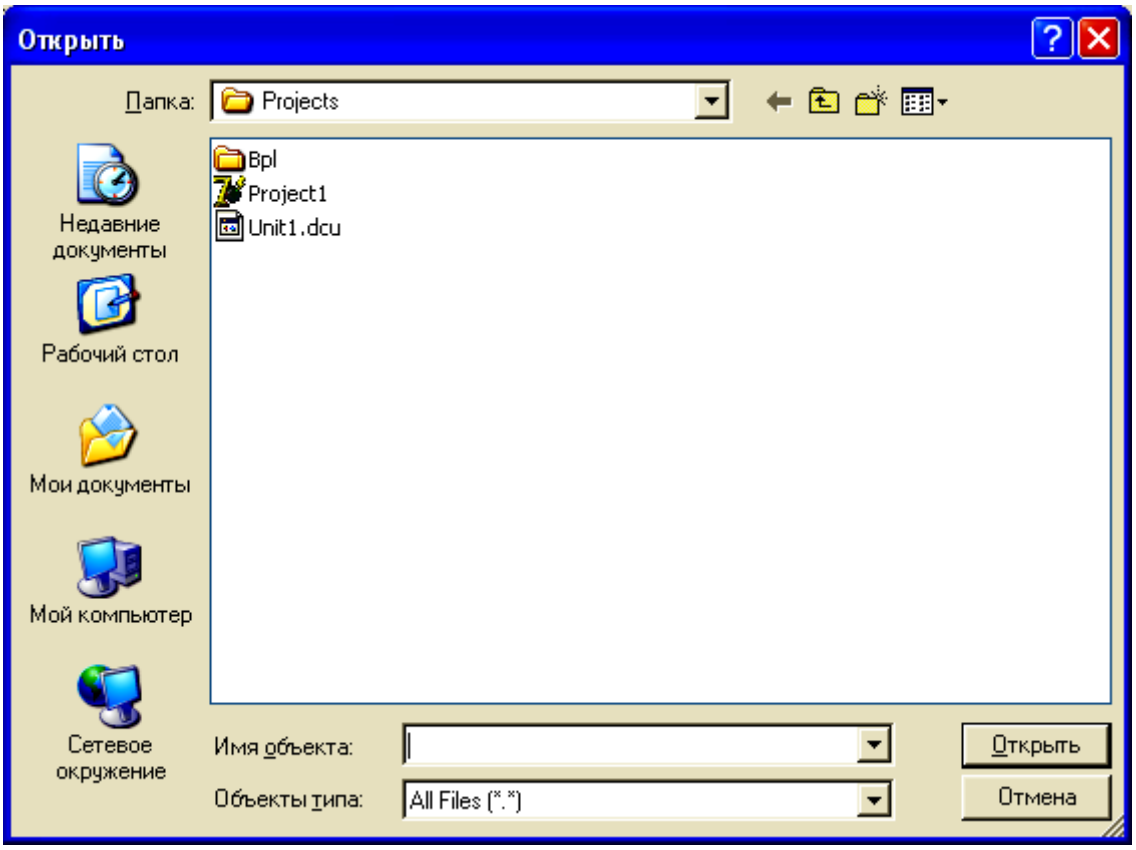


Рис. 21.1. Стандартне вікно відкриття файлу

У властивість `FileName` може бути записане (насамперед, на етапі проектування за допомогою Інспектора Об'єктів) ім'я файлу за умовчанням, що буде використовуватися у випадку відмови від вибору.

У діалоговому вікні можуть відображатися не обов'язково всі файли – можливе задання фільтра, за яким ці файли будуть відбиратися, для чого служить властивість

```
property Filter: string;
```

Цю властивість можна визначати як на етапі проектування, так і програмно.

Щоб установити значення цієї властивості за допомогою Інспектора Об'єктів, необхідно викликати спеціальний редактор фільтра, доступ до якого відкривається кліком над кнопкою ... в рядку властивості `Filter` (див. рис. 21.2). У лівому стовпчику (Filter Name) редактора фільтра дається короткий опис кожного з типів файлів, які можуть відображатися в діалоговому вікні на етапі виконання (наприклад, Всі файли (*.*)), а в правому стовпчику (Filter) міститься безпосередньо фільтр (наприклад, *.*).

Якщо властивість `Filter` не визначена, що відповідає порожньому рядку, то в діалоговому вікні відображаються всі файли.

Фільтр може задаватися як однією з прийнятих у Windows маскою файлів, так і одночасно декількома масками. В останньому випадку маски розділяються символом «крапка з комою» (наприклад, *.pas;*.dpr;*.dfm).

