Основи програмування серед Lazarus

Зміст

Глава 1 Основи програмування

1.1. Концепція алгоритму.

Комп'ютер - пристрій для вирішення завдань. Не обов'язково задач чисто математичного характеру. Це може бути завдання управління верстатами чи ракетами, і завдання планування виробництва, і завдання інформаційно-довідкового обслуговування, і завдання обробки гіпертекстової інформації та мультимедіа, тобто. обробки звукової та відеоінформації. Щоб розв'язати якусь завдання на комп'ютері необхідно спочатку придумати як її взагалі вирішити, тобто. придумати алгоритм її розв'язання. Алгоритм – одна із наріжних понять інформатики та програмування.

Отже, що розуміється під алгоритмом?

Алгоритм - це строга і чітка, кінцева система правил, яка визначає послідовність дій над деякими об'єктами і після кінцевого числа кроків призводить до досягнення поставленої мети.

З визначення алгоритму випливає, що він повинен задовольняти таку вимогу:

1) кінцівка (фінітність)

Алгоритм завжди повинен закінчуватись після кінцевого числа кроків. Процедуру, що має всі характеристики алгоритму, за винятком кінцівки, викликають обчислювальним методом.

2) визначеність (детермінованість)

Кожен крок алгоритму має бути суворо визначено. Дії, які необхідно зробити, повинні бути суворо і недвозначно визначені в кожному можливому випадку, так що якщо дати алгоритм кільком людям, то вони, діючи за цим алгоритмом, отримували один і той самий результат. Оскільки звичайна мова сповнена двозначностей, то щоб подолати цю скруту, для опису алгоритмів розроблено формально визначені

мови програмування, або машинні мови, у яких кожне твердження має абсолютно точний зміст.

Запис алгоритму мовою програмування називається програмою.

3) Алгоритм повинен мати кілька вхідних даних, тобто. величин, об'єктів заданих йому на початок роботи. Ці дані беруться з певної множини об'єктів.

4) Алгоритм має чи кілька вихідних величин, тобто. величин, що мають цілком певне відношення до вхідних даних.

5) Ефективність

Від алгоритму вимагають, щоб він був ефективним. Це означає, що всі операції, які необхідно зробити в алгоритмі, мають бути досить простими, щоб їх у принципі можна було виконати точно і за кінець часу за допомогою олівця та паперу.

Слід зазначити, що для практичних цілей "фінітність" є найважливішою вимогою – алгоритм, що використовується, повинен мати не просто кінцеву, а гранично кінцеву, розумну кількість кроків. Наприклад, у принципі є алгоритм, що визначає, чи є початкове положення в грі в шахи форсовано вигранim для білих чи ні. Але для цього алгоритму потрібен фантастично величезний проміжок часу. Нехай є комп'ютер, що має швидкодію 100 млн. операцій на секунду. Тоді цей комп'ютер виконуватиме алгоритм протягом 1023 років. Для порівняння зазначимо, що період часу від початку виникнення життя на землі і до наших днів набагато менший за 1023 роки.

приклад алгоритму.

1.1.1 Алгоритм Евкліда.

Алгоритм Евкліда знаходження найбільшого спільного дільника двох цілих чисел, тобто. найбільше ціле число, яке поділяє націло задані числа.

1. Розглянути як перше число і як друге число. До п.2.

2. Порівняти перше та друге числа. Якщо вони рівні, перейти до п.5. Якщо ні, перейти до п.3.

3. Якщо перше число менше за друге, то переставити їх місцями. До п.4.

4. Відняти з першого числа друге та розглянути отриману різницю як нове перше число. До п.2.

5. Розглянути перше число як наслідок.

Стоп.

Цей набір правил алгоритмом, т.к. слідуючи йому, будь-яка людина, що вміє віднімати, може отримати найбільший спільний дільник для будь-якої пари чисел. Дотримуючись цього алгоритму, знайдемо НОД чисел 544 та 119.

$$\begin{matrix} A = 544 \\ У \end{matrix} \quad = 119$$

1)
$$\begin{array}{r} 544 \\ -\ 119 \\ \hline 425 \end{array}$$

$$\begin{matrix} A = 425 \\ У \end{matrix} \quad = 119$$

2)
$$\begin{array}{r} 425 \\ -\ 119 \\ \hline 306 \end{array}$$

$A = 306 \quad = 119$
$\quad\quad V$

3)
$$\begin{array}{r} 306 \\ -\ 119 \\ \hline 187 \end{array}$$
$A = 187 \quad V = 119$

4)
$$\begin{array}{r} 187 \\ -\ 119 \\ \hline 68 \end{array}$$
$A = 68 \quad V = 119$

5)
$A < B$

Змінюємо місцями

$A = 119 \quad V = 68$

6)
$$\begin{array}{r} 119 \\ -\ 68 \\ \hline 51 \end{array}$$
$A = 51 \quad V = 68$

7)
$A < B$

$A = 68 \quad V = 51$

8)
$$\begin{array}{r} 68 \\ -\ 51 \\ \hline 17 \end{array}$$
$A = 17 \quad V = 51$

9)
$A < B$

$A = 51$ B $= 17$

10)
$$\begin{array}{r} 51 \\ -\ 17 \\ \hline 34 \end{array}$$

$A = 34 \quad У = 17$

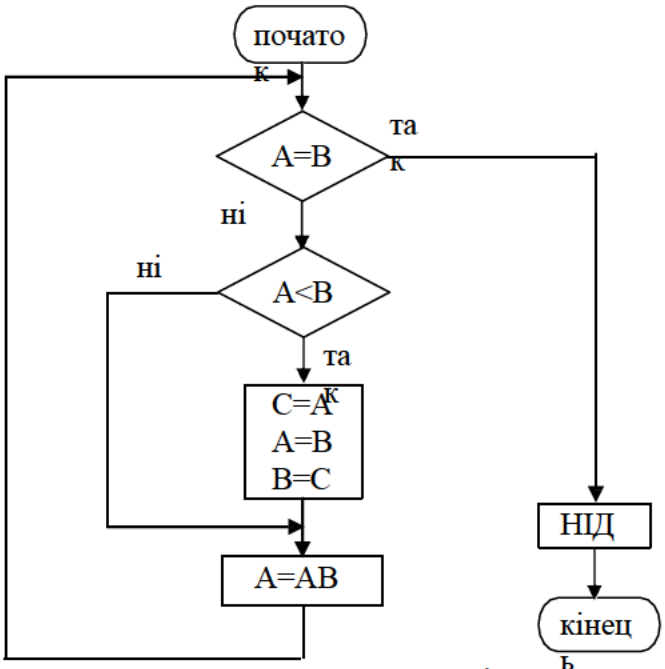
11)
$$\begin{array}{r} 34 \\ -\ 17 \\ \hline 17 \end{array}$$

A = B = 17 = НОД

Цей спосіб запису алгоритмів можливий, але незручний. По-перше, немає наочності, по-друге, "багатословенний". Одним із способів запису алгоритму є блок-схеми. Блок-схемою називається таке графічне зображення структури алгоритму, в якому кожен етап або крок процесу переробки даних представляється у вигляді прямокутника, ромба, овалу або іншої геометричної фігури, яка називається блоком.

Ці фігури поєднуються між собою лініями зі стрілками, що відображають послідовність виконання алгоритму. Усередині кожної фігури дозволяється писати довільний текст, у якому на зрозумілій людині мові повідомляються про необхідні обчислення у відповідній частині програми.

Прийнято певні стандарти графічних позначень. Так, прямокутник позначає обчислювальні дії, внаслідок яких змінюються значення даних. Ромбом позначають етап розгалуження алгоритму. Вибір одного з двох можливих напрямів подальшого рахунку здійснюється залежно від виконання умови, записаної у ромбі. Овалом позначають початок та кінець алгоритму. У паралелограмах записують процедури введення та виведення даних. Запишемо алгоритм Евкліда у вигляді блок-схеми:



Мал. 1.1. Алгоритм Евкліда

Існують інші способи запису алгоритмів. Зокрема, досить поширені записи алгоритмів на так званих псевдомовах. Псевдомова схожий на звичайну алгоритмічну мову програмування, але в ній можна використовувати і природну мову для більш чіткого вираження своїх думок. Деякі псевдомови більш близькі до мов високого рівня, так звані алголоподібні мови (у минулому існували мови програмування ALGOL-60, ALGOL-68, які нині не використовуються). Прикладом такої псевдомови є мова опису алгоритмів, що застосовуються в школах. Деякі псевдомови ближчі до машинних мов, звані асемблери. Прикладом такої мови опису алгоритмів служить мова, запропонована Д. Кнутом у його знаменитій книзі "Мистецтво програмування" [9].

Отже, щоб розв'язати якесь завдання, потрібно придумати алгоритм її розв'язання та записати його в тому чи іншому вигляді. Цей процес називається алгоритмізацією. Але в такому вигляді комп'ютер алгоритм зможе виконати. Слід

уявити цей алгоритм у вигляді, щоб комп'ютер міг його виконати. Для цього потрібно, по-перше, розбити алгоритм на елементарні операції, які називаються інструкціями або командами, які вміє виконувати комп'ютер, і, по-друге, записати кожну таку операцію мовою, зрозумілою комп'ютеру. Такий запис алгоритму мовою комп'ютера називається програмою. Процес розробки програми називається програмуванням. А людина, яка виконує цю роботу – програмістом. При цьому слід розрізняти програму в машинних кодах, кажуть ще програма і програму, написану якоюсь мовою програмування. Зазвичай людина пише програму мовою високого рівня, у крайньому разі, асемблері. Компілятор переводить її в програму машинною мовою, яку виконує комп'ютер.

Програмування – це наукова дисципліна. Якби процеси програмування різних завдань не мали між собою нічого спільного, то програмування як таке не було б науковою дисципліною. Але справа не так. Існують загальні методи, які дозволяють, поступово розчленовуючи завдання на підзавдання, зводити їх рішення, зрештою, до деяких елементарних операцій (найчастіше до елементарних арифметичних операцій), подібно до того, як розбираючи зовсім несхожі складні механізми, ми виявляємо, що вони складаються з однакових. підшипники, болти, гайки та ін.). З цього, звичайно, не випливає, що процес програмування не містить елементів творчості. Складання програми, такий самий творчий процес, як і розробка складного механізму із заданих наборів деталей.

Основне призначення алгоритмів полягає в їхньому фактичному виконанні тим чи іншим виконавцем. Тобто. алгоритм складається для того, щоб він був виконаний для вирішення будь-якої задачі. Як виконавець може виступати будь-хто – людина, верстат з ЧПУ, комп'ютер тощо.

У силу своєї природи комп'ютери стали найпоширенішими виконавцями алгоритмів. Власне, вони і були придумані для того, щоб з їх допомогою виконувати алгоритми.

Призначення комп'ютерів, таким чином, полягає у фактичному виконанні алгоритмів, розроблених людиною.

Важливо зрозуміти, що комп'ютер сам собою нічого не робить. Він лише сліпо виконує те, що вкаже йому людина. Тому, коли говорять, що комп'ютер керує верстатами, ракетами, складає музику, грає в шахи "як би самостійно", це не так!

Це людина вигадує алгоритм, тобто. спосіб управління верстатами, ракетами, твори музики, гри в шахи, а комп'ютер лише виконує цей алгоритм. Таким чином, комп'ютер це помічник, інструмент людини для вирішення різних завдань, але інструмент дуже потужний, різноплановий. За допомогою комп'ютера людина може вирішувати найрізноманітніші завдання.

Розглянемо приклад розробки алгоритму.

1.1.2 Завдання про поїзди та муху

Розглянемо завдання, яке ми запозичили з [11] та розглянемо на цьому прикладі основні прийоми та способи складання алгоритмів. Нехай два міста А та В видалені на відстань d=600 км. Одночасно з кожного міста відходять поїзди у напрямку один до одного. Поїзд, що вийшов із міста А, має швидкість v1=40 км/год, а з В – має швидкість v2=60 км/год. Одночасно з пункту А вилітає виключно швидка муха зі швидкістю v=200 км/год і летить на зустріч поїзду, що вийшов із пункту В. При зустрічі з ним муха робить поворот і летить назад назустріч з поїздом, що йде з А. Коли вона його зустріне, муха знову робить поворот і летить у зворотному напрямку. Так продовжується доти, доки поїзди не зустрінуться.

Завдання.

1) визначити довжину різних відрізків шляху, що пролетить муха
2) визначити загальну відстань, що пролетить муха

На друге питання легко відповісти: поїзди зустрінуться через 600/(40+60)=6 годин, а муха за цей час пролетить відстань 200*6=1200 км. Але залишимо осторонь цю хитрість, і вирішуватимемо завдання в «чоло».

Насамперед, потрібно скласти алгоритм, тобто. придумати, як вирішити завдання за допомогою елементарних кроків. Алгоритм записуватимемо у вигляді блок-схеми. З чого потрібно складати блок-схему? Як правило, блок-схема починається із введення вихідних даних та початкових установок, тобто. визначення початкових значень деяких допоміжних змінних. Але на початку буває важко відразу визначити початкові значення, тому малюють спочатку центральну частину блок-схеми (ядро), в якій визначені ті дії, які вирішують завдання, причому спочатку малюють укрупнену блок-схему, щоб якомога ясніше уявити весь процес розв'язання.

Які ідеї відразу спадають на думку?

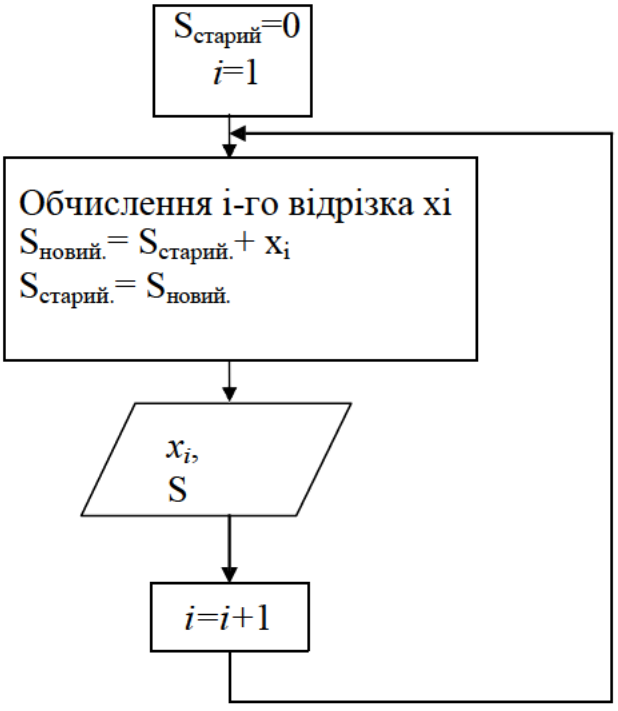
Найпростіша ідея полягає в тому, щоб робити дії наступним чином:

1) обчислити довжину першого відрізка шляху, який пролетить муха
2) обчислити довжину другого відрізка та додати до попереднього
3) обчислити довжину третього відрізка та скласти з раніше отриманою сумою тощо.

Ми можемо цей процес подати у вигляді

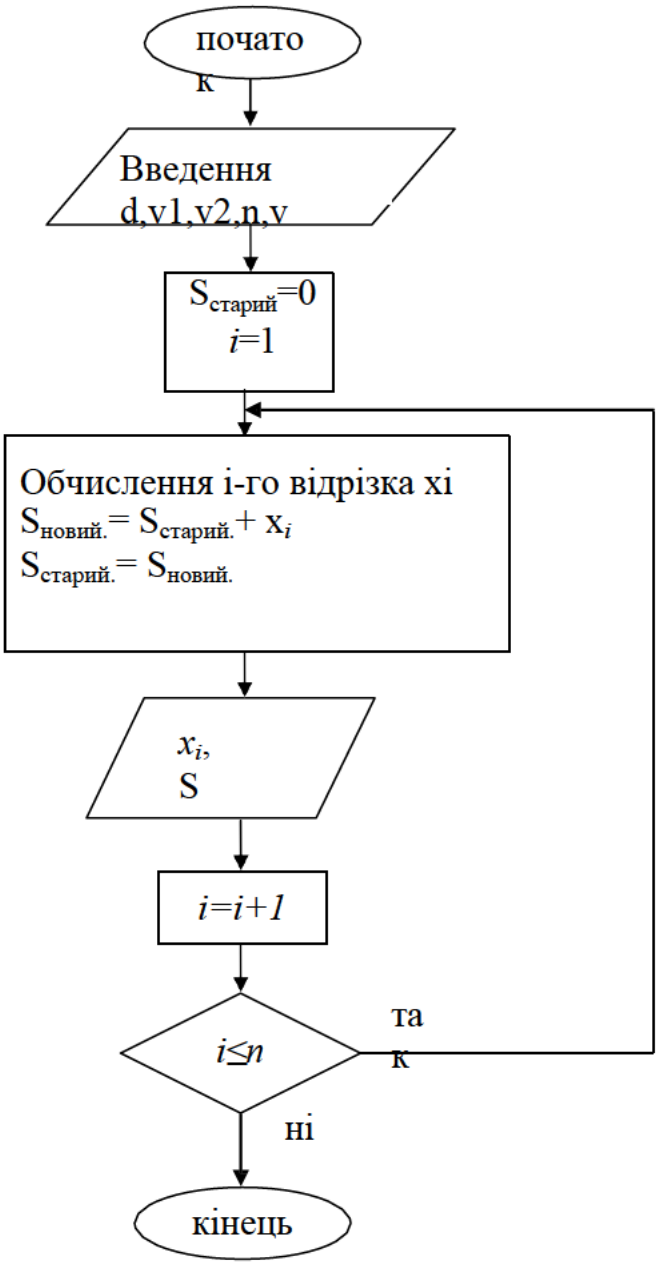
Мал. 1.2. Перший варіант алгоритму

Відразу виникає питання: т.к. схема повинна мати кінцеву довжину, потрібно засобами малюнка виразити поняття повторення, крім того, треба вказати які величини потрібно вивести на екран. Звідси схему перекреслюємо як:



Мал. 1.3. Другий варіант алгоритму

Трапляється, що коли деяка послідовність операцій закінчена, необхідно повернутися і повторити ці операції. Це називається циклом. На цій схемі видно серйозну ваду - обчислювальний процес за цією схемою ніколи не закінчиться. Необхідно вигадати критерій закінчення алгоритму. Поки не все ясно, будемо вважати, що нам задано число повторень n, а також задані всі вихідні дані, тоді блок-схема набуде вигляду:

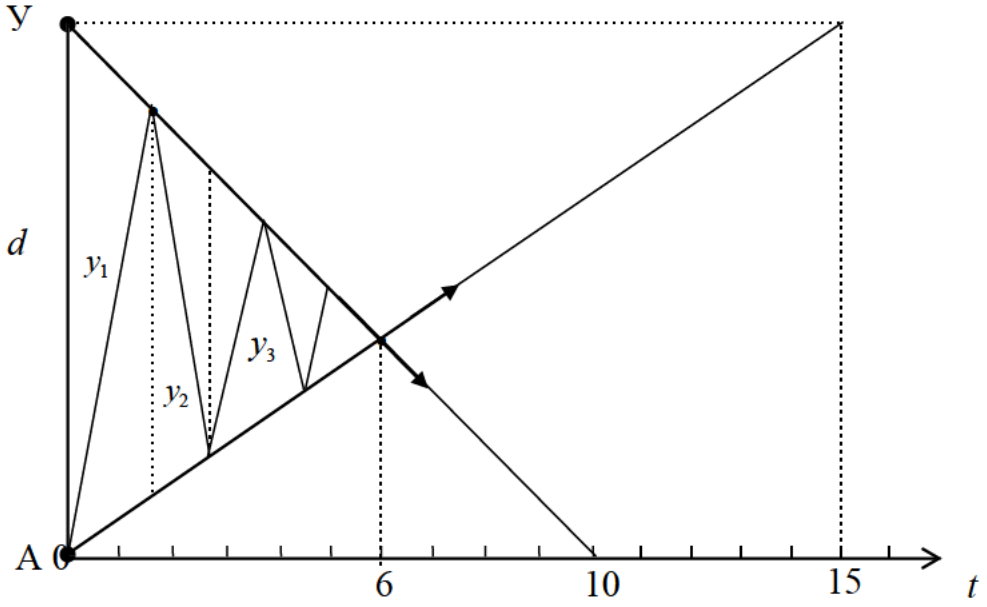


Мал. 1.4. Третій варіант алгоритму

Детальний аналіз відрізків колії.

Ми написали "обчислення відрізка хі". Але як же обчислювати ці відрізки? Ось тут багато хто "застряє" не в змозі придумати що-небудь. Єдиного рецепту як далі вигадувати алгоритм не існує. Відповідь одна - треба "просто" думати, шукати, пробувати! Пам'ятайте, що вигадувати, розробляти алгоритми – це такий самий творчий процес, як, наприклад, складати вірші, музику, писати картини тощо. Багато хто швидко придумує алгоритми, інші - важко і довго, ну а треті взагалі не можуть придумати навіть найпростіші алгоритми. Так що, шановний читачу, не кожен може стати програмістом! Для цього теж потрібні певні здібності до творчості, якщо бажаєте – таланту! Програміст – це творча професія! Зрозуміло, як і у будь-якій іншій творчій професії, знання теж відіграють важливу роль!