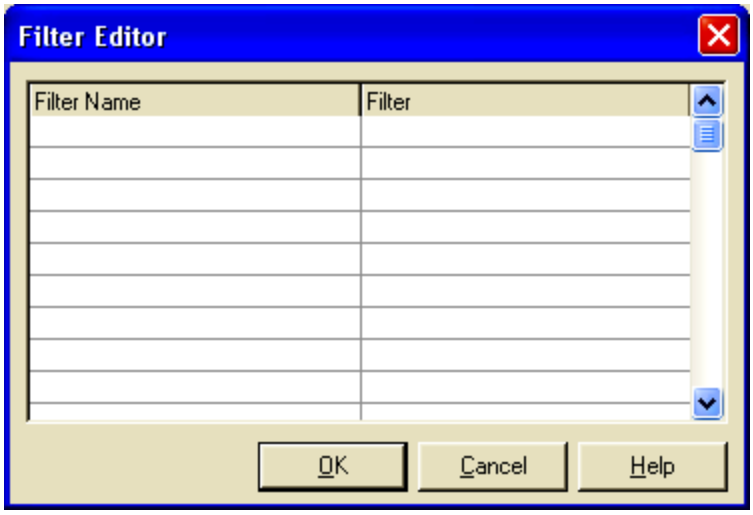


Рис. 21.2. Вікно редактора фільтру

Щоб визначити фільтр у коді програми, необхідно присвоїти непусте значення властивості `Filter`. Окремі фільтри в цьому рядку відокремлюються один від одного вертикальною рисою (`|`). Цей же символ служить для розділення опису файлів, що відфільтровуються, і безпосередньо фільтра. Як і в редакторі фільтра, у випадку використання декількох масок файлів у одному фільтрі вони відокремлюються крапкою з комою. Наприклад, для відбору TXT-файлів, PAS-файлів і файлів Delphi можна скористатися таким оператором:

```
OpenDialog1.Filter :=
       'Текстові файли|*.TXT|Pascal-файли|*.PAS' +
       '|Delphi-файли|*.PAS *.dpr;*.pas;*.dfm';
```

Властивість

```
property DefaultExt: string;
```

дозволяє виконати формування повного імені файлу, якщо при ручному введенні не зазначається розширення. У випадку, коли зазначене при введенні розширення не збігається з тим розширенням, що записане в даній властивості, до набраного імені праворуч формально приєднується точка та вміст властивості `DefaultExt` (природно, якщо користувач набере розширення, то приєднання однаково відбудеться). За умовчанням у цій властивості записано порожній рядок, і приєднання не виконується.

Можливе також задання каталогу, що буде поточним в момент відкриття діалогового вікна, для чого служить властивість

```
property InitialDir: string;
```

Якщо значенням `InitialDir` є порожній рядок або ім'я неіснуючого поточного каталогу, каталогом для діалогового вікна є або поточний каталог Delphi, або папка My Documents (Мої Документи) залежно від інших налаштувань і версії операційної системи.

Властивість

```
property Title: string;
```

визначає текст, що виводиться у вигляді назви вікна. Якщо вона не визначена, то як заголовок виводиться слово Open (Відкрити).

Налаштування вікна відкриття файлу проводиться зміною його властивості `Options`, визначеної в такий спосіб:

```
type
  TOpenOption = (ofReadOnly, ofOverwritePrompt,
  ofHideReadOnly, ofNoChangeDir, ofShowHelp,
  ofNoValidate, ofAllowMultiSelect,
  ofExtensionDifferent, ofPathMustExist,
  ofFileMustExist, ofCreatePrompt, ofShareAware,
  ofNoReadOnlyReturn, ofNoTestFileCreate,
  ofNoNetworkButton, ofNoLongNames, ofOldStyleDialog,
  ofNoDereferenceLinks, ofEnableIncludeNotify,
  ofEnableSizing, ofDontAddToRecent,
  ofForceShowHidden);
  TOpenOptions = set of TOpenOption;
property Options: TOpenOptions;
```

Значенням цієї властивості є множина, сформована з таких констант:

- ✓ `ofReadOnly` – встановлює у вікні прапорець Тільки читання;
- ✓ `ofOverwritePrompt` – забезпечує виведення повідомлення із запитом на підтвердження запису у вже існуючий файл (для вікна відкриття файлу це значення ігнорується);
- ✓ `ofHideReadOnly` – приховує у вікні прапорець Тільки читання;
- ✓ `ofNoChangeDir` – після кліку над кнопкою OK повторно встановлюється як поточний каталог папка, що була поточною до відкриття діалогового вікна;
- ✓ `ofShowHelp` – показує в діалоговому вікні кнопку Help (Довідка);
- ✓ `ofNoValidate` – забороняє перевірку введення неприпустимих символів в іменах файлів;
- ✓ `ofAllowMultiSelect` – дозволяє вибір більше одного файлу;

- ✓ `ofExtensionDifferent` – свідчить про те, що користувач під час діалогу ввів розширення, відмінне від того, яке зазначене в `DefaultExt`;
- ✓ `ofPathMustExist` – вказує на необхідність виведення повідомлення про відсутність каталогу при спробі користувача вибрати неіснуючий каталог;
- ✓ `ofFileMustExist` – вказує на необхідність виведення повідомлення про відсутність файлу при спробі користувача вибрати неіснуючий файл (подається тільки у вікнах відкриття);
- ✓ `ofCreatePrompt` – якщо користувач уводить ім'я відсутнього файлу, приводить до видачі повідомлення з пропозицією створити файл із введеним ім'ям;
- ✓ `ofShareAware` – дозволяє вибір файлів, використовуваних іншими програмами;
- ✓ `ofNoReadOnlyReturn` – приводить до видачі повідомлень про помилку при спробі доступу до файлу, що має атрибут «тільки для читання»;
- ✓ `ofNoTestFileCreate` – забороняє перевірку доступності мережевого або локального диска;
- ✓ `ofNoNetworkButton` – забороняє вставку кнопки створення сітьового диска (якщо не встановлений прапор `ofOldStyleDialog`, ігнорується);
- ✓ `ofNoLongNames` – забороняє використання довгих імен файлів (якщо не встановлений прапор `ofOldStyleDialog`, ігнорується);
- ✓ `ofOldStyleDialog` – створює діалогове вікно в стилі Windows 3.x;
- ✓ `ofNoDereferenceLinks` – забороняє розійменування ярликів (якщо цей прапорець не встановлений, то при виборі в діалоговому вікні ярлика у властивість `FileName` записується ім'я LNK-файлу, а не ім'я файлу, на який вказує ярлик);
- ✓ `ofEnableIncludeNotify` – посилає діалоговому вікну повідомлення, якщо користувач відкриває папку;
- ✓ `ofEnableSizing` – вказує на можливість зміни розмірів діалогового вікна за допомогою миші або клавіатури;
- ✓ `ofDontAddToRecent` – забороняє додавання файлу до списку нещодавно відкриваних файлів;
- ✓ `ofForceShowHidden` – показує в діалоговому вікні приховані файли.

Повне ім'я файлу, що міститься у властивості `FileName`, перед закриттям діалогового вікна записується першим рядком списку, визначеного властивістю

```
property Files: TStrings;
```

Якщо дозволено вибір тільки одного файлу, цей список буде містити єдиний рядок. У випадку ж, коли дозволено множинний вибір (тобто при встановленому прапорці `ofAllowMultiSelect` у властивості `Options`), у список `Files` заносяться повні імена всіх обраних файлів.

При показі діалогового вікна відкриття файлу генерується подія

```
property OnShow: TNotifyEvent;
```

Опрацьовувач цієї події повинен бути написаний, якщо необхідно виконання спеціальних дій перед показом діалогового вікна.

Аналогічно виконання яких-небудь операцій після гасіння діалогового вікна може бути реалізоване опрацьовувачем події

```
property OnClose: TNotifyEvent;
```

Будь-яка зміна вмісту діалогового вікна (зміна імені обираного файлу, зміна або створення папки, зміна фільтра відбору тощо) приводить до виникнення події

```
property OnSelectionChange: TNotifyEvent;
```

Зміна папки, вміст якої відображається в діалоговому вікні, і зміна фільтра відбору приводять відповідно до виникнення подій

```
property OnFolderChange: TNotifyEvent;
```

та

```
property OnTypeChange: TNotifyEvent;
```

Зауважимо, що зразу після появи однієї з подій OnFolderChange та OnTypeChange обов'язково з'являється подія OnSelectionChange.

Якщо діалогове вікно закривається кліком над кнопкою підтвердження вибору Open, то безпосередньо перед закриттям виникає така подія OnCanClose:

```
type
  loseQueryEvent = procedure(Sender: TObject;
                   var CanClose: Boolean) of object;
property OnCanClose: TCloseQueryEvent;
```

Опрацьовувач цієї події визначають у тому випадку, коли необхідно виконати деякі дії безпосередньо перед закриттям вікна (такими діями, наприклад, можуть бути перевірка існування обраного файлу чи папки,

видача запиту на підтвердження відкриття файлу тощо). Якщо опрацьовувач цієї події занесе значення `False` у свій параметр `CanClose`, то вікно не закриється, внаслідок чого користувач змушений буде виконати деякі додаткові дії.

```
//Приклад 21.1
//Користуючись стандартним вікном відкриття файлу
//вивести обраний файл у багаторядковий текстовий
//редактор. Організувати програмну установку фільтра вибору,
//що передбачає вибір txt-файлів, pas-файлів,  dpr-файлів,
//а також одночасний вибір pas-, dpr- і dfm-файлів.
```

Розмістимо на формі вікно відкриття файлу OpenDialog1 і кнопку Button1, визначивши для кнопки такий опрацьовувач події OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDialog1.Filter := 'Текстові файли |*.txt' +     //Фільтр
    '|Pascal-файли |*.pas|Проекти Delphi|*.dpr' +
    '|Delphi |*.dpr;*.pas;*.dfm';
  if (OpenDialog1.Execute)    //Якщо діалог завершився вибором
     and FileExists(OpenDialog1.FileName) then  //і файл існує
  begin
    Memo1.Clear;
    Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
  end;
end;
```

Замість того щоб перевіряти існування відкриваного файлу у процедурі `TForm1.Button1Click`, можна цю ж дію виконувати до закриття діалогового вікна. Для цього створимо для вікна опрацьовувач події CanClose, код якого, наприклад, може мати такий вигляд:

```
procedure TForm1.OpenDialog1CanClose(Sender: TObject;
                                     var CanClose: Boolean);
begin
  if FileExists(OpenDialog1.FileName) then
    CanClose := True
  else
    CanClose := False;
end;
```

У цьому випадку складний логічний вираз

```
(OpenDialog1.Execute) and FileExists(OpenDialog1.FileName)
```

записаний в операторі **if** процедури `TForm1.Button1Click`, можна спростити, залишивши тільки його першу складову частину:

```
if OpenDialog1.Execute then
```

Процедура `TForm1.OpenDialog1CanClose` при такому способі розв'язання задачі не дозволить закрити діалогове вікно при виборі неіснуючого файлу.

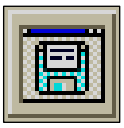
Найчастіше дії з відкриття файлів реалізують у головному меню форми. Наступний приклад демонструє цю можливість.