Повернемося до нашого завдання. Намалюємо графік полегшення.



Мал. 1.5. Графік руху поїздів

Розмірковуватимемо, використовуючи елементарні закони фізики про прямолінійний рівномірний рух тіл, які вивчаються в школі.

Коли муха вперше полетить у напрямку поїзда з пункту А, то до зустрічі з цим поїздом пройде час:

$$t_1 = \frac{d}{(\mathrm{v} + v_2)}; \tag{1.1}$$

У момент зустрічі мухи та поїзда В (будемо для стислості називати поїзд, що вийшов з пункту В поїздом В, а поїзд, що вийшов з пункту А, поїздом А) відстань між поїздами становитиме:

$$y_1 = d - t_1(\mathrm{v}_1 + v_2); \tag{1.2}$$

а муха пролетить відстань:

$$x = t_1 v; \tag{1.3}$$

І відповідно, при польоті мухи у зворотному напрямку маємо:

$$t_2 = \frac{y_1}{(\mathrm{v} + v_1)}, \qquad y_2 = y_1 - t_2(\mathrm{v}_1 + v_2), \qquad x = t_2 v; \tag{1.4}$$

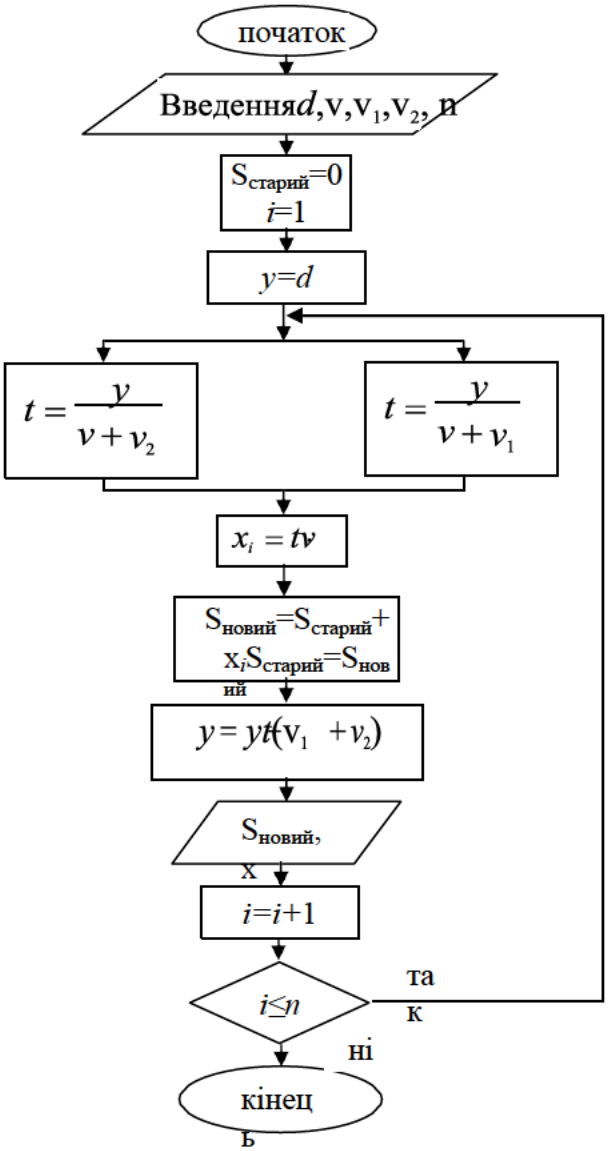
Звідси можна вивести загальну формулу:

$t = \frac{y}{v + v_2}$; $x = tv$ при польоті мухи з А до Ст. (1.5)

$t = \frac{y}{v + v_1}$; $x = tv$ при польоті мухи з У до А. (1.6)

$$y = y - t(\mathrm{v}_1 + v_2) \tag{1.7}$$

Намалюємо блок-схему з урахуванням отриманих формул:



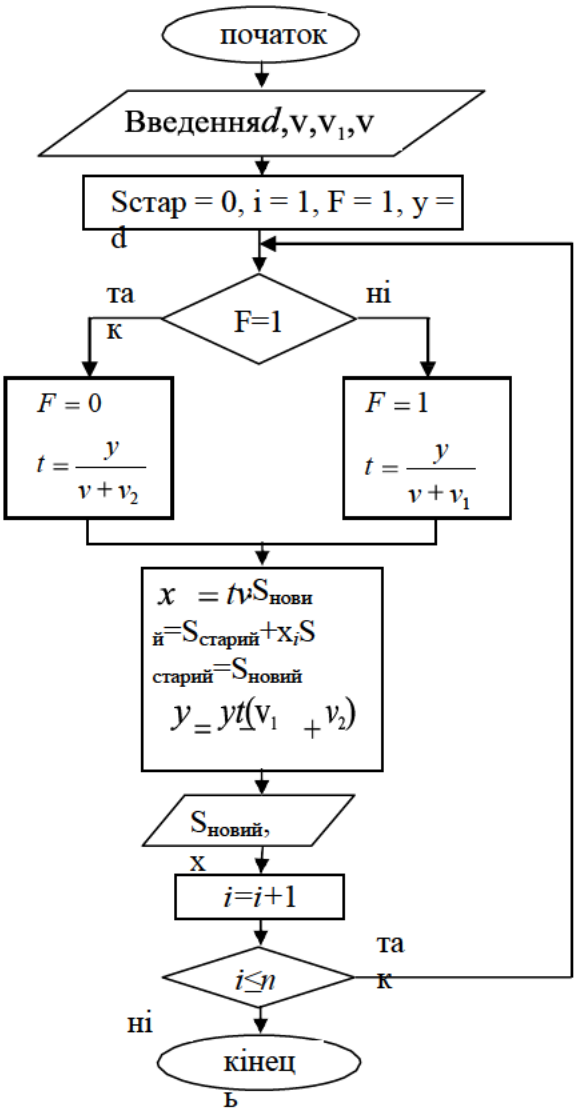
Мал. 1.6. Четвертий варіант алгоритму

Тут бачимо, що алгоритм має розгалужуватися на дві гілки (коли муха летить до поїзда і коли летить до поїзда А). Як комп'ютеру повідомити, що потрібно проходити поперемінно через ці гілки? Використовується прийом, який широко відомий у програмуванні та називається метод "прапорців" або "семафора". Вважатимемо що, якщо прапорець піднятий, то треба йти лівою гілочкою, якщо опущений, то правою. Як прапорець прийнято використовувати або цілу чисельну змінну, або булеву змінну, яка може приймати тільки два значення:

```
0 – означає, що прапорець опущений, 1 – означає, що прапорець піднято, якщо
```

це змінна цілого типу і false – прапорець опущений, true – прапорець піднятий, якщо це булева змінна.

Перемалюємо блок-схему



Мал. 1.7. П'ятий варіант алгоритму

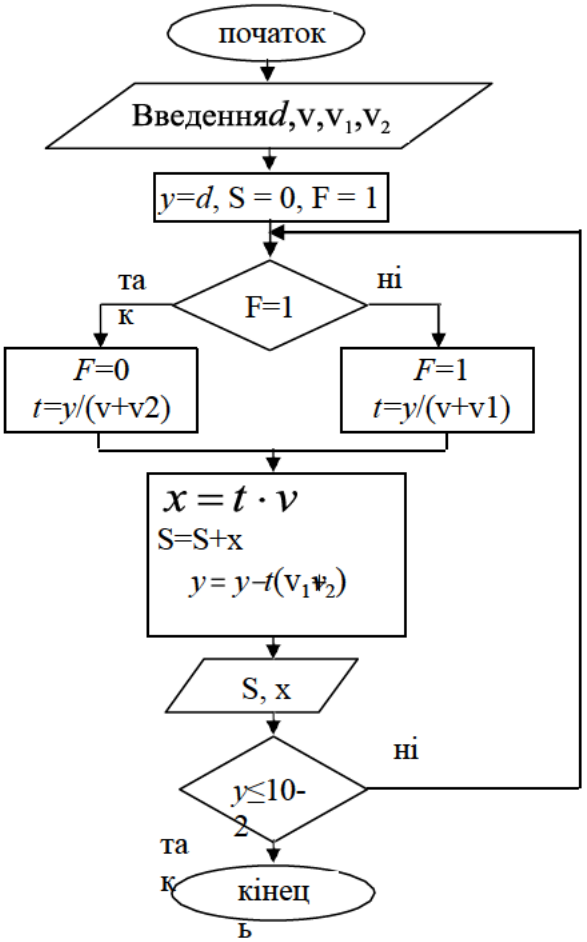
Збудувавши один варіант блок-схеми, завжди потрібно подивитися, чи не можна її спростити?

Аналізуючи блок-схему, бачимо, що ми поперемінно використовуємо Sстар, Sнов, причому після виведення на екран Sнов його значення нам не потрібно, воно все одно змінюється. Звідси можна використовувати лише одну змінну S.

S = S + xi

Далі: критерій закінчення алгоритму ми визначили не дуже добре.

Справді незрозуміло, чому дорівнює n. Може 100, а може 1000. Припустимо, ми прийняли n=100, а насправді число відрізків дорівнювало 10, тоді 90 разів алгоритм працюватиме «марно», т.к. отримані результати будуть безглуздими. Як бути? Чи не краще визначити кінець алгоритму з y. Справді, з малюнка видно, що y→0. Вважатимемо, що поїзди зустрілися, якщо y≤10-2. З іншого боку, бачимо, як і значення чергового відрізка xi після виведення їх у екран, нам потрібно, тобто. параметр і можна остаточно прибрати. Остаточно отримуємо:



Мал. 1.8. Остаточна блок-схема алгоритму

Вважатимемо, що алгоритм більше не спростити. У наступному розділі, коли вивчатимемо мову програмування Pascal, ми напишемо програму цього алгоритму (див. розділ 2, розділ 2.2).

1.1.3 Замість ліричного відступу

Які висновки можна зробити з аналізу цього завдання з погляду розробки алгоритму або, як ще кажуть, алгоритмізації цього завдання?

По-перше, вкрай рідко і для дуже простих завдань вдається одразу і без помилок записати алгоритм. У більшості випадків доводиться, як ми бачили, починати із найзагальнішого і дещо "розпливчастого" формулювання алгоритму. У будь-якому випадку варто навіть цей алгоритм записати як блок схеми (рис. 1.2). Тому що одна з безперечних переваг запису алгоритму у вигляді блок-схеми полягає в тому, що це дозволяє побачити структуру алгоритму.

Поступово ми деталізували алгоритм, з кожним кроком покращуючи та вдосконалюючи його. До речі, метод, яким ми скористалися, так і називається метод покрокової деталізації.

По-друге, багато молодих, особливо початківців, програмісти часто взагалі ігнорують етап складання алгоритму, і тільки-но встигнувши прочитати умову завдання, відразу сідають за комп'ютер і починають писати програму. Зустрічаються, звичайно, сверходарені люди, які можуть дозволити собі це, та й то не завжди. Але таких – одиниці! Більшості "смертних" доводиться поступово розробляти алгоритми і записувати їх у тому чи іншому вигляді.

По-третє, автор багаторазово спостерігав таку картину – студент отримує завдання, записує умову завдання собі у зошит та… застигає як сфінкс! Проходить хвилин п'ять, десять, двадцять, … багато минає часу, а він навіть не ворухнеться, не в змозі запропонувати щось путнє! Потім, коли питаєш у нього – про що ти думав у цей час, відповіді невизначені, найчастіше на кшталт "не знав з чого почати".

Крім стандартної відповіді – "якщо не знаєш з чого почати, почни з початку", що ще можна порадити:

1. Ще стандартніше - більше читайте книг, скрупульозно розбирайте

приклади, наведені у них. "Горе тому, хто читає лише одну книгу" (Джордж Герберт).

2. Одного читання та "розуміння" прикладів у книгах недостатньо. Треба самостійно тренуватися у складанні алгоритмів для різних завдань. Якщо можна так висловитись - тренуйте "міркування"! Найкращий спосіб такий. Якщо ви бачите у книзі докладно розібраний приклад, не поспішайте читати далі. Спробуйте скласти алгоритм самостійно! Це може забрати у вас досить багато часу. Не шкодуйте його! І тільки після завершення складання алгоритму можете звіритися з книгою. Спочатку у вас може нічого не виходити. Не впадайте у відчай, згодом прийде вміння і досвід.

3. Не соромтеся питати, переймайте досвід у інших. У мережі величезна кількість форумів із програмування. Зайдіть у будь-який із них, подивіться, які запитання там подаються і, знову, постарайтеся спочатку самі відповісти на це запитання і лише після цього можете дивитися, як на це запитання відповідають інші. Наприкінці книги наведено адреси кількох сайтів, у яких є форуми з програмування, зокрема форум, присвячений програмуванню в середовищі Lazarus.

4. У вас має бути створена (згодом, звичайно!) своя колекція алгоритмів, власних напрацювань, типових прийомів і навіть готових шматків коду, які ви вставлятимете в майбутні свої програми.

І, нарешті, часто зустрічаються ситуації, коли не допомагає ні досвід, ні знання. Як бути? Готових рецептів немає. І тут знову хочу наголосити, що програмування це творчість. А у творчості немає, і не може бути жодних готових рецептів. Тут, як то кажуть, справа в "іскре божій"! Доречно згадати і висловлювання Остапа Бендера - "блондин грає в шахи добре, брюнет грає погано і ніякі лекції не змінять цього співвідношення сил"!

1.2. Етапи підготовки задачі для вирішення на комп'ютері

Процес розв'язання задачі на комп'ютері складається з низки етапів, що включають як підготовку завдання до розв'язання, так і власне розв'язання.

Ці етапи включають:

1. постановку завдання;
2. побудова моделі досліджуваного процесу чи явища;
3. вибір методу розв'язання;
4. розробка алгоритму розв'язання задачі;
5. складання програми (кодування);
6. налагодження та тестування програми;
7. власне обчислення;
8. аналіз одержаних результатів.

Розглянемо ці етапи.

Постановка задачі.

Під постановкою задачі мається на увазі визначення мети або цілей, яких необхідно досягти в результаті розв'язання цього завдання. У постановку завдання входить визначення необхідної інформації (що дано), результату розв'язання задачі (що потрібно визначити) та вироблення загального підходу до розв'язання задачі.

Побудова відповідної моделі досліджуваного процесу або явле-
ня.

На цьому етапі проводиться вибір з усієї множини залежностей і зв'язей основних, визначальних той чи інший реальний процес, явище і формування гіпотез, дозволяють представити реальний, зазвичай досить складний процес, як вже відомих процесів. Моделювання передбачає деяку розумну абстракцію, що дає можливість з достатньою точністю уявити реальні фізичні, інформаційні або еко-

номічні процеси.

Для адекватного відображення суті процесу, що вивчається, або явища доводиться розробляти різні моделі. Найчастіше використовуються інформаційні та математичні моделі. В інформаційної моделі вказуються найбільш значущі характеристики об'єкта, що мають істотне значення для цього завдання. Наприклад, якщо створюється база даних таксопарку, то найбільш значущими характеристиками такого об'єкта, як автомобіль, є марка автомобіля, рік випуску, державний номер та ін.

Під математичним формулюванням завдання чи як його іноді називають математичною моделлю, мається на увазі будь-який математичний опис досліджуваного процесу чи явища у вигляді рівнянь чи нерівностей. Як рівняння можуть бути алгебраїчні та трансцендентні рівняння, системи лінійних рівнянь алгебри, диференціальні рівняння, інтегральні рівняння тощо.

До математичного опису пред'являються загалом суперечливі вимоги. З одного боку математичний опис має бути повним, з іншого боку, бажано, щоб математичні залежності були простішими.

Вибір способу рішення.

Вибір методу вирішення залежить від виду моделі, постановки задачі та можливостей наявних засобів обчислювальної техніки. Багато завдань можна вирішити різними методами та способами, тому актуальним стає вибір оптимального методу. Причому критерії оптимальності навіть однієї й тієї завдання можуть бути різними.

Наприклад, розробити базу даних таксопарку з максимальною кількістю сервісних функцій та процедур, причому керівництво таксопарку готове купити та встановити стільки комп'ютерів, скільки необхідно для успішного функціонування цієї бази.

Або через брак достатніх коштів, керівництво таксопарку згідно з встановленням тільки одного комп'ютера в планово-економічному відділі. Зрозуміло, що

бази даних у першому та другому випадках істотно відрізнятимуться за своїми функціональними можливостями та за методами, використаними розробниками для реалізації цих баз даних.

Оскільки комп'ютери оперують з числами, то як методи вирішення математичних моделей зазвичай застосовуються чисельні методи. Перетворення математичних виразів, характерне для класичної математики, не є типовим при застосуванні комп'ютерів. Хоча останніми роками з'явилися потужні програмні системи типу MAPLE, які дають змогу проводити і символьні обчислення. У той самий час чисельні методи часто дозволяють вирішувати завдання, які методами класичної математики зазвичай нерозв'язні.

Розробка алгоритму розв'язання задачі.

Характер роботи цьому етапі істотно залежить від попередніх етапів. Крім того, має значення та розмір завдання. Якщо це досить просте завдання, з яким може впоратися одна людина, то робота може йти приблизно так, як у прикладі, розглянутому в попередньому розділі (завдання про поїзди та муху).

Для середніх та великих проектів, для реалізації яких може знадобитися група або навіть цілий колектив програмістів, виникають проблеми визначення методології проектування, планування та розподілу робіт між членами групи, їх взаємодії та ін.

Важливе значення має вибір засобів проектування та розробки ПЗ. В даний час широкого поширення набули так звані RAD-системи (Rapid Application Development - швидка розробка додатків). Як приклад наведу такі системи так Microsoft Visual Studio 2008, Code Gear RAD Studio 2009, Embarcadero RAD Studio 2010, у яких є розвинені засоби для розробки ПЗ у колективі.

Складання програми (кодування).

Під цим етапом мається на увазі безпосередній запис отриманих раніше алгоритмів обраною мовою програмування. Сучасні системи програмування дозволяють значно полегшити цей процес, хоча цей етап, як і раніше, залишається одним із найбільш трудомістких. Зрозуміло, цей етап передбачає хороше знання тієї мови програмування, якою ведуться роботи.

У наступних розділах книги ми власне і займемося вивченням однієї з найпопулярніших мов програмування, якою є мова Pascal.

Налагодження та тестування програми.

Під цим розуміється пошук та виправлення помилок у програмі. Причому під налагодженням розуміється виправлення помилок безпосередньо в процесі кодування. Величезну допомогу програмісту у цій справі надає компілятор, який вказує програмісту місце виникнення помилки та характер помилки. Однак факт того, що компілятор не повідомив про помилку та програма почала працювати, ще не гарантує від відсутності помилок. Це звані логічні помилки чи помилки часу виконання. Для виявлення таких помилок розробляються система тестів – спеціальним чином підібрані контрольні приклади, котрим вирішення завдання відоме.

У великих проектах програми поділяються на версії. Альфа-версія – це перша працездатна версія програми. Бета-версія це версія або версії, які передаються замовнику для додаткового тестування в реальних умовах функціонування програми.

1.3. Приклади розробки алгоритмів

1.3.1 Розв'язання квадратного рівняння.

Знайти коріння квадратного рівняння AX2+BX+C=0, коефіцієнти A, B, C задано та вводяться з клавіатури.

З елементарної математики відома формула знаходження коренів цього рівняння:

$$X_{1,2} = \frac{-B \pm \sqrt{B^2 - 4\,AC}}{2\,A}, \tag{1.8}$$

Однак ця формула застосовна тільки для випадку дійсних коренів. Але ми вважаємо, що коефіцієнти A, B, C можуть бути довільними, тому необхідно провести аналіз задачі та визначити можливі варіанти обчислень. Аналіз завдання та визначення можливих ситуацій, що виникають у ході обчислень, є однією з найважливіших функцій програміста. Спроба запрограмувати лише формулу (1.8) може призвести до невизначеної ситуації, якщо A=0, або B2-4AC<0. Саме програміст повинен передбачити можливість виникнення таких ситуацій та явно вказати порядок обчислень у кожному конкретному випадку.

Якщо A=0, це означає, що вихідне рівняння виродилося лінійне $BX+C=0$. У цьому випадку рішенням його буде

$$X = -\frac{B}{C}, \tag{1.9.}$$

Якщо дискримінант B2-4AC<0, рівняння матиме комплексне пов'язане коріння. Кожне комплексне число можна уявити парою діючих чисел, одне з яких зображує дійсну частину, інше - уявну частину комплексного числа.

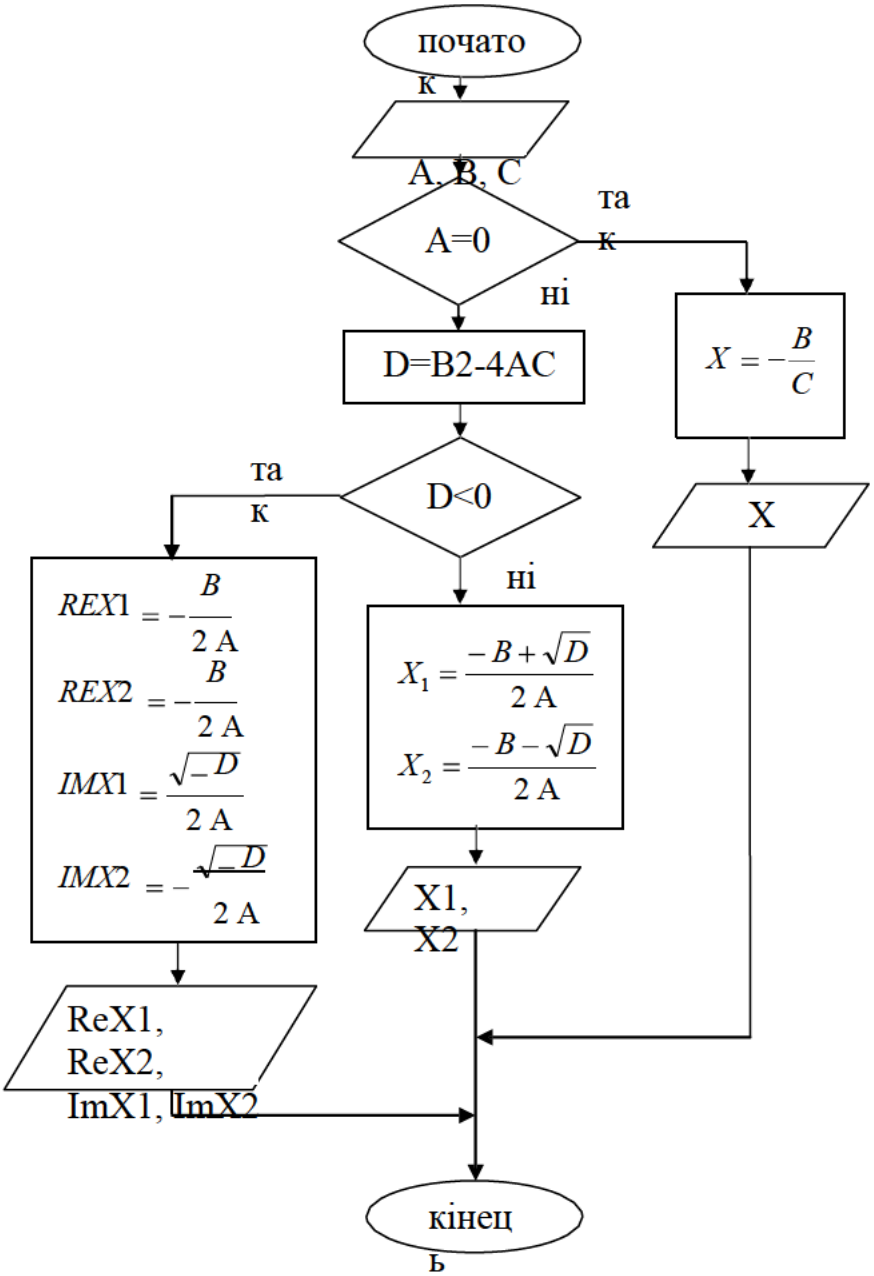
Дійсні частини обох коренів рівні.

$$\mathrm{Re}\,X_1 = \mathrm{Re}\,X_2 = -\frac{B}{2A}, \tag{1.10}$$

А уявні матимуть різні знаки, і обчислюватимуться за формулою

$$\mathrm{Im}\,X_1 = \frac{\sqrt{-(B\,2 - 4\,AC)}}{2\,A}, \qquad \mathrm{Im}\,X\,2 = -\mathrm{Im}\,X_1, \tag{1.11}$$

Виходячи з цих міркувань, неважко скласти блок-схему алгоритму обчислення коренів квадратного рівняння:



Мал. 1.9. Алгоритм обчислення коренів квадратного рівняння

1.3.2 Обчислення інтегралів

Обчислити інтеграл $\int_a^b f(x)dx$ за формулою Сімпсона з точністю $\varepsilon = 10^5$.

Формула Сімпсона, як відомо, має вигляд [1,2]:

$$\int_a^b f(x)dx \cong \frac{b-a}{n3}(y_0 + y_n + 2(y_2 + y_4 + \ldots + y_{n2}) + 4(y_1 + y_3 + \ldots + y_{n1})), \quad (1.12)$$

Досягнення необхідної точності застосуємо метод подвійного перерахунку, суть якого у наступному. Нехай n=4 – кількість точок розбиття інтервалу (a, b).

Обчислюємо інтеграл I4. Потім збільшуємо n вдвічі, (n=8) і обчислюємо I8.

Якщо $|I4-I8| \le \varepsilon$, то необхідна точність досягнуто. Як результат беремо I8. Якщо ж $|I4-I8| > \varepsilon$, то знову збільшуємо n вдвічі (n=16) обчислюємо I16, потім якщо $|I8-I16| \le \varepsilon$, То точність досягнуто. Якщо ні, то повторюємо вищевказаний процес до досягнення необхідної точності. Блок-схема алгоритму обчислення інтеграла за формулою Сімпсона методом подвійного перерахунку виглядатиме так:

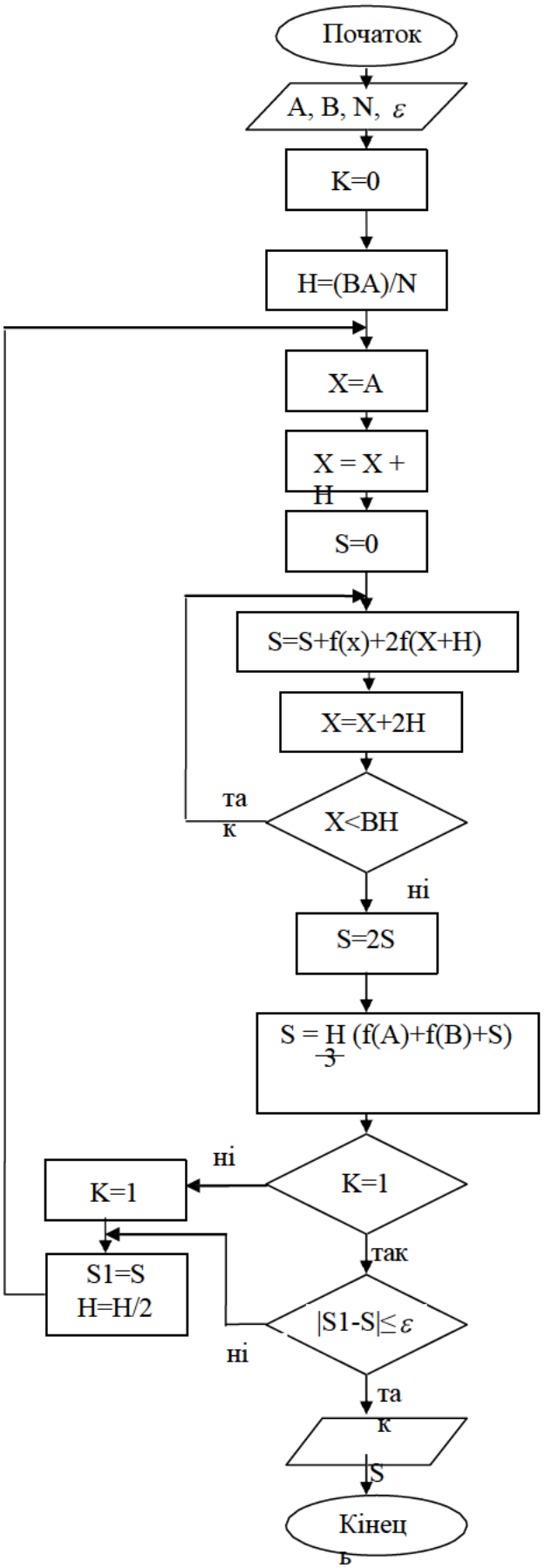


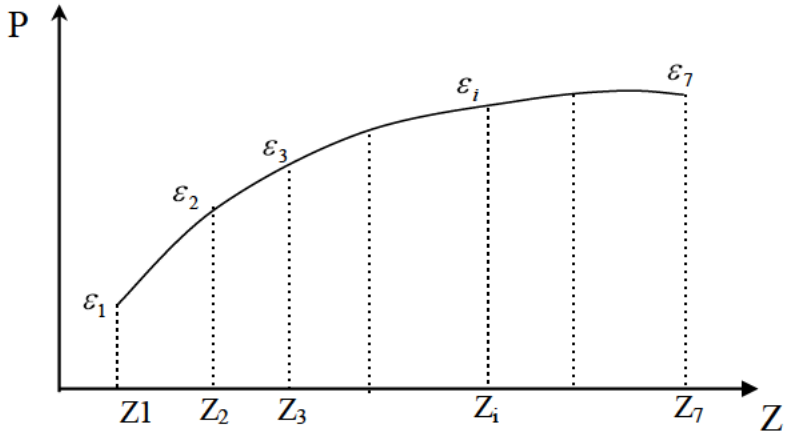
Рис.1.10. Алгоритм обчислення певного інтеграла за формулою Сімпсона

1.3.3 Обробка результатів експерименту

При вирішенні інженерних та економічних завдань часто виникає необхідність отримання математичних залежностей між різними параметрами, характерними для даної задачі. Вихідною інформацією для встановлення цих залежностей є фізичний експеримент чи економічні показники. Як у тому, і іншому випадку ми маємо або таблич-

ними даними, або точками на графіку. Нехай є залежність Pi , напів-

чена при дискретних значеннях Zi. Значення Pi отримані з експерименту з деякими похибками. Потрібно знайти залежність P= *f*(Z).



Мал. 1.11. Графік функції

Враховуючи, що P= *f*(Z) має явно виражену нелінійну залежність, запишемо рівняння кривої другого порядку.

$$P = X_0 + X_1 Z + X_2 Z^2 \qquad (1.13)$$

У цьому рівнянні X0, X1, X2 невідомі поки коефіцієнти. Для нахо-

дення цих коефіцієнтів запишемо для всіх наявних значень Pi залежність виду (1.13).

$$P_1 = X_0 + X_1 Z_1 + X_2 Z_1^2$$

$$P_2 = X_0 + X_1 Z_2 + X_2 Z_2^2$$

.....

$$P_i = X_0 + X_1 Z_i + X_2 Z_i^2 \qquad (1.14)$$

$$\cdots\cdots\cdots\cdots\cdots\cdots$$

$$P_7 = X_0 + X_1 Z_7 + X_2 Z_7^2$$

Отримано систему з 7 рівнянь із 3 невідомими. Необхідно таким методом знайти X 0 , X 1 , X 2 щоб залежність (1.13) найкращим способом описала результати, подані на графіку.

Для знаходження трьох невідомих належить вирішити систему з 7 рівнянь. Якщо ми відкинемо якісь 4 зайві рівняння, ми знайдемо значення невідомих без урахування цих відкинутих рівнянь. З іншого боку, система (1.14) може бути несумісною, тобто. при її вирішенні ми можемо не отримати тотожності і при підстановці знайдених значень невідомих рівнянь системи отримаємо різницю між лівою та правою частинами.

Позначимо ці різниці відповідно до номерів рівнянь через $\varepsilon_1, \varepsilon_2 \ldots \varepsilon_i, \ldots, \varepsilon_7$ і називатимемо їх нев'язками. Нев'язка є різницю між аналітичною залежністю і значеннями Pi , заданими в ка- Якість вихідної інформації в дискретних точках Zi.

Для того, щоб аналітична залежність найбільш повно відображала результати експерименту, мінімізуватимемо величину:

$$S = \sum_{i=1}^{7} \varepsilon_i^2 \qquad (1.15)$$

Нев'язки взяті в квадрат для того, щоб будь-яка нев'язка виходила з одним позитивним знаком, при цьому співвідношення малих і великих нев'язок збільшаться. Мінімізація S виражатиме найкраще наближення аналітичної залежності до експериментальних точок (при заданому ступені полінома). Розглянутий нами метод називається методом найменших квадратів [3].

Загальне формулювання завдання:

необхідно вирішити систему n-лінійних рівнянь із m невідомими.

$$
\begin{aligned}
&a11x1+a12x2+\ldots+a1jxj+\ldots a1mxm=b1\\
&a21x1+a22x2+\ldots+a2jxj+\ldots a2mxm=b2\\
&\\
&ai1x1+ai2x2+\ldots+aijxj+\ldots aimxm=bi\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
&an1x1+an2x2+\ldots+anjxj+\ldots anmxm=bn
\end{aligned}
\qquad (1.16)
$$

Запишемо і-е рівняння у більш компактному вигляді:

$$\sum_{j=1}^{m} a_{ij}x_j = b_i \qquad (1.17)$$

тоді $$S=\sum_{i=1}^{n}\varepsilon_i^2=\sum_{i=1}^{n}(\sum_{j=1}^{m}a_{ij}x_j-b_i)^2 \qquad (1.18)$$

Для мінімізації S візьмемо від цієї величини приватні похідні з кожної змінної xj і прирівняємо до 0.

$$\frac{\partial S}{\partial x_j}=2\sum_{i=1}^{n}(\sum_{j=1}^{m}a_{ij}x_j-b_i)a_{ij}, \qquad (1.19)$$

$\frac{\partial S}{\partial x_j}=0$, звідси:

$$\sum_{i=1}^{n}(\sum_{j=1}^{m}a_{ij}x_j-b_i)a_{ij}=0, \qquad (1.20)$$

Таких рівнянь буде стільки, скільки невідомих xj і отримаємо систему n-лінійних рівнянь алгебри з n невідомими, які вирішуються методом виключення з виділенням головного елемента.

1.3.4 Розв'язання системи лінійних рівнянь алгебри

Розглянемо систему з n рівнянь з n невідомими. Методи чисельного розв'язання систем лінійних рівнянь поділяються на дві групи: прямі (кінцеві) та ітераційні (нескінченні). Звичайно, ніякий практичний метод рішення не може бути нескінченним. Ми маємо на увазі лише те, що прямі методи можуть у принципі (з точністю до помилок довкілля) дати таке рішення, якщо воно існує, за допомогою кінцевого числа арифметичних операцій. З іншого боку, при використанні ітераційних методів, для отримання точного рішення теоретично потрібна нескінченна кількість арифметичних операцій. Отже, під час практичного дослідження ітераційних методів з'являються помилки обмеження. Не означає, що прямі методи краще, т.к. помилки округлення, які у прямих методах, грають велику роль. У деяких випадках через помилки округлення можуть бути отримані безглузді результати. Незважаючи на неминучу помилку обмеження, ітераційні методи можуть бути найзручнішими, т.к. при його використанні помилки заокруглення не накопичуються.

Розглянемо одне із прямих методів званих методом виключення (метод Гаусса).

Для ілюстрації методу розглянемо систему з 3 рівнянь із 3 невідомими:

$$\begin{aligned} a11x1+a12x2+a13x3=b1(1)\\ a21x1+a22x2+a23x3=b2(2)\\ a31x1+a32x2+a33x3=b3(3) \end{aligned} \quad (1.21)$$

Тут хоча б один з коефіцієнтів a11, a21, a31 має бути відмінний від 0, інакше ми мали справу з 3 рівняннями з двома невідомими. Нехай a11≠0, якщо це не так ми можемо переставити місцями рівняння, так щоб ко-

ефект при x1 у першому рівнянні був відмінний від 0. Перестановка рівнянь систему не змінить. Тепер введемо множник:

$$m_2 = \frac{a_{21}}{a_{11}} \tag{1.22}$$

Помножимо 1-е рівняння (1.21) на m2 і віднімемо його з 2-го рівняння (1.21). Маємо:

$$(a_{21} - ma_{2\,11})x_1 + (a_{22} - ma_{2\,12})x_2 + (a_{23} - ma_{2\,13})x_3 = b_2 - m_2 b_1 \tag{1.23}$$

Але $$a_{21} - ma_{2\,11} = a_{21} - \frac{a_{21}}{a_{11}} a_{11} = 0 \tag{1.24}$$

Позначимо

$$\begin{aligned} a_{22} &= a_{22} - ma_{2\,12} \\ a'_{23} &= a_{23} - ma_{2\,13} \\ b'_2 &= b_2 - m_2 b_1 \end{aligned} \tag{1.25}$$

Тоді 2-е рівняння (1.21) набуде вигляду:

$$a'_{22}x_2 + a'_{23}x_3 = b'_2 \tag{1.26}$$

Замінимо це рівняння (1.21) рівнянням (1.26), отримаємо систему:

$$\begin{aligned} &a11x1+a12x2+a13x3=b1 \quad (1) \\ &a'_{22}x_2 + a'_{23}x_3 = b'_2 \quad (4) \\ &a31x1+a32x2+a33x3=b3 \quad (3) \end{aligned} \tag{1.27}$$

Помножимо тепер (1) в (1.27) $m_3 = \frac{a_{31}}{a_{11}}$ і віднімемо з (3) на

$$a'_{32} = a_{32} - ma_{3\,12}$$

$$a'_{33} = a_{33} - ma_{3\,13}$$

$$b'_3 = b_3 - m_3 b_1$$

Рівняння (3) набуває вигляду:

$$d_{32}x_2 + a'_{33}x_3 = b'_3 \qquad (5)$$

І вихідна система (1.21) тепер має вигляд:

$$\begin{aligned} &a11x1+a12x2+a13x3=b1 \quad (1)\\ &d_{22}x_2 + a'_{23}x_3 = b'_2 \quad (4)\\ &d_{32}x_2 + a'_{33}x_3 = b'_3 \quad (5) \end{aligned} \qquad (1.28)$$

Ці нові рівняння еквівалентні вихідним, з тією перевагою, що *x1* не входить ні до другого, ні до третього рівняння системи.

Спробуємо тепер виключити x2 із рівнянь (4) та (5).

Якщо $a'_{22} = 0$, то ми знову переставимо місцями рівняння, так щоб $a'_{22} \neq 0$. Якщо ж $a'_{22} = 0$ та $a'_{32} = 0$, то система вироджена і або не має рішення, або має безліч рішень. Введемо новий множник $m_3' = \frac{a'_{32}}{a'_{22}}$. Помножимо його на (4) і віднімемо його з (5)

$$(a'_{32} - m'_3 a'_{22})x_2 + (a'_{33} - m'_3 a'_{23})x_3 = b'_3 - b'_2 m'_3 \qquad (1.29)$$

В силу вибору m3′

$$\begin{aligned} &d'_{32} - m'_3 a'_{22} = 0\\ &a''_{33} = a'_{33} - m'_3 a'_{23}\\ &b''_3 = b'_3 - b'_2 m'_3 \end{aligned} \qquad (1.30)$$

Рівняння (1.29) запишеться у вигляді

$$d''_{33}x_3 = b''_3 \tag{1.31}$$

Рівняння (1.29) можна замінити рівнянням (1.31).

Система (1.21) набуває вигляду:

$$\begin{aligned} &a11x1+a12x2+a13x3=b1 \quad (8)\\ &d'_{22}x_2 + a'_{23}x_3 = b'_2 \quad (9)\\ &a''_{33}x_3 = b''_3 \quad (10) \end{aligned} \tag{1.32}$$

Рішення цієї системи цілком очевидне.

$$\begin{aligned} x_3 &= \frac{b''_3}{a''_{33}}\\ x_2 &= \frac{b_2 - a'_{23}x_3}{a'_{22}}\\ x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \end{aligned} \tag{1.33}$$

Для чого ми завжди переставляємо рівняння таким чином, щоб a_{11}, a'_{22}, a''_{33} були не рівні 0? Щоб не було поділу на 0!