```
//Приклад 21.2
//Вивести в комбінований список імена файлів, вибраних
//у вікні відкриття файлу з дозволеним множинним вибором.
```

Розмістимо на формі вікно відкриття файлу OpenDialog1, головне меню та комбінований список ComboBox1. У головному меню створимо єдину опцію Файл, що містить Одну підопцію. Надамо властівостям Name та Caption цієї підопції значення Open та Відкрити відповідно. Для вікна відкриття файлу встановимо можливість множинного вибору, для чого в Інспекторі Об'єктів кліком мишкою над ім'ям властивості Options відкриємо список значень, що включаються в нього, і виберемо значення True у рядку ofAllowMultiSelect. Сформульована задача в цьому випадку розв'язується за допомогою такого опрацьовувача події OnClick підопції Відкрити головного меню:

```
procedure TForm1.OpenClick(Sender: TObject);
begin
  OpenDialog1.Filter := 'Текстові файли|*.txt' +      //Фільтр
    '|Pascal-файли|*.pas|Проекти Delphi|*.dpr' +
    '|Delphi|*.dpr;*.pas;*.dfm';
  if (OpenDialog1.Execute)   //Якщо діалог завершився вибором,
                      //то переписуємо вміст списку в ComboBox1
  then ComboBox1.Items.Assign(OpenDialog1.Files);
end;
```

21.3. Вікно збереження файлу – компонент TSaveDialog



Компонент `TSaveDialog`, що служить для створення та обслуговування модального вікна збереження файлу, є дочірнім компонентом класу `TOpenDialog` і має властивості й методи, ідентичні однойменним властивостям і методам свого прабатька. Саме вікно відрізняється від вікна відкриття файлу тільки своїм заголовком і назвою кнопки підтвердження вибору.

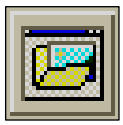
```
//Приклад 21.3
//Реалізувати вибір і завантаження файлів з розширенням .txt у
//багаторядковий текстовий редактор із забезпеченням
//можливості збереження вмісту редакторського вікна у файлі.
```

Розмістимо на формі головне меню з трьома опціями – Open, Save і SaveAs (дві останні з них зробимо недоступними), вікна відкриття (OpenDialog1) і збереження (SaveDialog1) файлу, а також багаторядковий текстовий редактор Memo1. Запишемо також у властивості DefaultExt кожного з діалогових вікон текст txt, а у властивості Options вікна SaveDialog1 установимо в стан True прапорець ofNoReadOnlyReturn. Тоді для розв'язання задачі можуть бути використані такі опрацьовувачі події OnClick опцій головного меню:

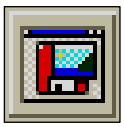
```
procedure TForm1.OpenClick(Sender: TObject);
begin
  OpenDialog1.Filter := 'Текстові файли|*.txt';
  if (OpenDialog1.Execute) then begin
   //Запам'ятовуємо повне ім'я файлу для наступного збереження
    SaveDialog1.FileName := OpenDialog1.FileName;
    if FileExists(OpenDialog1.FileName) then
                      //Якщо файл існує, то завантажуємо його
      Memo1.Lines.LoadFromFile(OpenDialog1.FileName)
    else Memo1.Clear;  //Новий файл - очищаємо для нього вікно
    Memo1.Visible := True;
    Save.Enabled := True;     //Встановлюємо доступність опції
    SaveAs.Enabled := True;   //Встановлюємо доступність опції
  end;
end;

procedure TForm1.SaveClick(Sender: TObject);
begin
  if FileExists(SaveDialog1.FileName)        //Якщо файл існує
     and ((FileGetAttr(SaveDialog1.FileName)     //і доступний
     and faReadOnly) <> 0)                 //тільки для читання,
  then begin           //то виводимо відповідне повідомлення й
    Application.MessageBox(PChar('Файл '+SaveDialog1.FileName
       + ' доступний тільки для читання. Укажіть інше ім''я'),
       'Увага', MB_OK);
    SaveAsClick(Self);    //звертаємося до опції "Зберегти Як"
  end
  else           //У противному випадку просто зберігаємо файл
    Memo1.Lines.SaveToFile(SaveDialog1.FileName);
end;

procedure TForm1.SaveAsClick(Sender: TObject);
begin
  if (SaveDialog1.Execute) then
    if not (FileExists(SaveDialog1.FileName)       //Див. вище
       and ((FileGetAttr(SaveDialog1.FileName)
       and faReadOnly) <> 0))
    then begin
```

```
      Memo1.Lines.SaveToFile(SaveDialog1.FileName);
      SaveDialog1.FileName := SaveDialog1.FileName;
    end;
end;
```

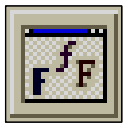
21.4. Вікна відкриття та збереження зображень – компоненти TOpenPictureDialog і TSavePictureDialog



Компоненти, що служать для створення модальних діалогових вікон завантаження та збереження графічних зображень `TOpenPictureDialog` і `TSavePictureDialog`, відрізняються від компонентів `TOpenDialog` і `TSaveDialog` лише незначними змінами діалогових вікон. По-перше, у їх діалогових вікнах передбачена наявність стандартних фільтрів для відбору графічних файлів у BMP-, ICO-, EMF- та WMF-форматах (як для кожного окремо, так і спільно для всіх чотирьох). По-друге, діалогові вікна цих компонентів містять панелі з вбудованим компонентом для попереднього перегляду графічних файлів у процесі їх вибору.



21.5. Вікно вибору шрифту – компонент TFontDialog



Компонент `TFontDialog` призначений для створення та обслуговування стандартного Widows-вікна вибору шрифту. При підтвердженні вибору шрифту (кліку над кнопкою OK діалогового вікна) обраний шрифт зберігається у властивості