Приклад:

$$\begin{cases} x + y + z = \\ +42x + 3y = \\ -z - 9x - y2 \\ z \end{cases} \tag{1.34}$$

Помножимо перше рівняння (1.34) на 2 і віднімемо з 2-го рівняння. Потім перше рівняння помножимо на 1 і віднімемо з 3-го. Отримаємо систему, екві-

стрічкову (1.34).

$$\begin{cases} x + y + z = 4 \\ y - z = 1 \\ -2\,y - 2z = -6 \end{cases} \qquad (1.35)$$

Помноживши друге рівняння (1.35) на (-2) і віднімаючи його з 3-го рівняння отримуємо

$$\begin{cases} x + y + z = 4 \\ y - z = 1 \\ -4z = -4 \end{cases} \qquad (1.36)$$

Звідси рішенням цієї системи буде:

$x = 1$

$y = 2$

$z = 1$

Узагальним цей метод у разі системи з n рівнянь з n невідомими

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned} \qquad (1.37)$$

Передбачається в силу розташування рівнянь $a_{11} \neq 0$. Введемо *n-1* множників:

$$m_i = \frac{a_{i1}}{a_{11}}, \qquad i = 1, 2, 3, \ldots n \qquad (1.38)$$

І віднімемо з кожного і-го рівняння перше, помножене на mi. Позначимо

$$a'_{ij} = a_{ij} - m_i a_{1j},$$

$$b_i' = b_i - m_i b_1, \qquad (1.39)$$

$$i=2, 3, \ldots, n, j=1, 2, \ldots, n$$

Для всіх рівнянь, починаючи з 2-го $a_{i1}=0$, i=2, 3, … n

Отримаємо систему

$$\begin{aligned} & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = \\ & 0 + a_{22}x_2 + \ldots + b_1 \\ & \qquad a'_{2n}x_n = b'_2 \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ & 0 + a_{n2}x_2 + \ldots + a'_{mn}x_n = b'_n \end{aligned} \qquad (1.40)$$

Продовжуючи в такий спосіб, ми можемо виключити x2 з останніх n-2 рівнянь, x3 з останніх n-3 рівнянь тощо. На деякому k-му етапі ми виключимо xk за допомогою множників.

$$m_i^{(k-1)} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \quad i=k+1,\ldots n \qquad (1.41)$$

Причому

$$a_{kk}^{(k-1)} \neq 0,$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - m_i^{(k-1)} a_{kj}^{(k-1)}$$

$$b_i^{(k)} = b_i^{(k-1)} - m_i^{(k-1)} b_k^{(k-1)}$$

де $i = k+1, k+2, \ldots, n; \quad j = k, \ldots, n; \quad k = 1, \ldots, n-1$

Остаточно трикутна система рівнянь записується в такий спосіб.

$$\begin{aligned} & a_{11}x_1 + a_{12}x_2 + \ldots \quad a_{1n}x_n \qquad a_{2n}x_n \\ & \qquad a'_{22}x_2 + \ldots + \quad ' \qquad = b'_1 \end{aligned}$$

$$+ \quad =$$

(1.42)

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$a_{mm}^{(n-1)} x_n = b_n^{(n-1)}$$

Зворотна підстановка для знаходження значень невідомих визначається формулами:

$$x_n = \frac{b_n^{(n-1)}}{a_{mm}^{(n-1)}}$$

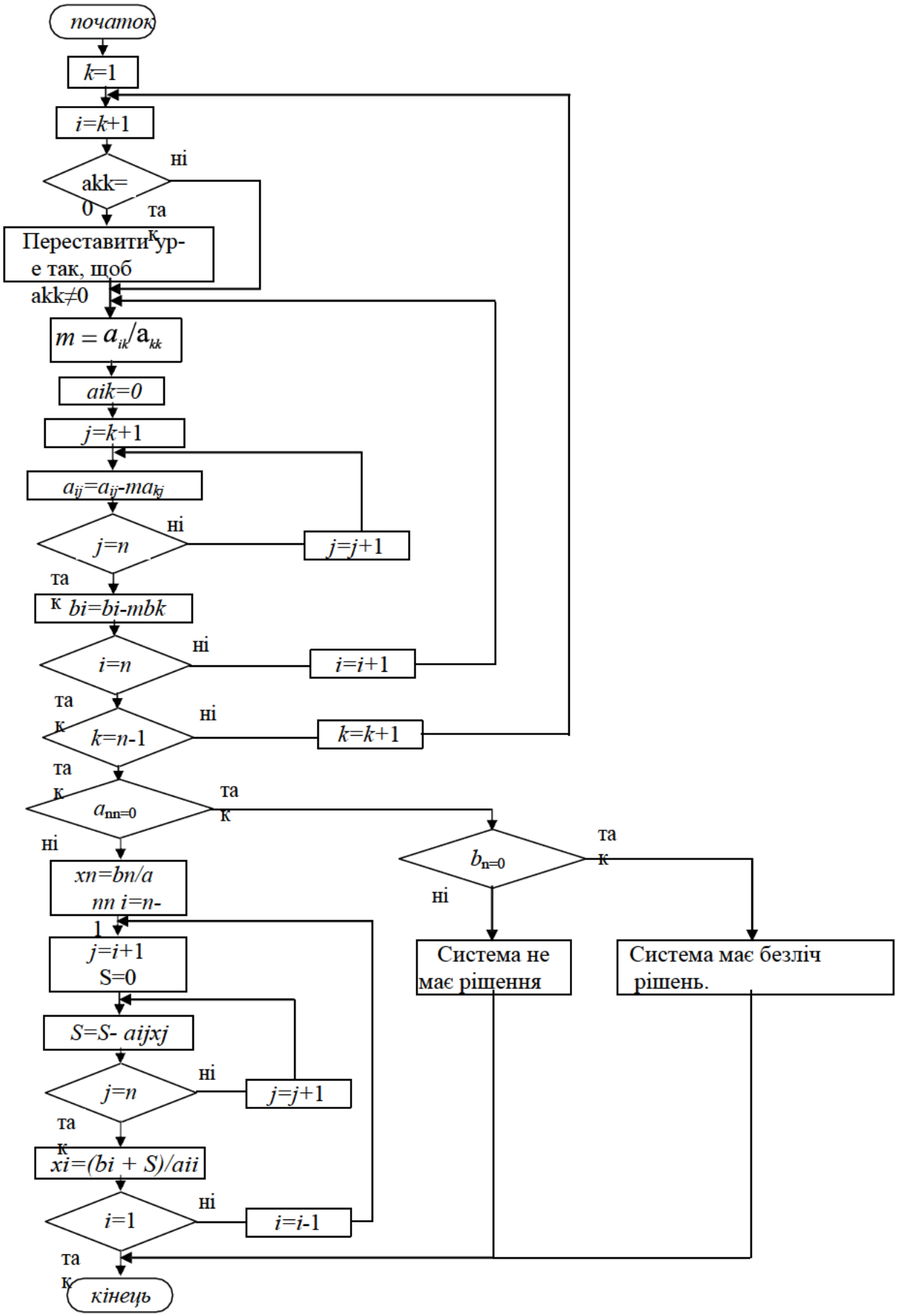
$$x_{n-1} = \frac{b_{n-1}^{(n-2)} - a_{n-1,n}^{(n-2)} \cdot x_n}{a_{n-1,n-1}^{(n-2)}} \qquad (1.43)$$

$$x_j = \frac{b_j^{(j-1)} - a_{j,n}^{(j-1)} \cdot x_j - \ldots - a_{j,j+1}^{(j-1)} \cdot x_{j+1}}{a_{jj}^{(j-1)}}, \quad j = n-2,\ n-3,\ \ldots,\ 1$$

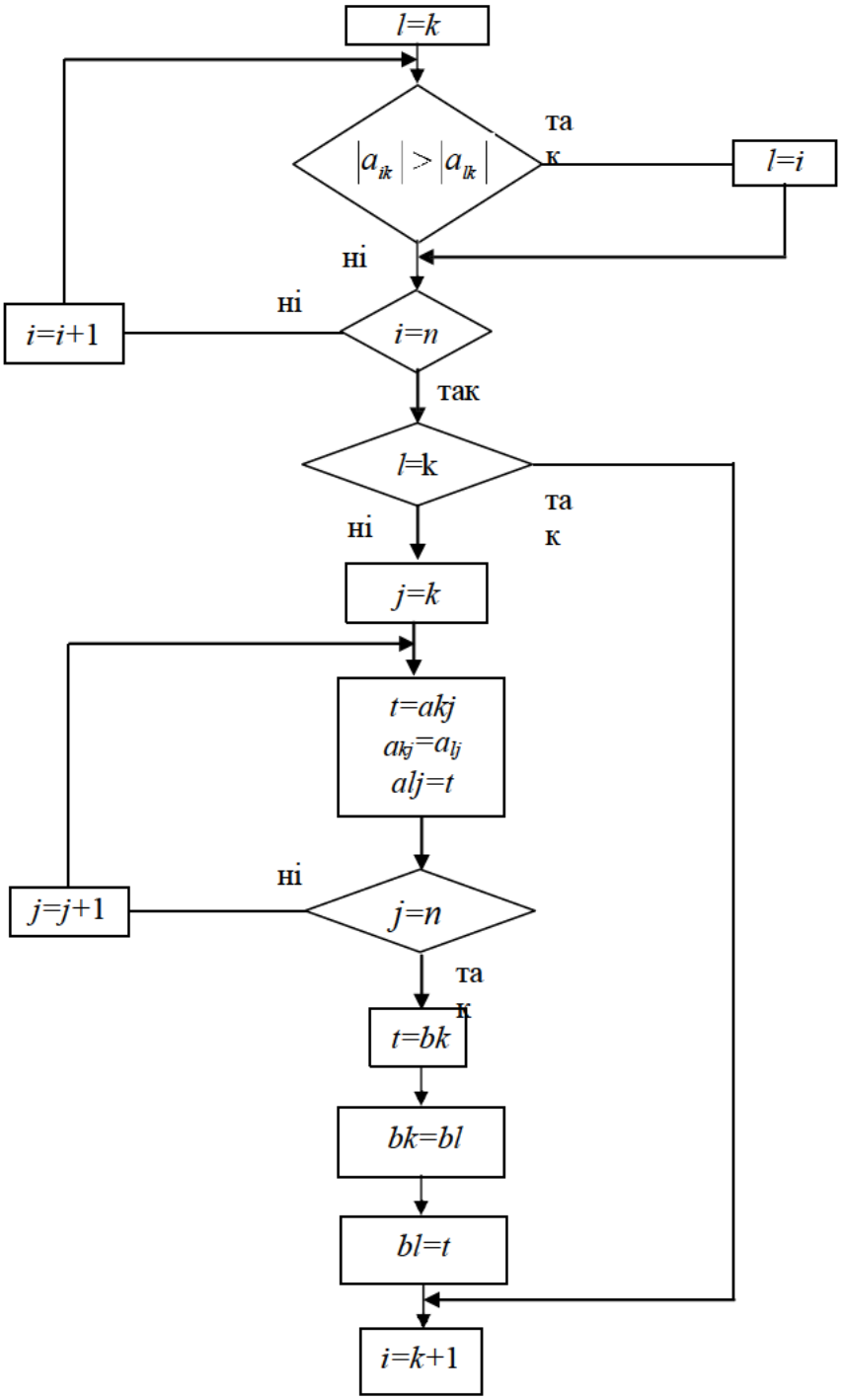
Блок-схема алгоритму показано на рис. 1.12.

Тут, щоб не захаращувати блок-схему, ми припустили, що коефіцієнти системи вже введені. У цій блок-схемі неясно лише одне – що означає переставити рівняння, як це зробити?

Виявляється, що якщо переставити рівняння таким чином, щоб коефіцієнт при xk був найбільшим, помилки округлення будуть мінімальними. Цей коефіцієнт називається основним елементом. І перестановка рівнянь із вибором головного елемента називається методом головних елементів, рис. 1.13.



Мал. 1.12. Алгоритм Гауса



Мал. 1.13. Алгоритм перестановки рівнянь

Глава 2 Введення у мову програмування Pascal

2.1. Основні елементи мови

Мова Pascal (Паскаль), винайдена на початку 70-х років 20-го століття Н. Віртом і названа на честь французького математика та філософа Блеза Паскаля, є однією з найпоширеніших мов програмування. Від інших мов він вигідно відрізняється можливістю зрозуміліше і логічне записувати програми.

Програма мовою Паскаль складається з двох частин: опис дій, які мають бути виконані та опис даних, над якими вони виконуються. У тексті програми опис даних передує опису дій. У цьому виражається загальне правило мови – кожен об'єкт, що зустрічається в програмі, повинен бути попередньо описаний.

Опис даних складається з опису змінних. Операторами називають дії над даними. Загалом будь-яка Паскаль – програма має вигляд:

```
заголовок програми
розділ опису змінних розділ
операторів
```

Заголовок програми має вигляд:

```
program ім'я програми;
```

Тут слово "program" - це так зване ключове (або службове або ще кажуть зарезервоване) слово. Воно має записуватися саме

так (без лапок), а не інакше. Допускається використовувати як малі, так і великі літери. Записи PROGRAM, Program, ProgRam – дозволені та означають те саме.

Так розпочинаються всі програми, написані мовою Паскаль. Тут нічого розуміти, просто так прийнято розробником мови.

В принципі допускається не використовувати заголовок program, але краще все-таки починати програму саме із заголовка!

Ім'я програми – це будь-яка послідовність букв, цифр та деяких знаків. Такі послідовності називаються ідентифікаторами. Ідентифікатор складається з 1-127 символів – букв, цифр або знаків підкреслення, при цьому першим має бути буква або знак підкреслення. Ідентифікатор не повинен збігатися з жодним з ключових слів. В ідентифікаторі не повинно бути (.) – точки, (,) – коми, самих дужок ( ), а також пропусків та знаків операцій.

Приклади правильних ідентифікаторів:

```
X3
Summa
VOLVO
Select_screen_color
```

Приклади неправильних ідентифікаторів:

```
3x починається з цифри
Sum.ma всередині ідентифікатора є точка
VOL VO є пробіл
```

2.1.1 Змінні. Стандартні типи.

Кожна змінна має ім'я та тип. Ім'я змінної – це довільний ідентифікатор. Надалі говоритимемо "змінна x", замість "змінна з ім'ям x".

Тип змінної визначає безліч її можливих значень, набір допустимих операцій над змінною та розмір пам'яті, що займається.

У Паскалі існують такі стандартні типи змінних:

`integer (цілий), real (речовий), boolean (логічний), char`

(символьний), string (рядковий).

Значеннями змінних цілого типу є цілі (і лише!) Числа. Приклади цілих чисел:

25 +150 -200 10000

Операції над цілими числами такі:

`+ (додавання), - (віднімання), * (множення), div (розподіл націло), mod (залишок від розподілу двох цілих чисел).`

Значеннями змінних речовинного типу є речові числа. Визначено такі операції над речовими числами:

`+ (додавання), - (віднімання), * (множення), / (розподіл).`

Запис дійсних чисел схожий на загальноприйнятий, тільки замість коми використовується точка і замість ступеня 10 використовується буква E.

Приклад:

Таблиця 2.1

Загальноприйнята	на Паскалі
5,30	5.30
-1,0	-1.0
41000	41000 або 4.1E4
-0,73·10-2	-0.73E-2

Значення змінних логічного типу є true (істина), false (брехня). Визначено операції: not (не), and (і), or (або), xor (що виключає або).

Значення змінних символьного типу – одиночні символи. Для представлення символів у пам'яті комп'ютера використовуються спеціальні таблиці кодування, про які йтиметься пізніше.

Значення змінних рядкового типу – ланцюжок символів. При записі констант символьного та рядкового типу використовують одиночні лапки.

приклад.

```
'A' - це символ A
'Це ланцюжок символів'
```

2.1.2 Операції відносини

Існують такі операції відносини:

= одно, <> не дорівнює, < менше, > більше, <= менше або дорівнює,

>= більше чи одно.

Результатом цих операцій є логічні значення true чи false.

2.1.3 Розділ змінних змін

Цей розділ має вигляд:

```
var опис 1; опис 2; …; опис n;
var – ключове слово (від англійської variable
– змінна) опис має вигляд:
```

змінна 1, змінна 2, … змінна m: тип; змінна 1, змінна

2, …, змінна до: тип;

……змі

нна 1, змінна 2, … змінна s: тип;

тип – одне з ключових слів: integer, real, boolean, char, string

Приклад розділу описів:

```
var a, b, c, x, y: real;
    i, j, k, m, n: integer;
    FLAG: boolean;
    symbol: char;
```

2.1.4 Вирази. порядок виконання операцій.

Сукупність змінних та констант, з'єднаних знаками операцій та дужками, називається виразом.

приклад.

```
(a+b)/c
((n+q1)*dx+(i+j)*dy)/(x1-2)*(y1-2)
```

Правила виконання операцій у Паскалі:

1. Множення та розподіл виконуються раніше, ніж додавання та віднімання. Говорять також, що множення та поділ мають більш високий пріоритет, ніж додавання та віднімання.
2. Операції, що мають однаковий пріоритет, виконуються зліва направо.

Множення і поділ мають однаковий пріоритет, додавання і віднімання. ня мають також однаковий пріоритет.

Виходячи з цих правил вираз

```
4/8*5
```

буде обчислюватися таким чином:

Спочатку буде обчислено 4/8 (=0.5), а потім результат буде помножено на 5. Вийде 0.5*5=2.5

Будь-яке відхилення цих правил має регламентуватися дужками, т.к. дії над змінними стоять у дужках виконуються насамперед.

2.1.5 Константи

У Паскалі є можливість присвоїти константі ім'я, при цьому в наступному тексті програми усюди замість цієї константи можна використовувати її ім'я. Усі визначення констант перераховуються у спеціальному розділі – розділі опису констант, що має вигляд:

```
const
    ім'я 1 = значення 1;
    ім'я 2 = значення 2;
    ……ім
    'я n = значення n;
```

приклад.

```
const
    r = 1.87E+5;
    g = 981E-2;
    атмосфера = 760;
    pi = 3.14159;
```

Даючи імена константам, ми робимо програму зрозумілішою. Запис `2*pi*r` набагато зрозуміліше та інформативніше, ніж запис

```
2*3.14159*1.87E+5
```

Крім того, при внесенні змін до програми нам буде достатньо змінити лише значення константи. Слід пам'ятати, що пам'ять для констант не приділяється. Компілятор вставляє їх значення у потрібні місця у двійковий код програми.

2.1.6 Коментарі у програмі

Для того, щоб програмістам було легше читати та розбиратися у програмах, у мові передбачено кошти для коментування фрагментів програмного коду. Коментар називається деякий пояснювальний текст звичайною людською мовою, який пояснює ті чи інші дії програміста.

Коментарі бувають однорядкові та багаторядкові. Однорядковий коментар починається із символів // та розміщується лише в одному рядку. Наприклад:

```
// змінна і використовується як індекс
// В операторах циклу
i:= 0; // Ініціалізація змінної
```

Як ми бачимо, коментар можна розташовувати в одному рядку з операром!

Багаторядковий коментар, як випливає з назви, дозволяє розмічати коментарі в кількох рядках. Багаторядковий коментар починається з символів (* і закінчується символами *). В якості обмежувачів коментаря використовуються також фігурні дужки, що відкривають і закривають {}. Наприклад:

```
(* змінна і використовується як індекс
 в операторах циклу *)
```

```
i:= 0; (* Ініціалізація змінної *)
```

або

```
{змінна i використовується як індекс в
 операторах циклу}
i:= 0; {Ініціалізація змінної}
```

Програмісти найчастіше використовують однорядковий коментар і багатостроковий коментар з фігурними дужками.

2.1.7 Оператори

Основна частина програми на Паскалі – розділ операторів. Він починається ключовим словом begin і закінчується ключовим словом end, за яким слідує крапка. Оператори відокремлюються один від одного крапкою з комою (;). Розглянемо основні оператори:

2.1.7.1. Оператор присвоєння

Елементарна дія над змінною – зміна її значення. Для цього застосовується оператор присвоєння, що має вигляд:

ім'я змінної: = вираз;

У ньому змінна та вираз мають бути одного типу. приклад.

Нехай x – змінна цілого типу. Запишемо наступний оператор присвоєння:

```
x: = x + 1;
```

У лівій частині оператора x позначає змінну, а в правій частині – число, що є поточним значенням. Виконання цього оператора призводить до збільшення значення змінної на одиницю.

2.1.7.2. Оператори введення/виводу

Під час виконання програми вона обмінюється інформацією із "зовнішнім світом". Наприклад, вона може видавати інформацію на екран або отримувати інформацію з клавіатури. Для цього використовуються оператори введення та виведення.

Оператор виведення має вигляд:

```
write (вираз);
```

або

```
writeln (вираз);
```

В результаті виконання цього оператора значення відповідного виразу буде виведено на екран. Вираз може бути будь-яким із зазначених вище типів.

приклад.

```
write(2 + 2); буде виведено на екран 4
write(x = y); буде виведено true або false залежно від значень x, y
```

Оператор writeln відрізняється від оператора write тим, що виведе значення виразу з початку нового рядка, а оператор write з тієї позиції рядка, де знаходиться курсор.

В операторі виводу можна вказувати кілька виразів, розділяючи їх комами, а також будь-який текст, укладений в одинарні лапки.

Нехай значення A = 5. Тоді під час виконання оператора

```
writeln('Значення A=', A);
буде виведено на екран з
нового рядка Значення A=5
```

Оператор введення має вигляд:

```
read(ім'я змінної 1, ім'я змінної 2, ..., ім'я змінної n);
readln(ім'я змінної 1, ім'я змінної 2, ..., ім'я змінної n);
```

У результаті виконання змінної (змінним) присвоюється зчитане з клавіатури значення. Значення, що вводиться, повинно записуватися при введенні так, як описана змінна, тобто. якщо, наприклад, змінна цілого типу, то число, що вводиться, повинно бути цілим. При цьому якщо використовується read, то значення вводиться в те місце на екрані, де в даний момент знаходиться курсор, якщо використовується readln, то з нового рядка і з першої колонки екрана. Якщо в одному операторі readln або read вводяться значення кількох змінних, то при виконанні програми значення, що водяться з клавіатури, можна розділяти пробілом або символом табуляції (клавіша Tab). Після введення значення останньої змінної необхідно натиснути клавішу Enter.