```
property Font: TFont;
```

цього компонента (про особливості цього класу див. у п. 14.2.3).

Мінімальний і максимальний розміри шрифту (в пунктах) визначають відповідно властивості:

```
property MinFontSize: Integer;
property MaxFontSize: Integer;
```

За настроювання на тип пристрою, для якого вибирається шрифт, відповідає властивість `Device`, що оголошена в такий спосіб:

```
type
  TFontDialogDevice = (fdScreen, fdPrinter, fdBoth);
property Device: TFontDialogDevice;
```

Залежно від того, якого значення набуває ця властивість, можна вибрати шрифт, що відображається тільки на екрані (`fdScreen`), тільки на принтері (`fdPrinter`) або на двох цих пристроях одночасно (`fdBoth`).

Як і в раніше розглянутих компонентів організації діалогів, діалогове вікно, створюване компонентом `TFontDialog`, налаштовується за допомогою зміни властивості `Options`:

```
type
  TFontDialogOption = (fdAnsiOnly, fdTrueTypeOnly,
  fdEffects, fdFixedPitchOnly, fdForceFontExist,
  fdNoFaceSel, fdNoOEMFonts, fdNoSimulations,
  fdNoSizeSel,fdNoStyleSel,fdNoVectorFonts,fdShowHelp,
  fdWysiwyg,fdLimitSize,fdScalableOnly,fdApplyButton);
  TFontDialogOptions = set of TFontDialogOption;
property Options: TFontDialogOptions;
```

Зміст констант, що визначають тип `TFontDialogOption`, такий:

- ✓ `fdAnsiOnly` – показує в діалоговому вікні тільки шрифти, що використовують множину символів Windows;
- ✓ `fdTrueTypeOnly` – показує в діалоговому вікні тільки TrueType-шрифти;
- ✓ `fdEffects` – показує в діалоговому вікні прапорці видозміни шрифту (Підкреслений і Закреслений) і список вибору кольорів шрифту;
- ✓ `fdFixedPitchOnly` – показує в діалоговому вікні тільки моноширинні шрифти;
- ✓ `fdForceFontExist` – забороняє вибирати шрифти, не виведені в діалоговому вікні, з видачею повідомлення про помилку при спробі такого вибору;
- ✓ `fdNoFaceSel` – забезпечує відкриття діалогового вікна без виділення імені шрифту;
- ✓ `fdNoOEMFonts` – забороняє виведення у діалоговому вікні шрифтів MS DOS;
- ✓ `fdNoSimulations` – показує в діалоговому вікні тільки шрифти та стилі, які безпосередньо створюються файлом визначення шрифту, крім напівжирних і курсивних шрифтів, що синтезуються Windows;
- ✓ `fdNoSizeSel` – забезпечує відкриття діалогового вікна без виділення розміру шрифту;
- ✓ `fdNoStyleSel` – забезпечує відкриття діалогового вікна без виділення стилю шрифту;

- ✓ `fdNoVectorFonts` – забороняє показ у діалоговому вікні векторних шрифтів;
- ✓ `fdShowHelp` – включає в діалогове вікно кнопку Help (Довідка);
- ✓ `fdWysiwyg` – показує в діалоговому вікні тільки шрифти, підтримувані і екраном, і принтером;
- ✓ `fdLimitSize` – забезпечує врахування властивостей `MinFontSize` і `MaxFontSize`, що обмежують розміри шрифту;
- ➢ `fdScalableOnly` – показує в діалоговому вікні тільки векторні та TrueType-шрифти (тобто тільки масштабовані шрифти);
- ✓ `fdApplyButton` – включає в діалогове вікно кнопку Apply (Застосувати).

За умовчанням у множину `Options` з всіх перерахованих значень включається тільки значення `fdScalableOnly`.

Крім успадкованих від `TCommonDialog` подій OnShow і OnClose, компонент `TFontDialog` характеризує також подія OnApply, описувана в такий спосіб:

```
type  DApplyEvent = procedure(Sender: TObject;
                              Wnd: HWND) of object;
property OnApply: TFDApplyEvent;
```

Ця подія виникає, коли користувач клацає мишкою в діалоговому вікні над кнопкою Apply (Застосувати, Применить).

Наступний приклад ілюструє методику зміни властивості `Options` компонента `TFontDialog` (у цьому випадку з метою включення в діалогове вікно вибору шрифту кнопки Apply). Крім того, у наведеному тексті реалізовано опрацьовувач події OnApply.