Насправді краще використовувати оператор readln т.к. по-перше, курсор буде розміщуватися завжди на початку рядка, що дозволить користувачеві легше орієнтуватися при введенні великої кількості значень. По-друге, після закінчення введення буфер введення з клавіатури повністю очищається. Справа в тому, що всі символи, що вводяться з клавіатури, спочатку накопичуються в спеціальній тимчасовій області пам'яті – буфері і лише після натискання клавіші Enter присвоюються відповідним змінним. При використанні оператора read у буфері залишається код клавіші Enter. Це може призвести до неправильної роботи наступного оператора введення.

Ви, звичайно, повинні розуміти, що під час введення чисел з клавіатури вони вводяться у вигляді рядка символів. Після натискання клавіші Enter вони переводять-

ся у внутрішнє уявлення числа.

2.1.7.3. Оператори інкременту та декременту

Крім оператора присвоювання існують оператор інкременту та декременту, які також дозволяють змінювати значення змінних цілого типу.

Оператор inc(x, n) збільшує значення змінної x цілого типу *n*. Параметр n можна опустити, тоді значення x збільшиться на одиницю. приклад.

```
inc(x, 10);
```

збільшить значення x на 10, а оператор

```
inc(x);
```

збільшить значення x на 1. Записи

```
x: = x + 10;
```

і

```
inc(x, 10);
```

абсолютно ідентичні за своїм результатом.

Оператор dec(x, n) зменшує значення змінної x на n, а оператор `dec(x)зменшує x на одиницю.`

Програмісти найчастіше використовують короткі форми операторів инкременту та декременту для збільшення або зменшення значень цілочислових змінних на одиницю, тобто. можуть використовувати

```
inc(i) замість i:= i + 1 і dec(i) замість i:= i - 1
```

2.1.8 Середа розробки Lazarus

Вже цих знань нам достатньо, щоб написати найпростішу програму. Для того, щоб писати та виконувати програми, нам знадобиться компілятор та середовище розробки. Існує досить багато компіляторів для мови Pascal. Ми з вами використовуватимемо компілятор Free Pascal

*Compiler*версії 2.2.4.

Free Pascal Compiler(часто застосовується скорочення FPC) це вільно розповсюджуваний компілятор мови Паскаль з відкритими вихідними кодами, що поширюється на умовах GNU General Public License (GNU GPL). Він сумісний з Borland Pascal 7 та Object Pascal – Delphi, але при цьому має ряд додаткових можливостей, наприклад, підтримує перевантаження операторів. FPC - кросплатформовий інструмент, що підтримує величезну кількість платформ. Серед них – AmigaOS, DOS, Linux, *BSD, OS/2, MacOS(X) та Win32.

Free Pascal підтримує компіляцію в кількох режимах, що забезпечують сумісність із різними діалектами та реалізаціями мови:

- TP — режим сумісності з Turbo Pascal: сумісність практично повна, за винятком кількох моментів, пов'язаних з тим, що FPC компілює програми для захищеного режиму процесора, де неможливе пряме звернення до пам'яті, портів тощо.
- FPC — власний діалект: відповідає попередньому, розширеному додатковим можливостям, таким як, наприклад, перевантаження операторів.
- DELPHI – режим сумісності з Delphi: включає підтримку класів та інтерфейсів.
- OBJFPC - поєднує об'єктно-орієнтовані можливості Delphi та власні розширення мови.
- MACPAS — це режим сумісності з Mac Pascal.
- GNU — це режим часткової сумісності з GNU Pascal.

Free Pascal Compiler має своє власне інтегроване середовище розробки. Застосовується також абревіатура IDE (Integrated Development Environment). Середовище має текстовий інтерфейс дуже схожий на ін-

Терфейс Turbo Pascal 7.0.

Але часи змінилися! Текстові інтерфейси практично повністю витіснені так званими графічними інтерфейсами, працювати в яких значно зручніше та приємніше.

У 1999 р. троє людей - Cliff Baeseman, Shane Miller і Michael A. Hess. зробили спробу написати безкоштовне графічне середовище для безкоштовного компілятора FPC. Проект одержує назву Lazarus. На сьогоднішній день слід визнати, що ідея виявилася дуже плідною тому, що середовище існує і розвивається й досі.

Lazarus є безкоштовним інструментом розробки з відкритим кодом. IDE Lazarus є середовищем з графічним інтерфейсом для швидкої розробки програм, аналогічне Delphi, і базується на оригінальній кросплатформовій бібліотеці візуальних компонентів LCL (Lazarus Component Library), сумісних з VCL Delphi. До складу IDE входять і не візуальні компоненти. У принципі, такого набору достатньо для створення програм з графічним інтерфейсом та додатків, що працюють з базами даних та Інтернетом.

У середовищі Lazarus використовують власний формат управління пакетами та свої файли проектів.

Lazarus це стабільне багате можливостями середовище розробки для створення самостійних графічних та консольних додатків. В даний час вона працює на Linux, FreeBSD і Windows і надає редактор коду, що настроюється, і візуальне середовище створення форм разом з менеджером пакетів, відладчиком і графічним інтерфейсом, повністю інтегрованим з компілятором FreePascal.

Чому для цієї книги було обрано саме цей компілятор та його середовище швидкої розробки? Тому що вони безкоштовні та розповсюджуються за вільною ліцензією GNU GPL. Крім того, компілятор Free Pascal дозволяє створювати кросплатформні програми, тобто. програми, які можуть ви-

повнятися на різних платформах. Тому в цій книзі виклад ведеться стосовно і Linux, і Windows.

Розглянемо основні елементи середовища розробки Lazarus. Якщо ви ще не встановили його, попередньо встановіть. Дистрибутиви знаходяться на DVD диску, що додається окремо для Linux і Windows.

Також ви можете безкоштовно скачати свіжі дистрибутиви за адресоюhttp://sourceforge.net/projects/lazarus/files/

Слід зазначити, що у тих, хто має Alt Linux Master School - Lazarus вже має бути автоматично встановлений.

Запускати Lazarus можна кількома способами. Розповім про деяких, які часто використовуються.

У Linux у Головному меню має бути пункт Lazarus. Залежно від вашого дистрибутива він може перебувати або в меню

`Розробка->Середовища розробки->Lazarus (Mandriva)`

`або Освіта->Розробка->Lazarus (Alt Linux) або`

`Програмування->Lazarus (Ubuntu).`

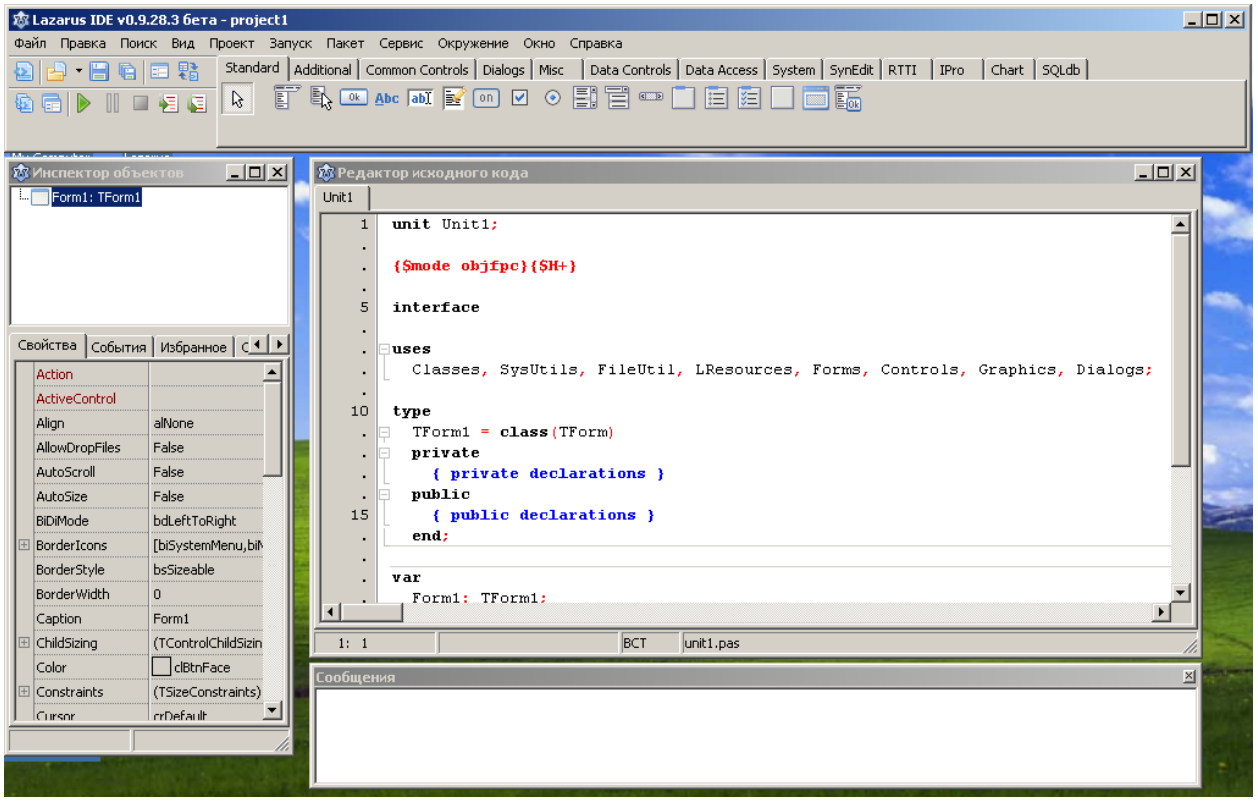
Найпростіше цей ярлик перетягнути на робочий стіл. Крім того, можна перейти в каталог установки Lazarus (частіше всього це каталог `/usr/lib/lazarus) і двічі клацнути на ім'я файлу lazarus або`

`startlazarus, якщо ви використовуєте файловий менеджер або дати команду`

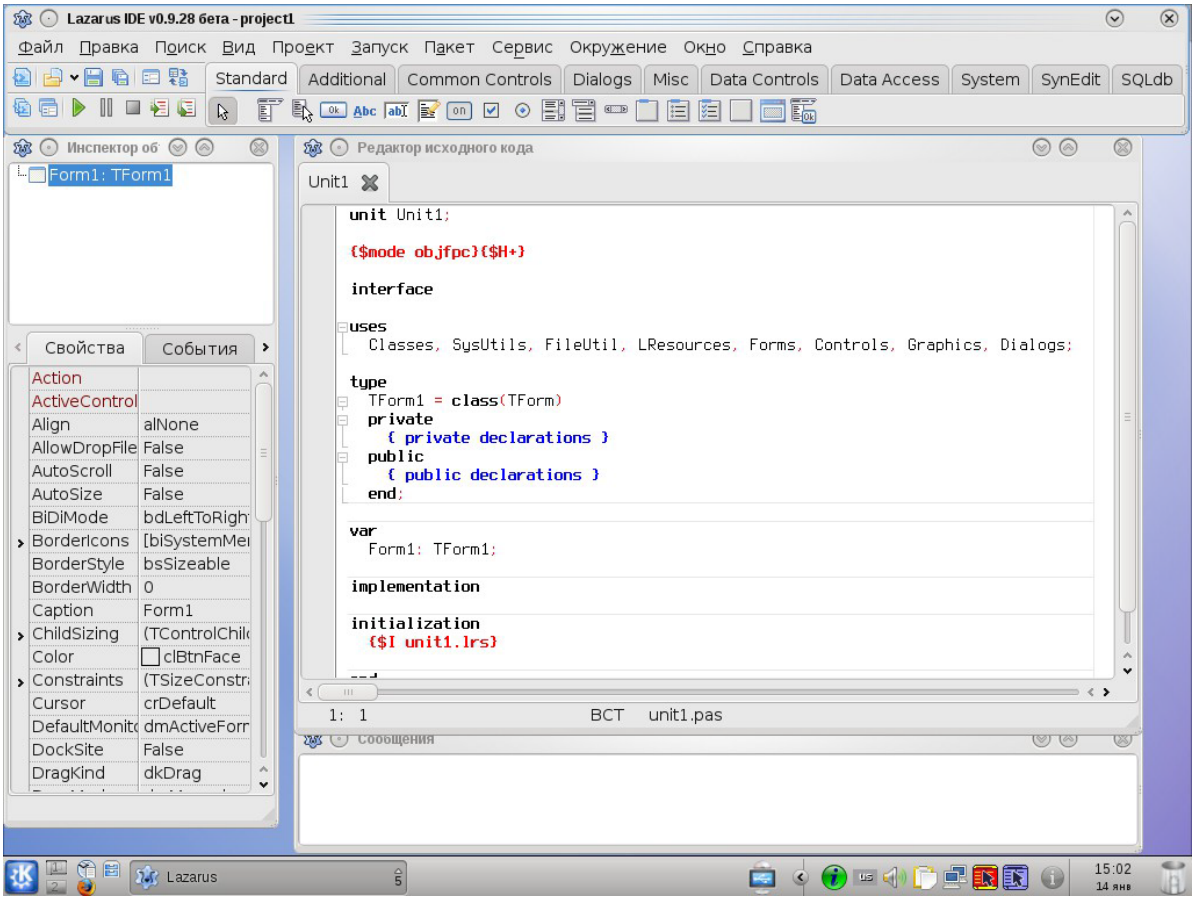
`./lazarus (./startlazarus), якщо ви використовуєте консоль.`

У Windows можна встановити параметр "Створити піктограму на робочому столі" під час інсталяції. Якщо ви цього не зробили, то можете запускати з меню Пуск->Програми->Lazarus. Ви також можете скопіювати цей ярлик на робочий стіл. Нарешті, можна перейти в папку установки (найчастіше C:\lazarus) і двічі клацнути на ім'я файлу lazarus.exe або startlazarus.exe.

Отже, IDE Lazarus має вигляд, показаний на рис. 2.1, 2.2.



Мал. 2.1 Вигляд IDE Lazarus у Windows

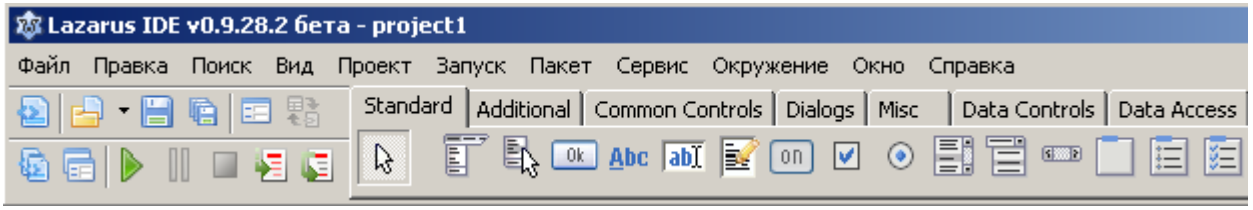


Мал. 2.2 Вигляд IDE Lazarus у Linux, робочий стіл KDE

Як бачите, IDE Lazarus виглядає однаково в обох операційних системах, тільки колірне оформлення вікон трохи відрізняється.

Середовище Lazarus складається з кількох, взагалі кажучи, не пов'язаних вікон.

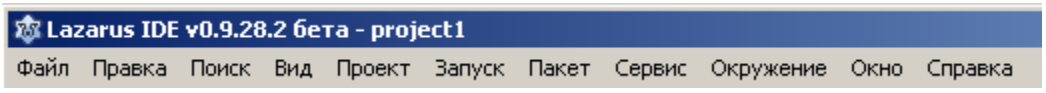
1. Головне вікно, Мал. 2.3.



Мал. 2.3. Головне вікно IDE Lazarus

За допомогою цього вікна можна керувати процесом розробки програми. У ньому передбачені команди управління файлами, компіляцією, редагуванням, вікнами тощо. Вікно розбите на три функціональні блоки:

- Головне меню. У ньому розташовані команди управління файлами, команди управління компіляцією та властивостями всієї програми, команди управління вікнами та налаштуваннями середовища та багато іншого. Меню знаходиться у верхній частині основного вікна.



Мал. 2.4. Головне меню

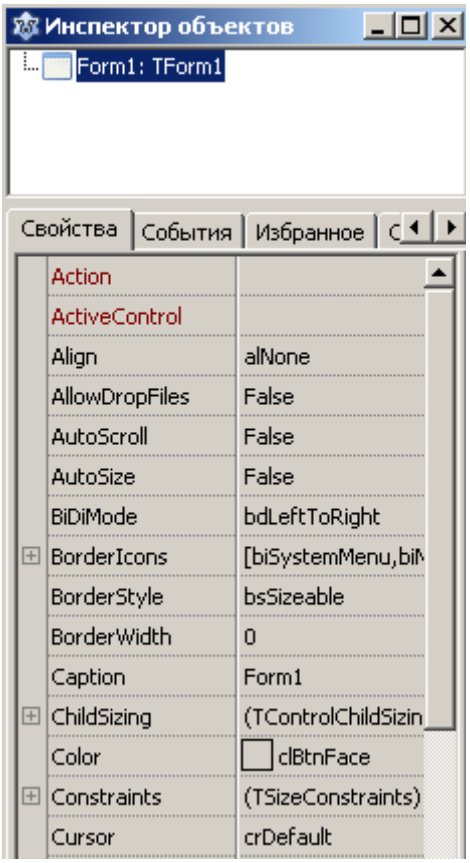
- Панель інструментів. Панель інструментів надає швидкий доступ до основних команд головного меню. Вона розташована у лівій частині головного вікна, під головним меню.

Мал. 2.5. Панель інструментів

- **Палітра компонентів.**Надає доступ до основних компонентів середовища розробки, наприклад поле введення, напис, меню, кнопка і т.п.

Мал. 2.6. Палітра компонентів

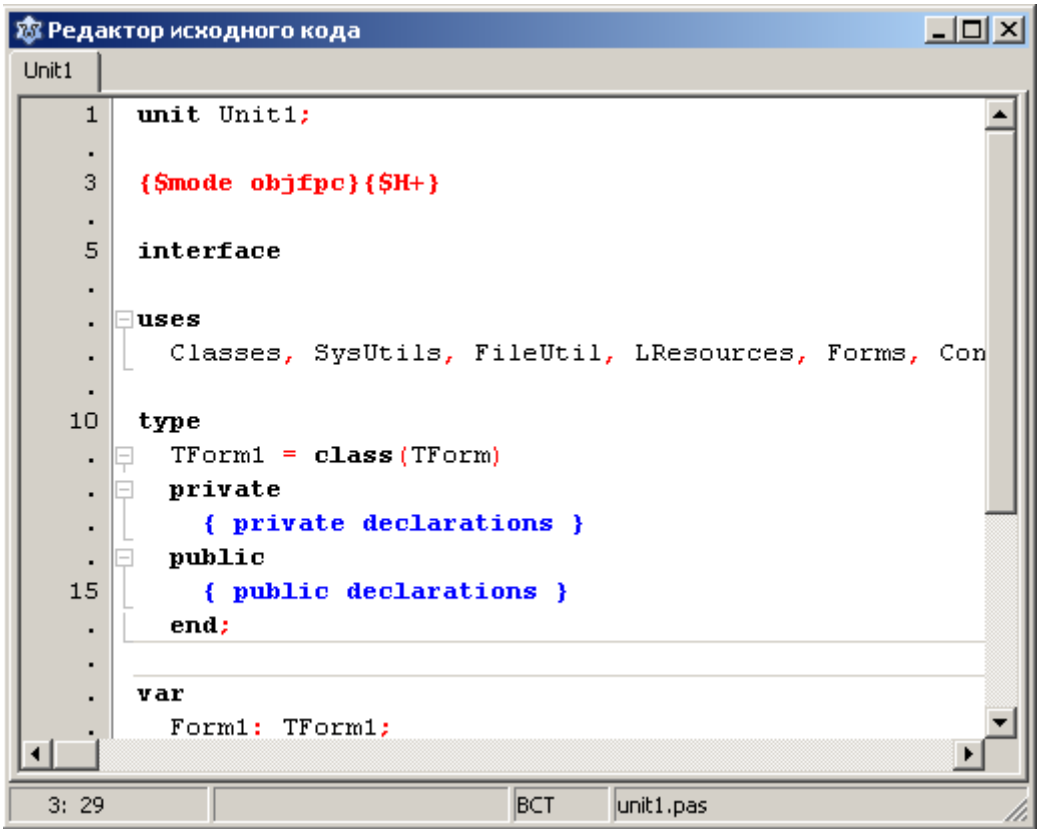
2. **Інспектор об'єктів**, Мал. 2.7.



Мал. 2.7. Інспектор об'єктів

У верхній частині вікна відображається ієрархія об'єктів, а знизу розташовані три вкладки: "Властивості", "Події", "Вибране". Призначення інспектора об'єкта – це перегляд усіх властивостей та методів об'єктів. На вкладці "Властивості" перераховуються всі властивості вибраного об'єкта. На вкладці "Події" перераховуються всі події об'єкта. На вкладці "Вибране" вибрані властивості та методи. Докладніше про це буде сказано у розділі 6.

3. Редактор вихідного коду Lazarus

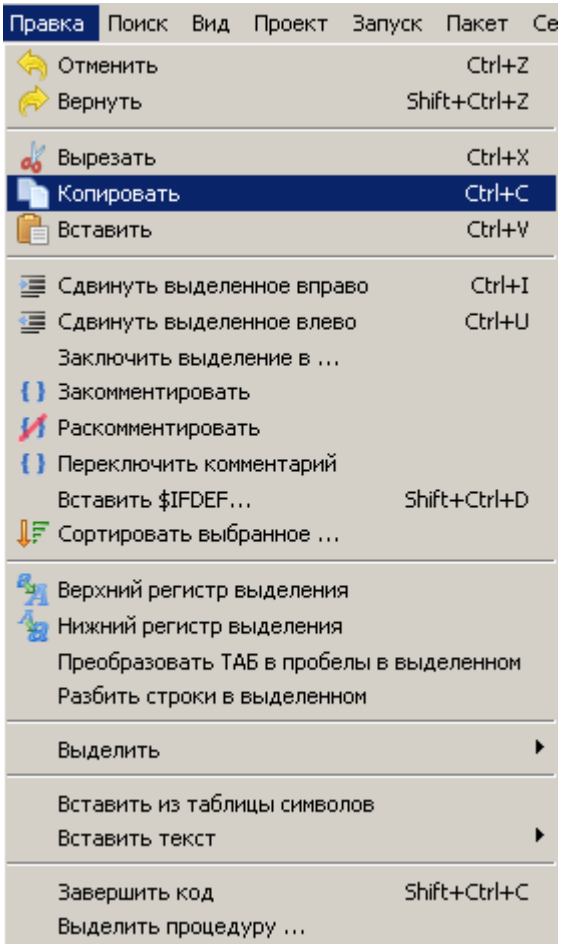


Мал. 2.8. Редактор вихідного коду

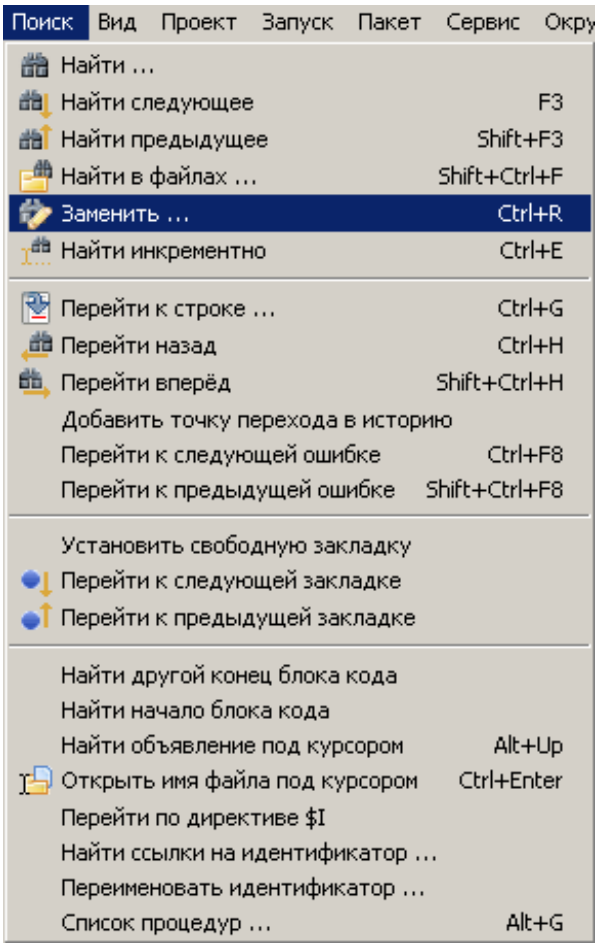
Саме в цьому вікні ми набиратимемо тексти своїх програм. Багато функцій та можливості цього редактора збігаються з можливостями звичайних текстових редакторів, наприклад Блокнота. Текст у редакторі можна виділяти, копіювати, вирізати, вставляти. Крім того, в редакторі можна здійснювати пошук заданого фрагмента тексту, виконувати вставку та заміну. Але, звичайно, цей редактор вихідних текстів Lazarus має ще ряд додаткових можливостей для комфортної роботи стосовно розробки програм. Основна перевага редактора полягає в тому, що він має можливості підсвічування синтаксису, причому не тільки Pascal, але й інших мов, а також інших зручностей. Зокрема, виділений фрагмент тексту можна зрушувати праворуч або ліворуч на кількість позицій, вказаних у налаштуваннях Оточення ->Параметри…->Редактор -> Загальні -> Відступ блоку, що дуже зручно для форматування з метою структурування коду. Виділено-

ний фрагмент можна закоментувати або розкоментувати, перевести у верхній або нижній регістр і т.д.

Усі можливі операції у редакторі зібрані у меню Правка та Пошук головного меню Lazarus, рис. 2.9, 2.10.



Мал. 2.9. Меню "Довідка"



Мал. 2.10. Меню "Пошук"

4. Вікно повідомлень

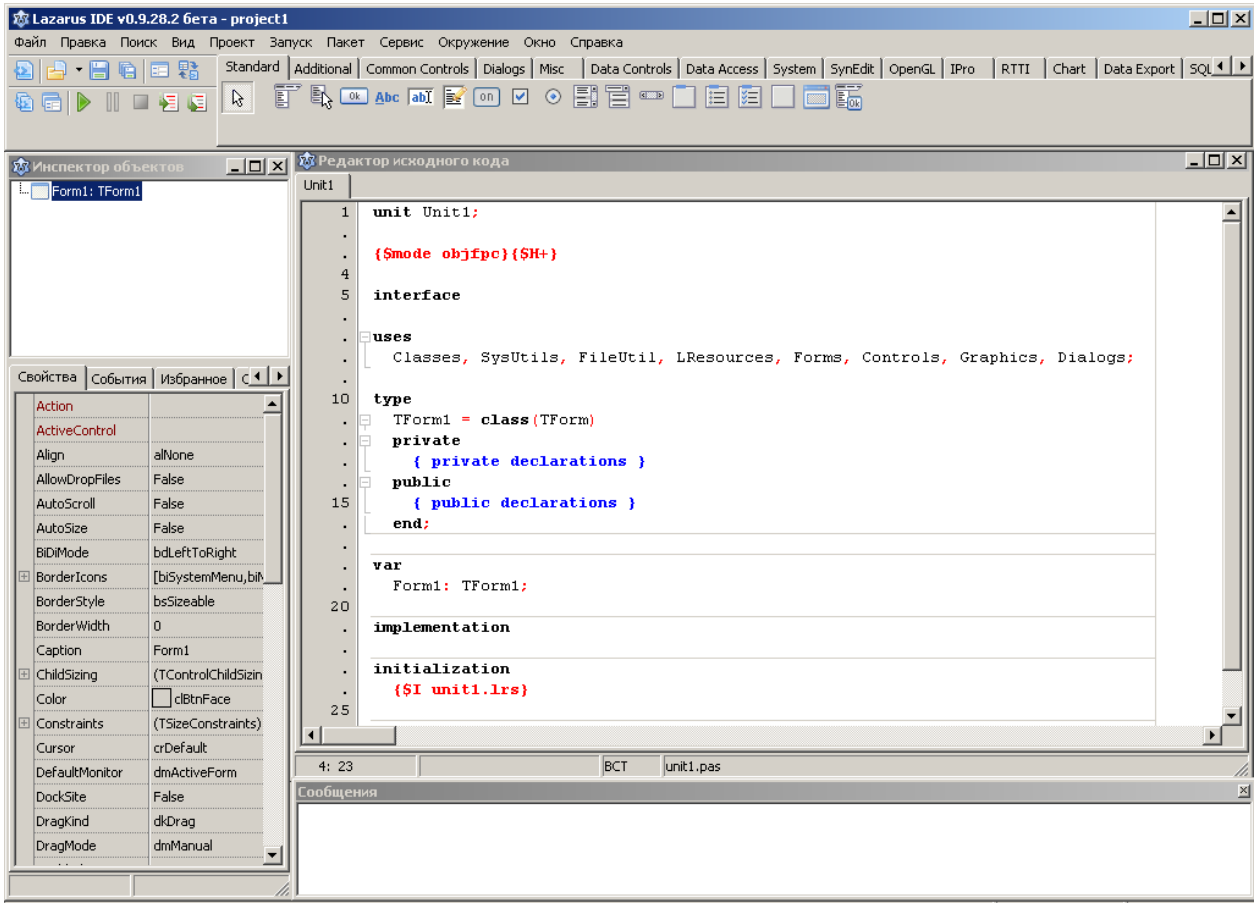
У цьому вікні виводяться повідомлення компілятора, компонувальника та налагодження
ка.

На цьому ми закінчимо наш короткий огляд середовища Lazarus. Ми розглянули
далеко не всі види вікон IDE, та й ті, що розглянули, ми розглянули швидко,

лише для того, щоб отримати перше уявлення про середовище Lazarus. Зані-

бути нудною розповіддю про всі пункти, опції та можливості Lazarus я зараз не буду. Адже не все буде зрозуміло, та й… нудно! Адже вам, шановний читачу, "не терпиться в бій"! Тому розглядатимемо лише ті елементи, які будуть нам потрібні спочатку. А решту ми будемо вивчати в міру потреби і в потрібних місцях. Так, на мою думку, буде краще!

Спочатку налаштуємо IDE так, як нам буде зручніше працювати. Як уже говорилося, при першому запуску вікна IDE Lazarus не пов'язані і є "плаваючими" вікнами. Раджу вам з'єднати їх так, як показано на рис. 2.11. шляхом зміни розмірів вікон, щоб Lazarus займав весь екран.

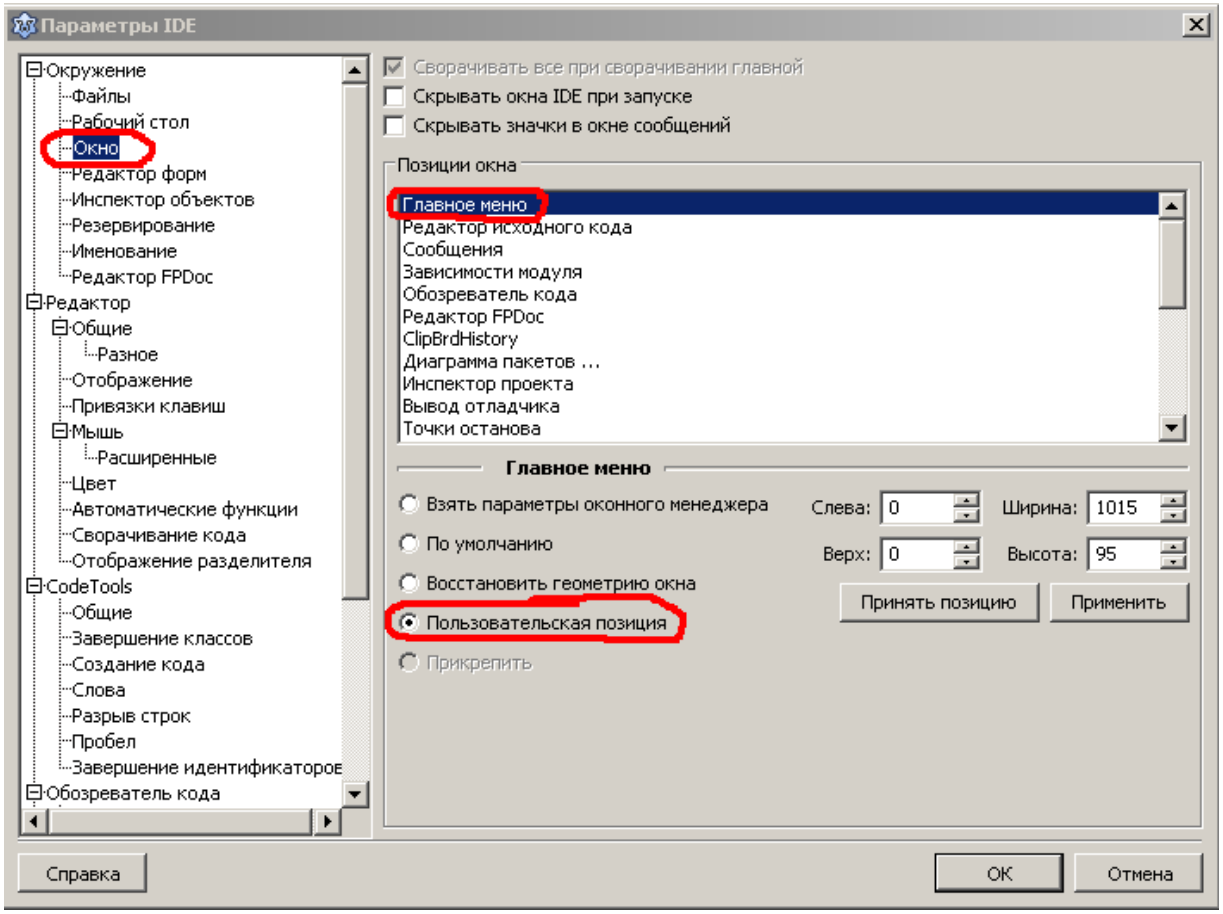


Мал. 2.11. Вигляд IDE Lazarus після зміни розмірів та положення вікон

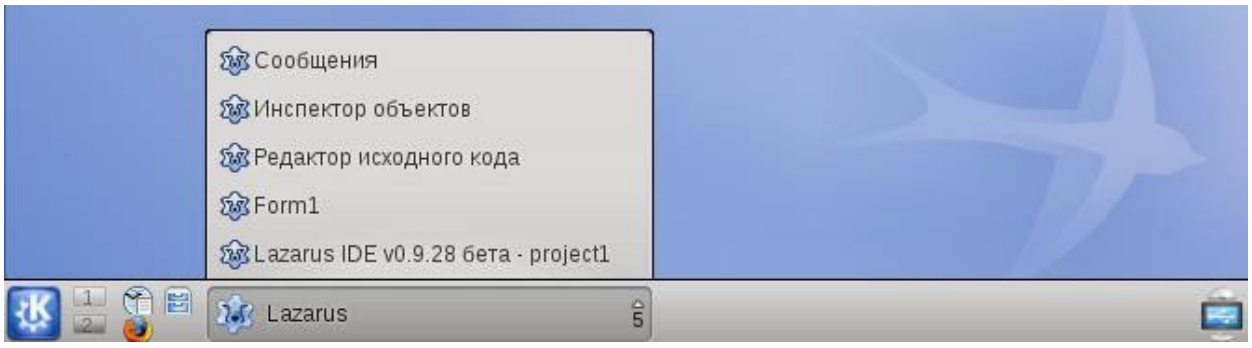
Після цього в Головному меню зайдіть в меню Оточення->Параметри... Відкриється вікно Параметри IDE. У цьому вікні у вкладці

Оточення виберіть Вікно. Далі для кожного вікна встановіть опцію "Позиція користувача" і натисніть кнопку "Застосувати", рис. 2.12.

У Windows всі вікна Lazarus після цього жорстко скріплюються, тому якщо, наприклад, у вас Lazarus був згорнутий, то після розгортання будуть видно всі вікна. У Linux негаразд. Розміри та положення вікон зберігаються, але ви можете відкривати вікна окремо, мал. 2.13.

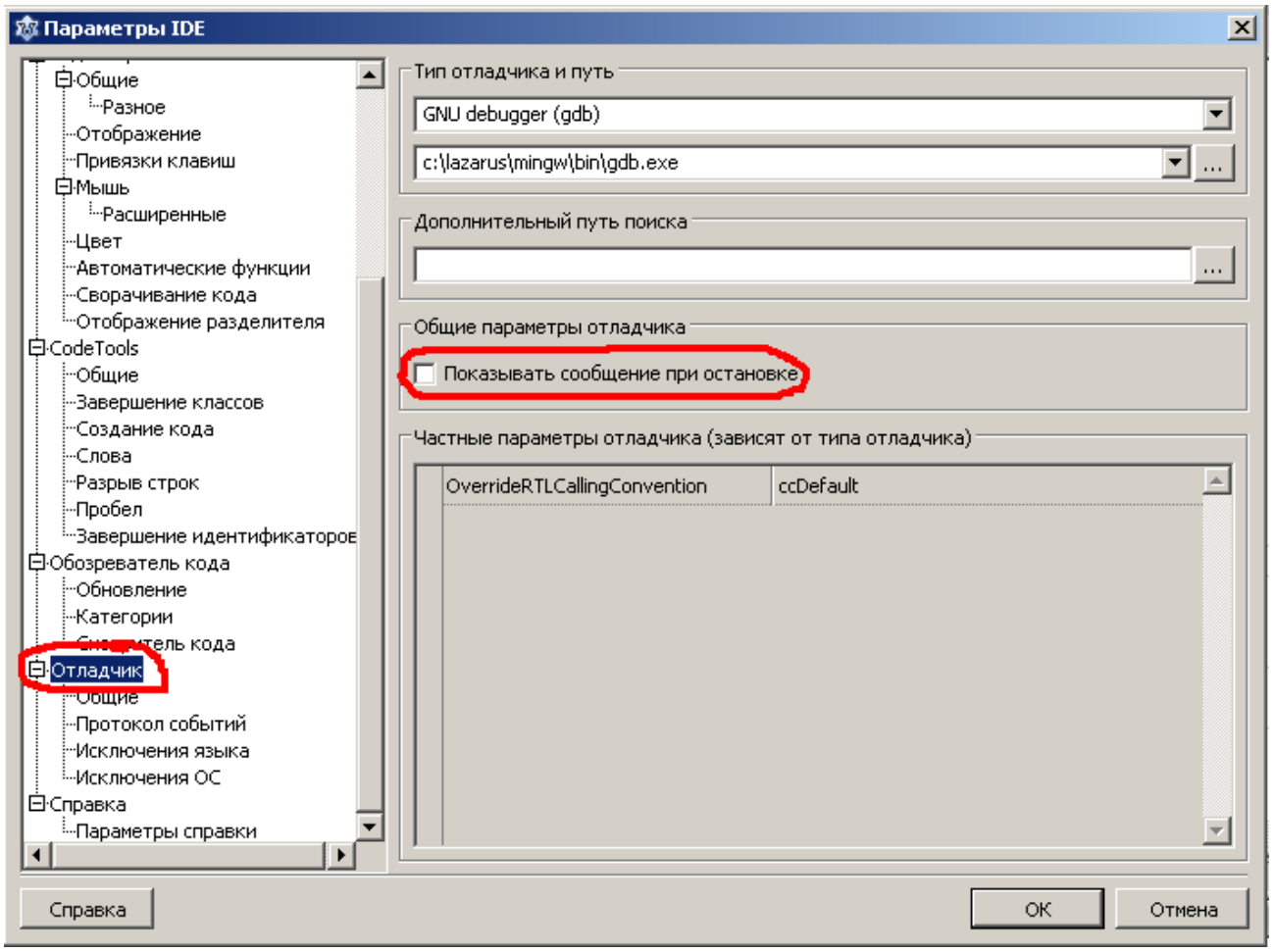


Мал. 2.12. Вкладка "Вікно" меню "Оточення"



Мал. 2.13. Вікна IDE Lazarus в Linux

Далі у тому ж вікні Параметри IDE виберіть пункт Відладчик і зніміть галочку з опції "Показувати повідомлення під час зупинки". Цим ми позбудемося набридливого повідомлення "Виконання зупинено" при кожному завершенні наших програм, рис. 2.14.



Мал. 2.14. Вікно налаштування параметрів відладчика

Вивчення мови найкраще починати з консольних програм. Консольна програма — програма, яка не має графічного інтерфейсу та виконується в текстовому режимі в консолі. У Windows консолі зазвичай відповідає вікно командного рядка. У Linux консолі відповідає вікно термінала. Для таких програм пристроєм введення є клавіатура, а пристроєм виведення — монітор, що працює в текстовому режимі відображення сим-

вільної інформації (літери, цифри та спеціальні знаки).

Консольна програма дозволяє зосередитися на суті тієї чи іншої конструкції мови, тому ми почнемо з консольних програм. Проте не можна вважати, що консольні програми призначені лише для вивчення мови. Можна і за допомогою консольних програм розробляти дуже складні програми. Наприклад, компілятор Free Pascal є консольною програмою, тобто може бути запущений в консолі, з командних файлів або з IDE Lazarus.

Для створення консольної програми потрібен лише текстовий редактор та компілятор Free Pascal. У принципі, для цього Lazarus не потрібен. Однак ми все ж таки використовуватимемо Lazarus, оскільки IDE Lazarus дозволяє створювати в тому числі і консольні програми. А його потужний текстовий редактор із підсвічуванням синтаксису та іншими широкими можливостями значно полегшить нам написання програм.

2.1.9 Російська мова в консольних додатках

У консольних програмах під Windows, на жаль, виникають проблеми з виведенням на екран російських букв. Це викликано різницею у кодуванні символів у Windows, що працює у графічному режимі та її консолі, яка працює у текстовому режимі. Не вдаючись поки в тонкощі, скажу лише, що ми у своїх програмах будемо використовувати спеціальну функцію UTF8ToConsole(), яка дозволить нам коректно відображати російські букви на екрані в консольних додатках для платформи Windows в операторах writeln та write.