```
//Приклад 21.4
//Організувати вибір шрифту для всіх компонентів TButton,
//власниками яких є головна форма.
procedure TForm1.Button1Click(Sender: TObject);
begin
 FontDialog1.Options := FontDialog1.Options + [fdApplyButton];
 FontDialog1.Execute;
end;
procedure TForm1.FontDialog1Apply(Sender: TObject; Wnd: HWND);
var
  i: Integer;
begin
  for i := 0 to ComponentCount - 1 do
    if Components[i] is TButton then
      TButton(Components[i]).Font.Assign(FontDialog1.Font);
end;
```

21.6. Вікно вибору кольору – компонент TColorDialog

Компонент `TColorDialog` показує стандартне Windows-вікно для вибору кольору. Вікно не виводиться на екран, поки не буде виконано його активізацію за допомогою методу `Execute`. При кліку мишкою над кнопкою OK у діалоговому вікні воно закривається, а обраний колір зберігається у властивості `Color` даного компонента:

```
property Color: TColor;
```

У процесі виконання програми користувач може визначити до 16 користувальницьких кольорів, інформація про які у вигляді рядків у форматі

`Color`Літера = номер_кольору

зберігається у властивості

```
property CustomColors: TStrings;
```

У рядках для визначення користувальницьких кольорів Літера – латинська літера від `A` до `P`, номер_кольору – шістнадцятковий номер RGB-кольору (наприклад, рядок `ColorB=A3AB28` визначає другий користувальницький колір). Після їх визначення користувальницькі кольори можуть брати участь у виборі в процесі виконання програми.

Доступ до властивості може реалізовуватися як програмно, так і за допомогою Інспектора Об'єктів після кліку мишкою над кнопкою  у рядку з ім'ям цієї властивості.

Налаштування вікна вибору кольору виконується зміною його властивості `Options`, що має таке визначення:

```
type
  TColorDialogOption = (cdFullOpen, cdPreventFullOpen,
    cdShowHelp, cdSolidColor, cdAnyColor);
  TColorDialogOptions = set of TColorDialogOption;
property Options: TColorDialogOptions;
```

Зміст можливих значень, що включаються у властивість `Options`, такий:

- ✓ `cdFullOpen` – при відкритті вікна показує розгорнуту палітру вибору кольору;
- ✓ `cdPreventFullOpen` – при відкритті вікна робить недоступною кнопку Define Custom Colors (Визначити Колір), забороняючи користувачеві визначити новий колір;
- ✓ `cdShowHelp` – показує у вікні кнопку Help (Довідка);

- ✓ `cdSolidColor` – вказує операційній системі на необхідність вибору найближчого суцільного кольору;
- ✓ `cdAnyColor` – дозволяє вибір несуцільних кольорів.

За умовчанням множина `Options` є порожньою.

Для компонента `TColorDialog` визначені тільки події OnShow і OnClose.

21.7. Вікно пошуку – компонент TFindDialog

Компонент `TFindDialog` створює стандартне Windows-вікно пошуку, що дозволяє проводити пошук тексту у файлі. Як і у випадку інших діалогових вікон, показ цього вікна реалізуються за допомогою методу `Execute`.

Текст, пошук якого виконується, міститься у властивості

```
property FindText: string;
```

Якщо цю властивість визначити до звертання до методу `Execute` або в Інспекторі Об'єктів, то текст буде автоматично з'являтися при першому показі вікна.

Діалогове вікно настроюється за допомогою його властивості `Options`, що дозволяє зробити недоступними або невидимими деякі елементи керування, а також забезпечити зміну враховуваних параметрів тексту і напрямку пошуку:

```
type TFindOption = (frDown, frFindNext,
  frHideMatchCase, frHideWholeWord, frHideUpDown,
  frMatchCase, frDisableMatchCase, frDisableUpDown,
  frDisableWholeWord, frReplace, frReplaceAll,
  frWholeWord, frShowHelp);
  TFindOptions = set of TFindOption;
property Options: TFindOptions;
```

У множину `Options` можна включити такі значення:

- ✓ `frDown` – встановлює за умовчанням перемикач Down (Униз);
- ✓ `frFindNext` – повідомляє програмі про клік над кнопкою Find Next (Знайти далі);
- ✓ `frHideMatchCase` – видаляє з вікна прапорець Match Case (З урахуванням регістру);
- ✓ `frHideWholeWord` – видаляє з вікна прапорець Whole Word (Тільки слово цілком);
- ✓ `frHideUpDown` – видаляє з вікна перемикачі напрямку пошуку;

- ✓ `frMatchCase` – встановлює у вікні прапорець Match Case (З урахуванням регістру);
- ✓ `frDisableMatchCase` – встановлює у вікні недоступність для прапорця Match Case (З урахуванням регістру);
- ✓ `frDisableUpDown` – встановлює у вікні недоступність для перемикачів напрямку пошуку;
- ✓ `frDisableWholeWord` – встановлює у вікні недоступність для прапорця Match Whole Word (Тільки слово цілком);
- ✓ `frReplace` – враховується тільки у вікні пошуку та заміни і вказує на те, що в діалоговому вікні натискалася кнопка Replace (Замінити);
- ✓ `frReplaceAll` – ураховується тільки у вікні пошуку та заміни і указує на те, що в діалоговому вікні натискалася кнопка Replace All (Замінити все);
- ✓ `frWholeWord` – встановлює у вікні прапорець Match Whole Word (Тільки слово цілком);
- ✓ `frShowHelp` – показує у вікні кнопку Help (Довідка).

За умовчанням у множину `Options` включене тільки значення `frDown`.

Прапорці `frFindNext`, `frReplace` та `frReplaceAll` при відкритті діалогового вікна скидаються операційною системою й установлюються автоматично при натисканні відповідних їм кнопок. За фактом включення в множину `Options` одного зі значень `frReplace` і `frReplaceAll` можна довідатися, чи натискалася у вікні пошуку та заміни одна з кнопок Replace (Замінити) і Replace All (Замінити все), а при позитивній відповіді встановити, яка саме кнопка натискалася.

Для компонента `TFontDialog`, крім успадкованих від свого прабатька `TCommonDialog` подій OnShow і OnClose, визначена також подія OnFind, описувана в такий спосіб:

```
property OnFind: TNotifyEvent;
```

Ця подія відбувається як результат кліку користувачем над кнопкою Find Next (Знайти далі).