Використовуйте функцію UTF8ToConsole() таким чином:

```
    writeln(UTF8ToConsole('Російський текст'));
```

замість

```
writeln('Російський текст');
```

У Linux таких проблем немає, тому можна писати просто

```
writeln('Російський текст');
```

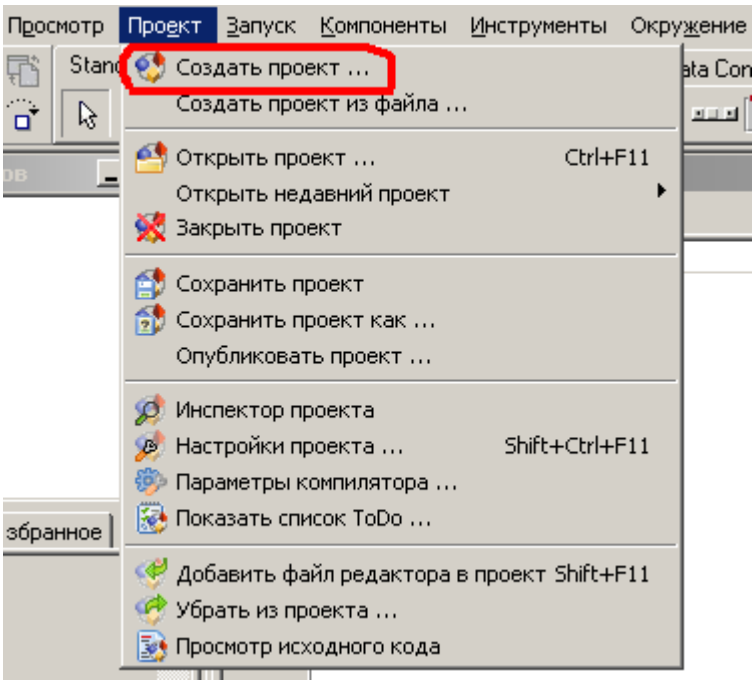
Але ми і в Linux писатимемо

```
writeln(UTF8ToConsole('Російський текст'));
```

Це дозволить нам без проблем переносити програми з Linux до Windows, тобто. наші програми без будь-яких переробок безпомилково компілюватимуться і виконуватимуться як на платформі Linux, так і на платформі Windows.

2.1.10 Перша програма

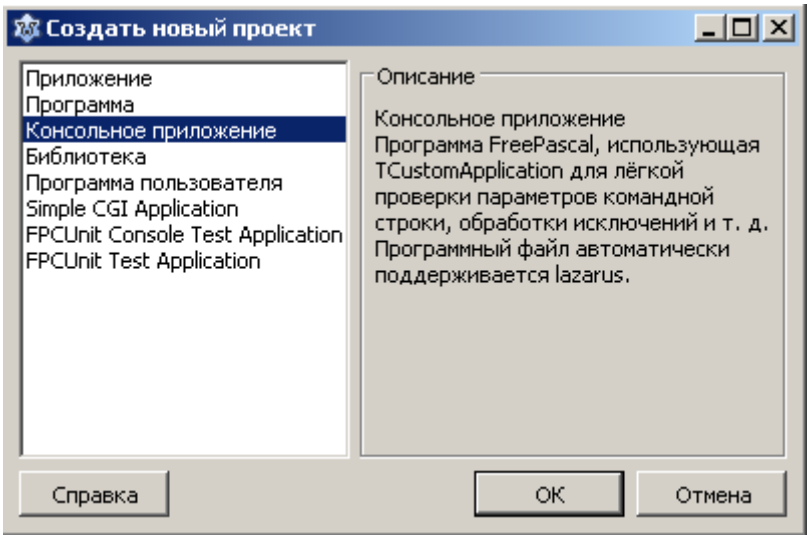
Запустіть Lazarus. Виберіть пункт меню Проект, Створити проект… (мал. 2.15).



Мал. 2.15. Меню "Проект"

Створіть консольну програму (рис. 2.16). Для цього виберіть Консольну

програму та натисніть Створити.



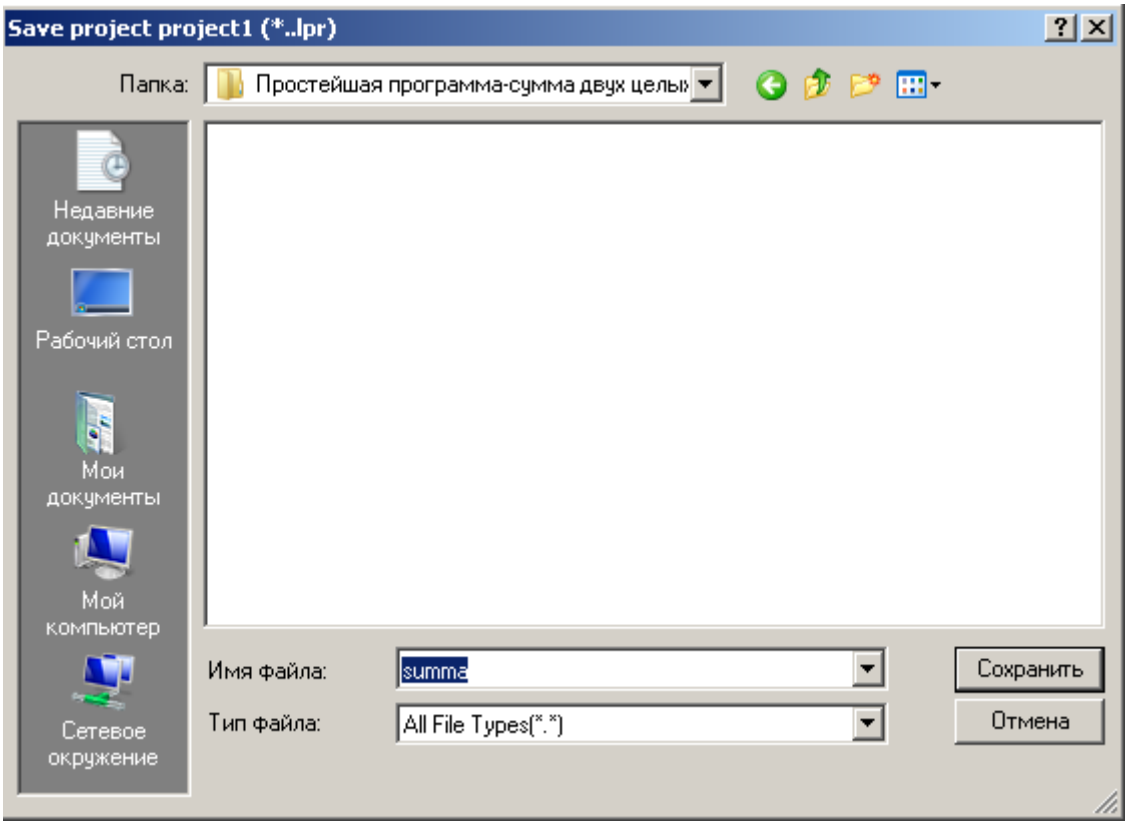
Мал. 2.16. Вікно створення нового проекту

Візьміть собі за правило відразу ж зберігати щойно створений проект, навіть якщо він поки що порожній. Це має стати вашою гарною звичкою, такою самою, як чистити зуби вранці та ввечері! Справа в тому, що під час збереження ви можете створити папку для свого проекту, і всі файли поточного проекту будуть збережені в окремій папці. Це допоможе вам структурувати ваші проекти та не заплутатися в них, якщо їх буде багато.

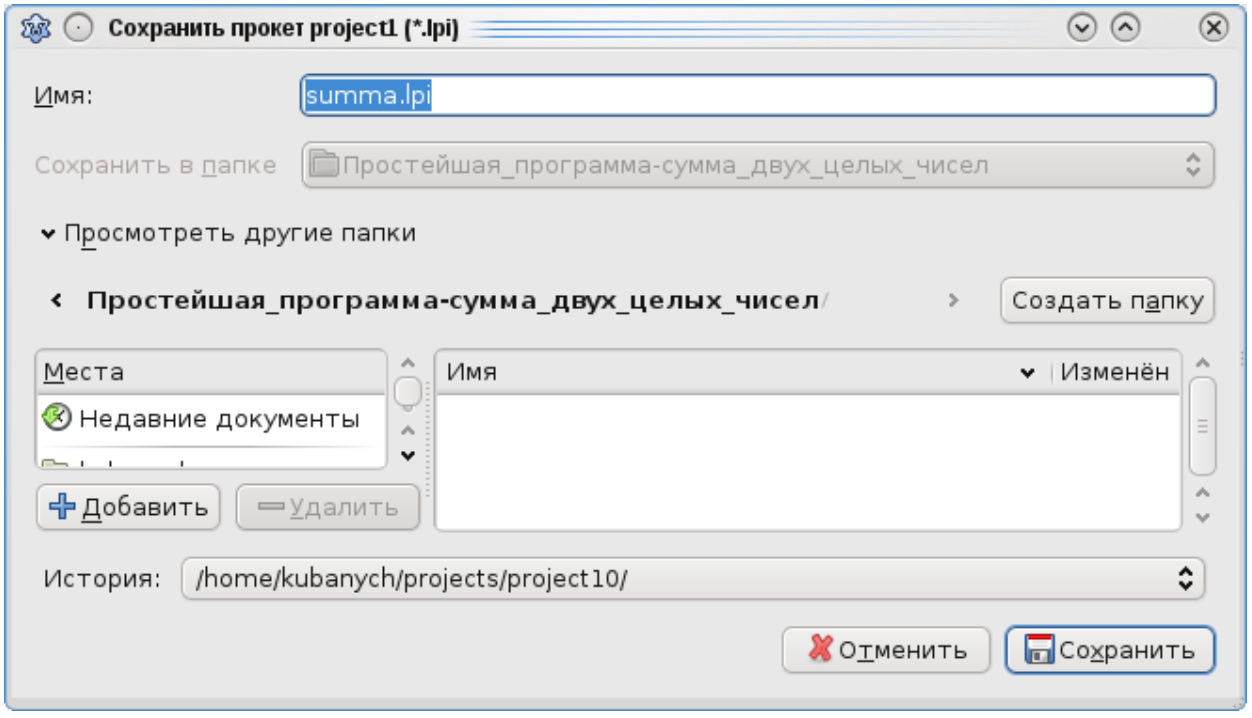
Для збереження проекту найпростіше користуватися кнопками на панелі інструментів, рис. 2.17.

Мал. 2.17. Кнопки збереження проекту

У діалоговому вікні збереження проекту, що відкрилося, створіть нову папку в потрібному місці, вкажіть ім'я проекту і натисніть Зберегти, мал. 2.18, 2.19.



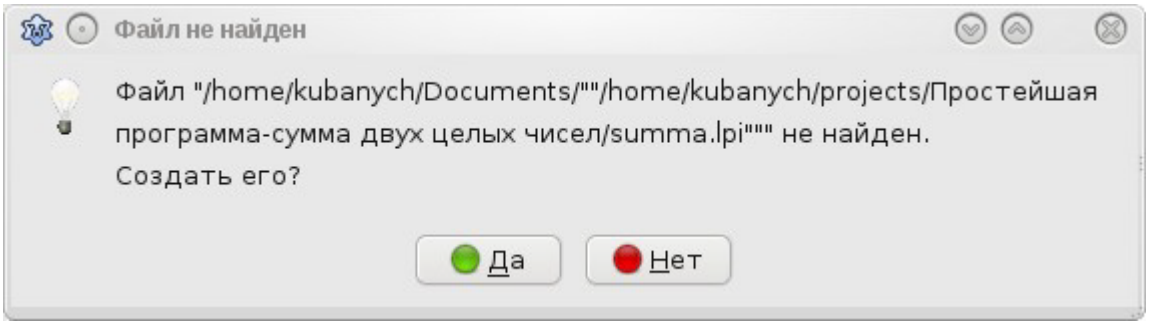
Мал. 2.18. Стандартне діалогове вікно збереження у Windows



Мал. 2.19. Стандартне діалогове вікно збереження у Linux

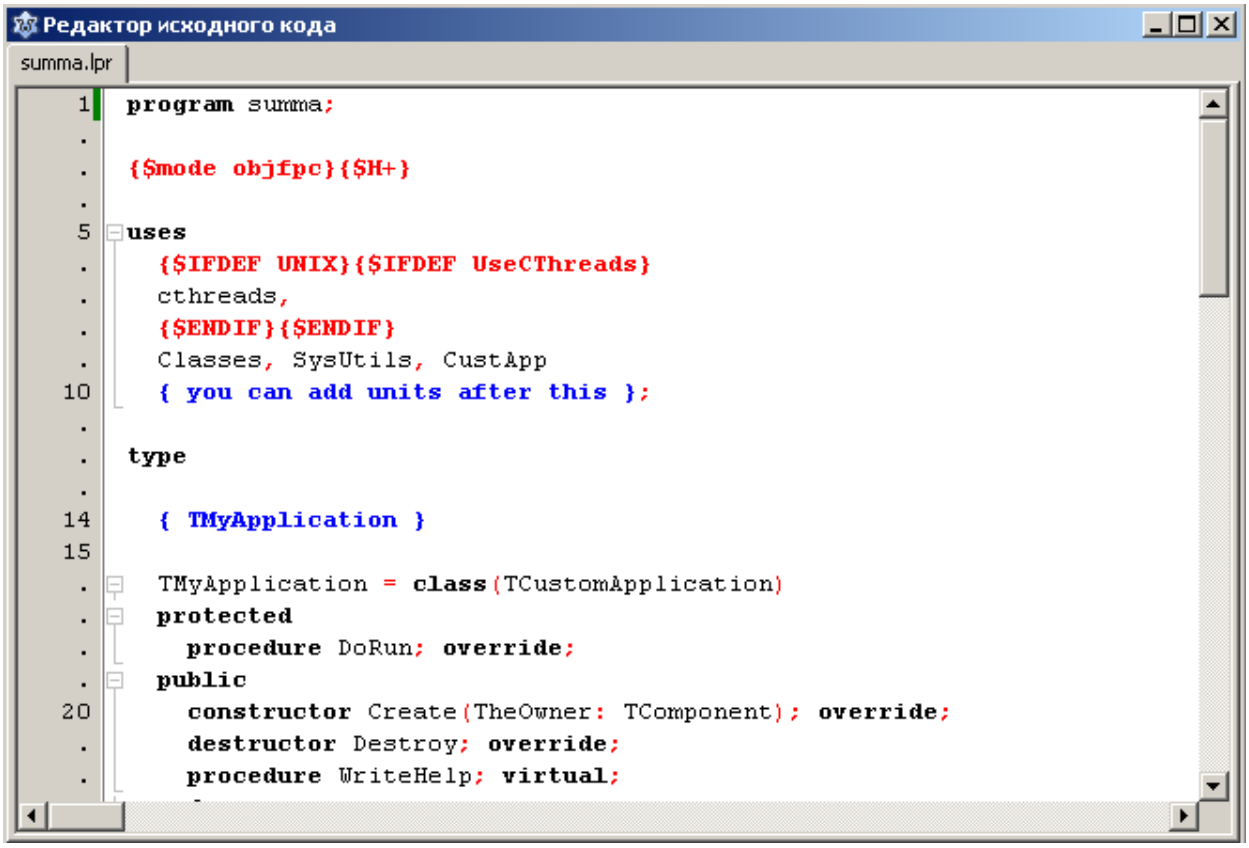
Під час встановлення імені папки та імені проекту намагайтеся, щоб імена відображали суть проекту. Це допоможе вам легко орієнтуватися у своїх проектах,

особливо коли їх накопичиться чимало. Також пам'ятайте, що якщо ви даєте ім'я, що складається з декількох слів, то в Linux не можна ставити прогалини між словами. В цьому випадку Lazarus не зможе відкрити ваш проект, рис. 2.20. Ім'я проекту завжди задавайте у нижньому регістрі.



Мал. 2.20. Вікно повідомлення "Файл не знайдено"

Після збереження в папці з проектом з'явиться кілька файлів, які ми розглянемо пізніше. У вікні редактора вихідного коду ви побачите текст. Це заготовка коду для консольної програми, що автоматично вставляється Lazarus (рис. 2.21).

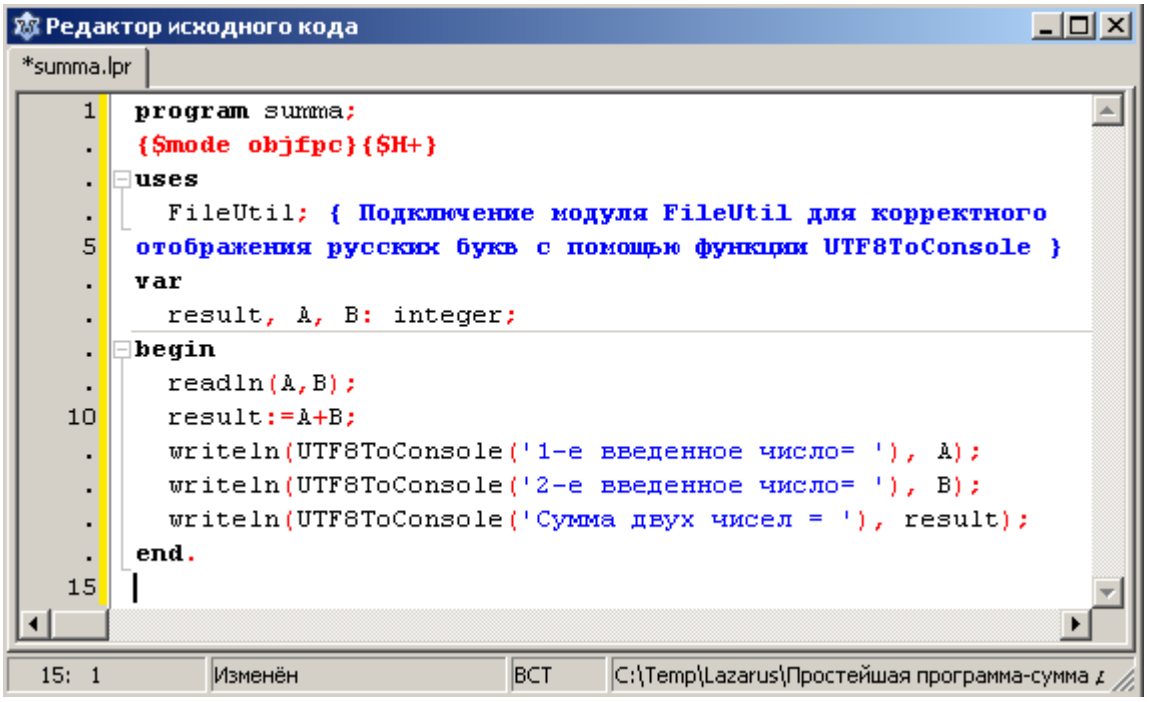


Мал. 2.21. Заготовка коду для консольної програми, що автоматично вставляється Lazarus

Ми не будемо зараз звертати увагу на цей код і розбирати його, оскільки у нас для цього поки що недостатньо знань. Просто видаліть цей код. Для цього встановіть курсор у будь-яке місце у вікні редактора вихідного тексту і натисніть Ctrl+A. Увесь текст у вікні виділиться. Натисніть Delete. Введіть наступний код програми:

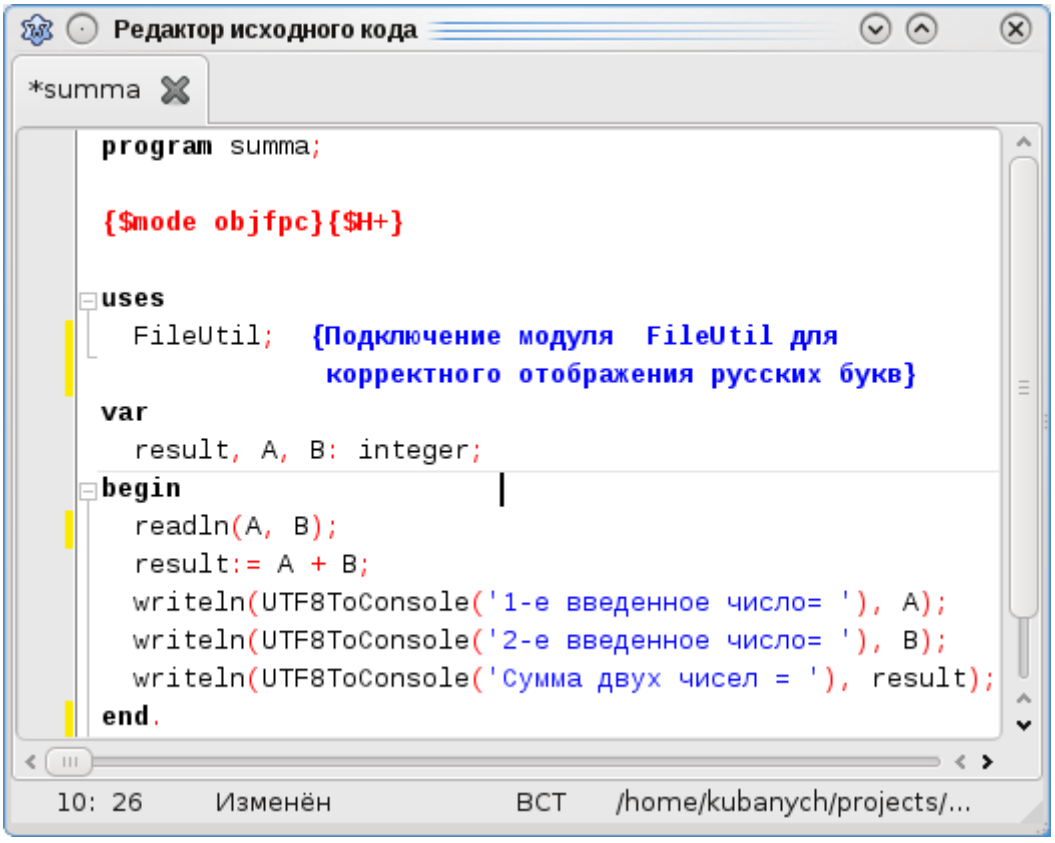
```
program summa;
{$mode objfpc}{$H+}
uses
  FileUtil; { Підключення модуля FileUtil для
коректного відображення російських букв за
допомогою функції UTF8ToConsole } var
  result, A, B: integer;
begin
  readln(A, B);
  result:=A + B;
  writeln(UTF8ToConsole('1-е введене число='), A);
  writeln(UTF8ToConsole('2-е введене число='), B);
  writeln(UTF8ToConsole('Сума двох чисел = '),
  result);
end.
```

Вікно редактора вихідного коду Windows матиме вигляд, рис. 2.22:



Мал. 2.22. Вікно редактора вихідного коду Windows

У Linux це вікно матиме вигляд, рис. 2.23.