```
//Приклад 21.5
//Організувати пошук тексту усередині вікна багаторядкового
//редактора униз від позиції курсору в цьому вікні.
```

Будемо використовувати властивість Tag вікна пошуку файлу FindDialog1 як змінну для зберігання значення зсуву від початку опрацьовуваного тексту з метою зазначення позиції, з якої продовжується пошук при натисканні в діалоговому вікні кнопки Find Next (Знайти далі).

Виведення вікна пошуку файлу та початкових його установок забезпечимо за допомогою опрацьовувача події OnClick кнопки Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with FindDialog1 do begin
    Tag := 0;                        //Початковий зсув дорівнює 0
    Options := Options + [frDisableUpDown];//Забороняємо зміну
    Execute;     //напрямку пошуку та виводимо діалогове вікно
  end;
end;
```

Створимо також опрацьовувач події OnFind для вікна пошуку файлу FindDialog1:

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
  SelPos: Integer;      //Номер позиції, у якій знайдено текст
begin
  with TFindDialog(Sender) do begin
      //Спочатку шукаємо фрагмент тексту, починаючи з позиції,
     //зміщеної на значення TFindDialog.Tag від початку тексту
    SelPos := Pos(FindText,PChar(@Memo1.Lines.Text[1]) + Tag);
    if SelPos > 0 then begin          //Пошук виявився успішним
      Memo1.SelStart:=Tag+SelPos-1;//Початок виділеного тексту
      Memo1.SelLength := Length(FindText);   //та його довжина
      Tag := Memo1.SelStart + Length(Memo1.SelText);    //Зсув
    end
    else MessageDlg('Текст "' + FindText +
                '" в Memo1 не знайдений', mtError, [mbOk], 0);
  end;
end;
```

21.8. Вікно пошуку та заміни – компонент TReplaceDialog

Компонент TReplaceDialog є безпосереднім спадкоємцем компонента TFindDialog і служить для створення стандартного Windows-вікна пошуку та заміни фрагмента тексту з використанням деякого рядка заміщення.

Порівняно з вікном пошуку, у вікні пошуку та заміни відсутні перемикачі напрямку пошуку і, крім того, в нього додані редаговане поле для відображення тексту заміни та кнопки Replace (Замінити) і Replace All (Замінити все). Відповідно до цього в компонента TReplaceDialog додатково визначена властивість ReplaceText, що містить текст заміни, і

подія OnReplace, що виникає при кліку над однією кнопок Replace (Замінити) або Replace All (Замінити все):

```
property ReplaceText: string;
property OnReplace: TNotifyEvent;
```

Упізнавання кнопки, що натискалася, можна здійснити в опрацьовувачі події OnReplace перевіркою включення в множину `Options` значень `frReplace` та `frReplaceAll`.

```
//Приклад 21.6
//Організувати пошук і заміну тексту у вікні багаторядкового
//редактора вниз від позиції курсору в цьому вікні. Кнопка
//Replace повинна забезпечувати заміну виділеного фрагмента,
//а ReplaceAll – заміну з виділенням наступного фрагмента.
```

Замінимо на формі з прикладу 21.5 вікно пошуку файлу FindDialog1 вікном пошуку та заміни ReplaceDialog1, а в текстах двох опрацьовувачів подій замінимо рядок `FindDialog` рядком `ReplaceDialog`. Процедура обробки події OnReplace може мати такий вид:

```
procedure TForm1.ReplaceDialog1Replace(Sender: TObject);
begin
  with TReplaceDialog(Sender) do  begin
    if Memo1.SelText<>FindText then begin//Текст не знайдений?
      ReplaceDialog1Find(Sender);      //Шукаємо текст у вікні
      Exit;
    end;
    if (frReplace in Options) or (frReplaceAll in
       Options) then //Якщо натиснута одна з кнопок заміни, то
    begin
      Memo1.SelText := ReplaceText;        //заміняємо текст і
      Tag := Memo1.SelStart;              //фіксуємо новий зсув
      if frReplaceAll in Options then          //Для ReplaceAll
        ReplaceDialog1Find(Sender);          //повторний пошук
    end;
  end;
end;
```

21.9. Діалогові вікна для керування друком – компоненти TPrintDialog, TPageSetupDialog і TPrinterSetupDialog

Компонент `TPrintDialog` служить для створення стандартного Windows-вікна для вибору параметрів друку. При цьому вигляд і можливості виведеного за допомогою цього компонента вікна залежать від типу принтера.

Компонент `TPageSetupDialog` забезпечує створення та обслуговування стандартного Windows-вікна, призначеного для налаштування принтера. Створюване за допомогою цього компонента вікно взаємодіє з драйвером принтера, у зв'язку з чим його вигляд залежить від типу принтера.

Компонент `TPageSetupDialog` служить для створення та обслуговування стандартного модального діалогового вікна настроювання параметрів друку сторінки (зокрема, для керування розміром сторінки, її орієнтацією, розмірами полів, форматом (розміром) паперу тощо).

Докладні відомості про ці компоненти можна одержати, скориставшись довідковою службою Delphi.

Запитання для контролю і самоконтролю

1. Опишіть загальну методику роботи з діалоговими вікнами.
2. Чи можлива робота з діалоговим вікном без виклику його методу `Execute`?
3. Опишіть загальну методику роботи з діалоговими вікнами відкриття та збереження файлу.
4. Для чого і як використовується властивість `Filter` у діалогових вікнах відкриття та збереження файлу?
5. Яке призначення властивості `InitialDir`?
6. Яке призначення властивості `Options` у діалогових вікнах?
7. Опрацьовувачі яких подій повинні бути визначені для того, щоб забезпечити виконання яких-небудь дій перед відкриттям діалогового вікна і після його закриття?
8. Що таке події OnFolderChange та OnTypeChange? Як вони пов'язані з подією OnSelectionChange?
9. Опрацьовувач якої події повинен бути визначений, якщо необхідно виконати які-небудь дії при закритті вікна за допомогою кнопки підтвердження вибору у випадку роботи з вікнами відкриття та збереження файлу (зображення)?
10. Опишіть загальну методику роботи з діалоговими вікнами відкриття та збереження зображення.
11. Опишіть роботу з вікном вибору шрифту.
12. Опишіть роботу з вікном вибору кольору.
13. Охарактеризуйте вікно пошуку.
14. Які основні події характеризують вікна пошуку файлу?
15. Охарактеризуйте вікно пошуку та заміни.

16. Яка додаткова подія є у компонента пошуку та заміни порівняно з компонентом пошуку?

Завдання для практичного відпрацьовування матеріалу

1. Створити форму з компонентом Image і забезпечити завантаження в нього зображення (у тому числі у форматі JPEG), вибір якого організувати за допомогою діалогового вікна відкриття зображення.
2. Для Memo-компонента, розміщеного на формі, організувати налаштування шрифту при виконанні програми.
3. Зобразити коло, виконавши вибір кольору контуру і кольору заливання за допомогою вікна вибору кольору.

СПИСОК ЛІТЕРАТУРИ

1. Безменов, М. І. Турбо Паскаль 7.0 : навч. посіб. / М. І. Безменов. – Х. : НТУ «ХПІ»; Парус™, 2005. – 240 с.
2. Голованов, М. Е. Создание компонентов в среде Delphi / М. Е. Голованов, Е. О. Веселов. – СПб. : БХВ-Петербург, 2004. – 320 с.
3. Кэнту, М. Delphi 7 : Для профессионалов / М. Кэнту – СПб. : Питер, 2004. – 1101 с.
4. Архангельский, А. Я. Приемы программирования в Delphi 6 на основе VCL / А. Я. Архангельский. – М. : ООО «БИНОМ-Пресс», 2006. – 944 с.
5. Архангельский, А. Я. Программирование в Delphi 6 / А. Я. Архангельский. – М. : БИНОМ, 2002. – 1120 с.
6. Дарахвелидзе, П. Г. Программирование в Delphi 7 / П. Г. Дарахвелидзе, Е. П. Марков. – СПб. : БХВ-Петербург, 2003. – 784 с.
7. Культин, Н. Б. Основы программирования в Delphi 7 / Н. Б. Культин. – СПб.: БХВ-Петербург, 2003. – 608 с.
8. Пестриков, В. М. Delphi на примерах / В. М. Пестриков, А. Н. Маслобоев. – СПб. : БХВ-Петербург, 2005. – 496 с.
9. Ремкеев, А. А. Курс Delphi для начинающих. Полигон нестандартных задач / А. А. Ремкеев, С. В. Федотова. – М. : СОЛОН-Пресс, 2006. – 360 с.
10. Митчелл, К. Керман. Программирование и отладка в Delphi™ : учебный курс / Митчелл К. Керман. – М. : Вильямс, 2004. – 720 с.
11. Сухарев, М. В. Основы Delphi : Профессиональный подход / М. В. Сухарев. – СПб. : Наука и Техника, 2004. – 600 с.
12. Парижский, С. М. Delphi : Только практика / С. М. Парижский. – К. : МК-Пресс, 2005. – 208 с.
13. Культин, Н. Б. Основы программирования в Delphi 2007 / Н. Б. Культин. – СПб. : БХВ-Петербург, 2008. – 480 с.

Навчальне видання

БЕЗМЕНОВ Микола Іванович

ОСНОВИ ПРОГРАМУВАННЯ В СЕРЕДОВИЩІ DELPHI

Навчальний посібник для студентів вищих навчальних закладів, які навчаються за напрямами галузі знань «Інформатика та обчислювальна техніка» та «Інформатика»

Роботу до видання рекомендував проф. О. В. Горелий

Редактор А. А. Копієвська

План 2008 р., поз. 29/36–10

Підписано до друку 15.03.10. Формат 60×84 1/16.
Папір офсетний. Гарнітура Times New Roman. Друк офсетний.
Умов. друк. арк. 34,0. Обл.-вид. арк. 42,5. Наклад 400 прим. Зам. № 17.

Видавничий центр Національного технічного університету
«Харківський політехнічний інститут».
61002, Харків, вул. Фрунзе, 21.
Свідоцтво про державну реєстрацію ДК № 3657 від 24.12.2009 р.

ФОП Стеценко І. І., 61019, Харків, пр. Ілліча, 103а, кв. 21,
тел. 758-17-35