Мал. 2.23. Вікно редактора вихідного коду в Linux

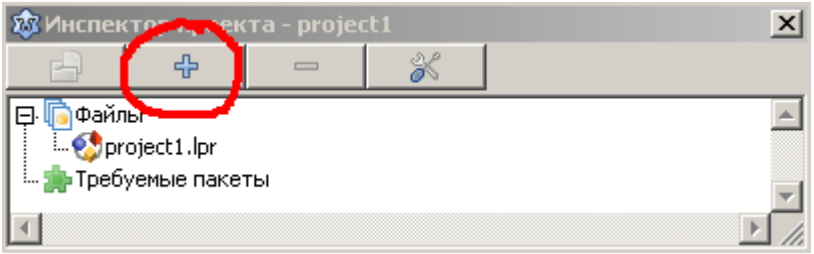
Зверніть увагу на оголошення

```
uses
   FileUtil;
```

Цим оголошенням ми підключаємо модуль FileUtil, в якому визначено функцію UTF8ToConsole().

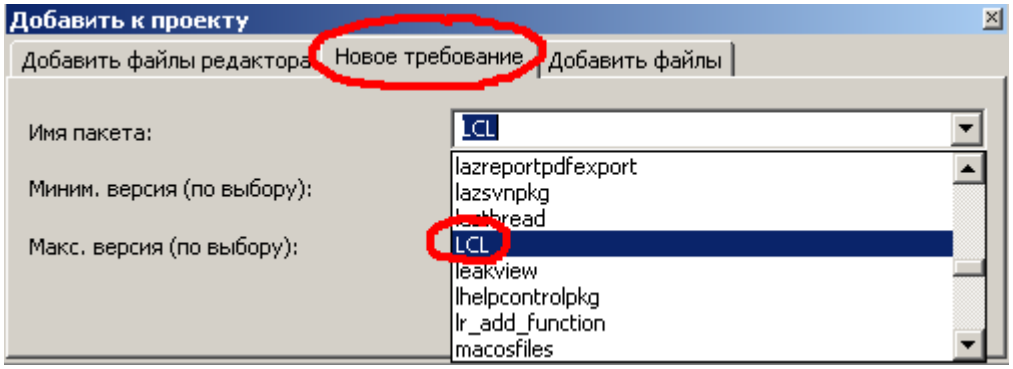
Якщо вас бентежить, що означає модуль і функція в Паскалі, то трохи потерпіть. У розділі 3 ми докладно розглянемо ці питання. Нагадую, що ми змушені це робити, щоб у Windows у вікні DOS під час роботи вашої програми коректно відображався російський шрифт. Також поки що прийміть на віру і проробіть наступне.

Відкрийте меню Проект->Інспектор проекту та натисніть на кнопку зі значком "+", мал. 2.24.



Мал. 2.24. Вікно інспектора проекту

У вікні "Додати до проекту" натисніть на кнопку "Нова вимога", рис. 2.25.



Мал. 2.25. Додавання нової вимоги

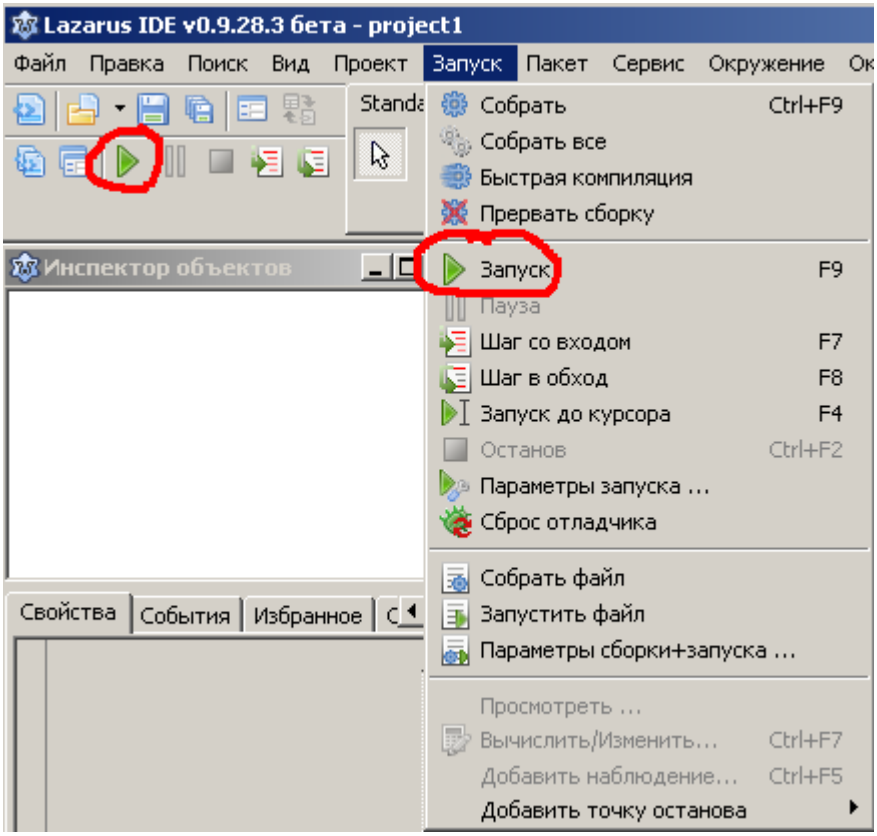
У розкривному списку "Ім'я пакета" знайдіть та виберіть пакет LCL. Натисніть клавіші Ctrl+F9. Почнеться компіляція та складання програми. Ес-

Чи ви ввели текст програми без помилок точно як наведено вище, то компіляція завершиться успішно. У вікні Повідомлення ви побачите повідомлення Проект summa успішно зібраний.

У папці проекту з'явиться, на додаток до існуючих, ще кілька файлів. Зокрема готовий до виконання файл. У Windows це буде файл із розширенням exe, у Linux файл без розширення.

Щоб запустити програму виконання прямо з середовища Lazarus на-

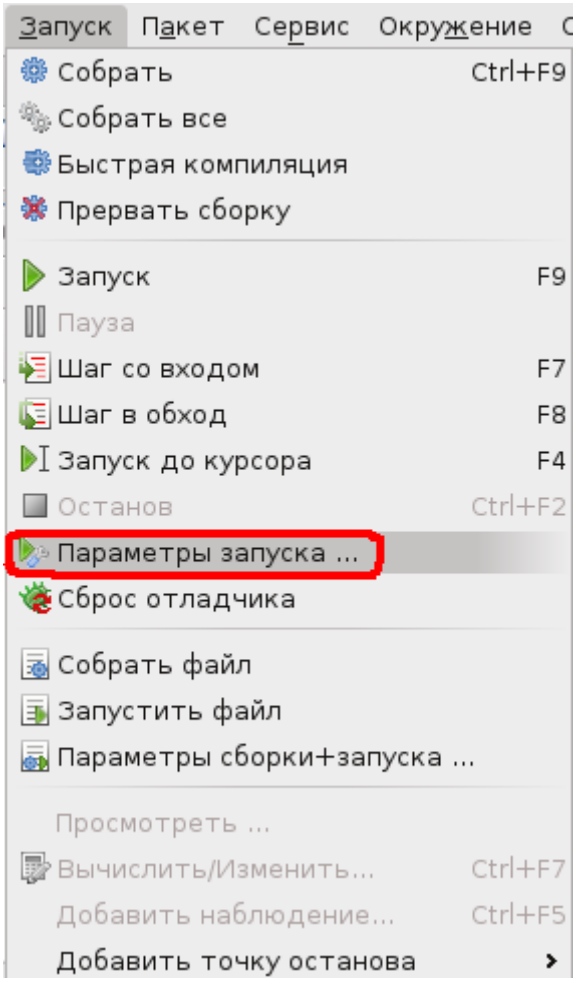
натисніть клавішу F9 або кнопку "Запуск" (зелений трикутник) на панелі інструментів або меню Запуск->Запуск, мал. 2.26.



Мал. 2.26. Способи запуску програми

Користувачам Linux для того, щоб запускати програми з середовища Lazarus в терміналі, необхідно в меню Запуск->Параметри запуску встано-

вити прапорець "Використовувати програму для запуску", рис. 2.27, 2.28.



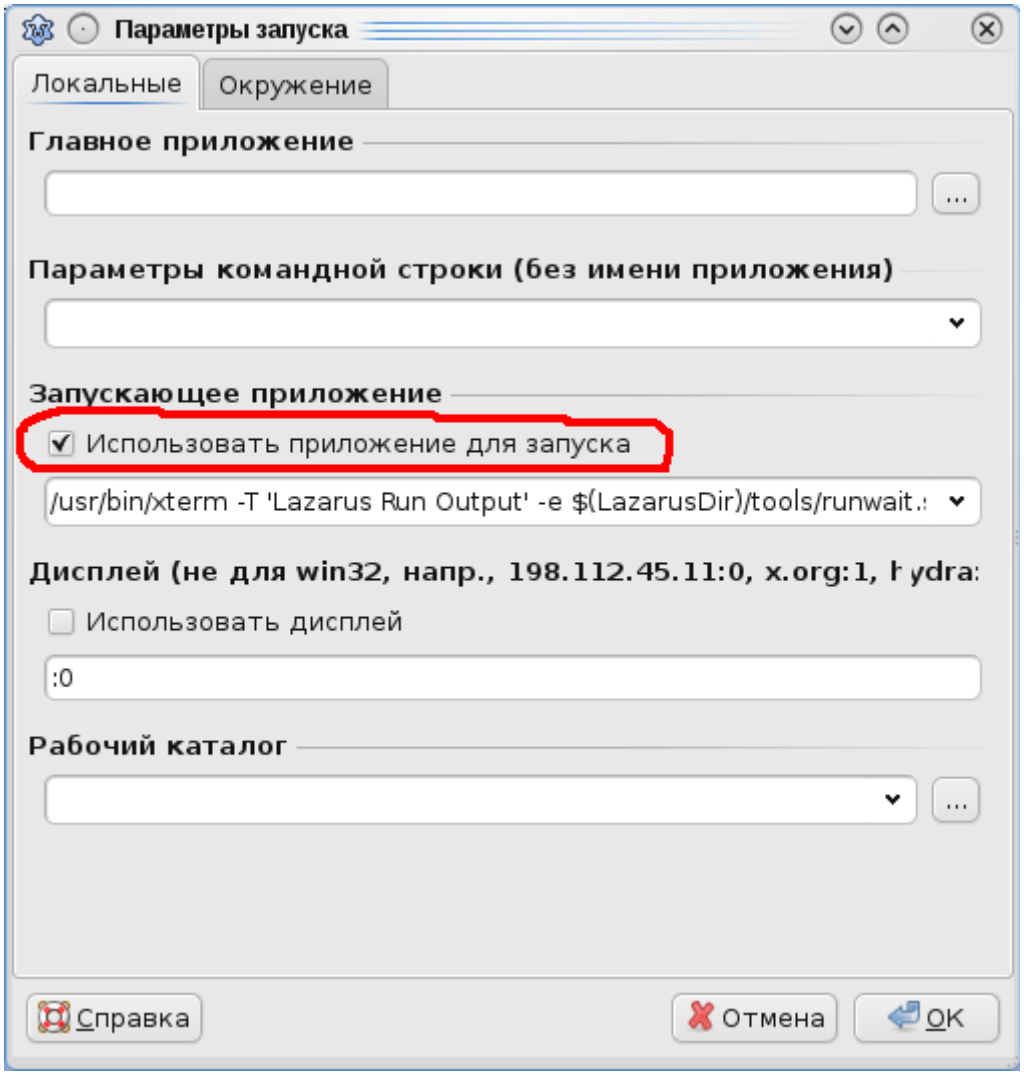
Мал. 2.27. Меню "Запуск"

При цьому для деяких дистрибутивів Linux слід замінити рядок

/usr/X11R6/bin/xterm -T 'Lazarus Run Output' -e $(LazarusDir)/tools/runwait.sh $(TargetCmdLine)

на

/usr/bin/xterm -T 'Lazarus Run Output' -e $(LazarusDir)/tools/runwait.sh $(TargetCmdLine)



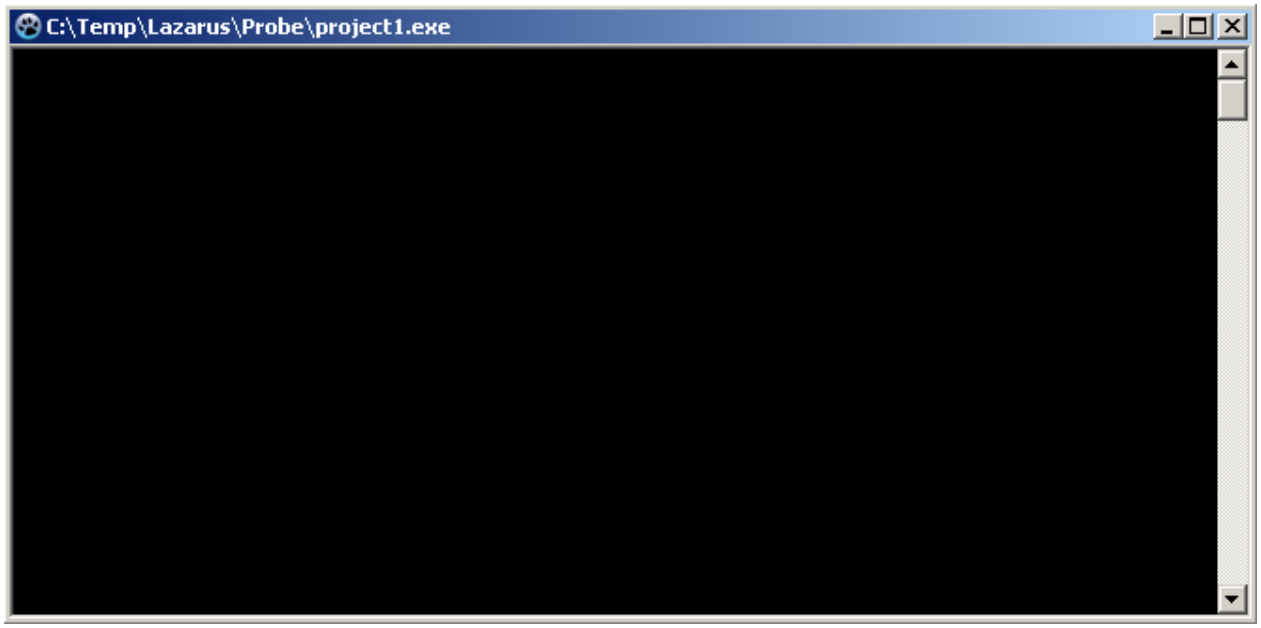
Мал. 2.28. Налаштування проекту для запуску у терміналі

Для запуску програми поза середовищем Lazarus в Windows достатньо двічі клацнути на ім'я виконуваного exe-файлу.

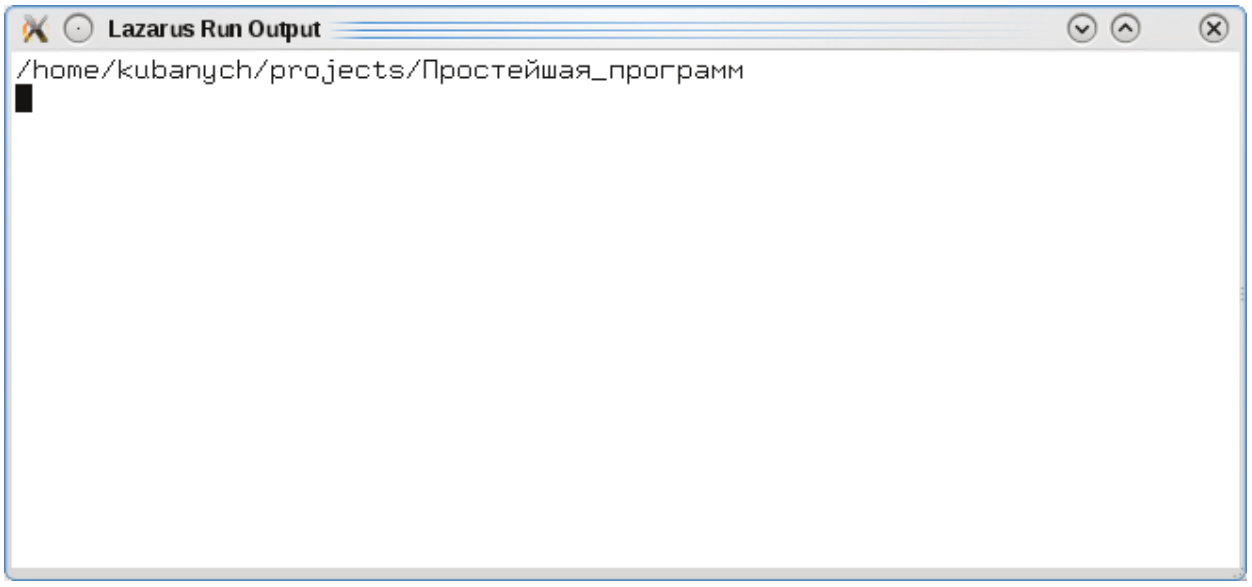
У Linux видати команду <шлях до файлу> ./<ім'я файлу>

Надалі, для одноманітності у викладі, будемо припускати, що це приклади у книзі запускаються з середовища Lazarus.

Після запуску програми з'явиться вікно виду, рис. 2.29 (Windows) та рис. 2:30 (Linux).



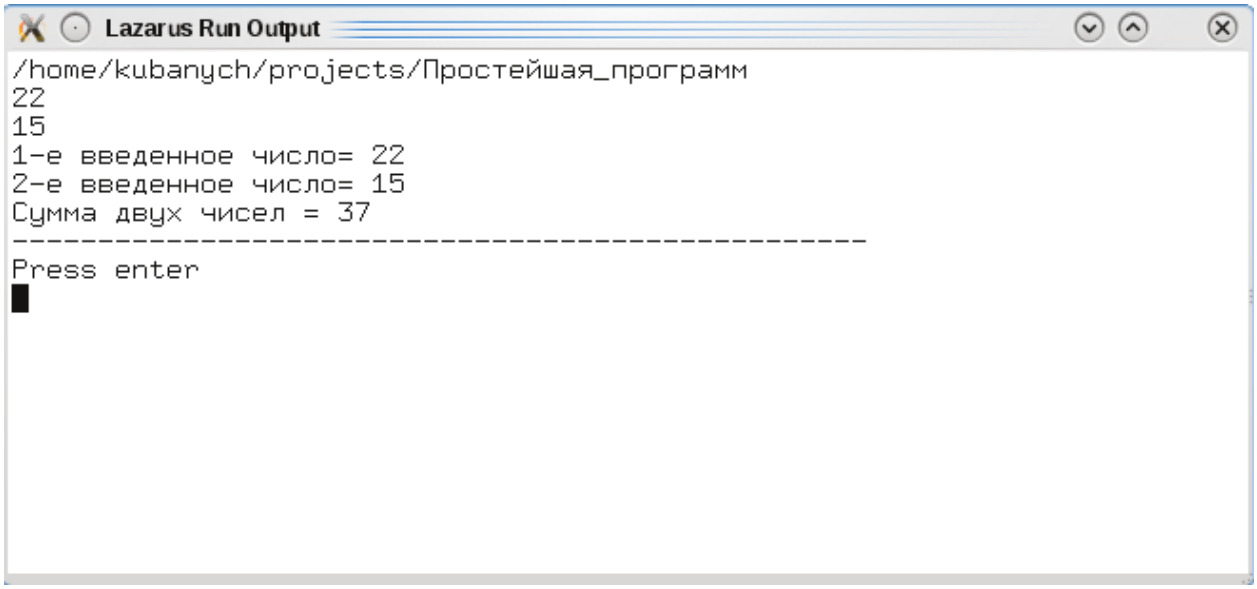
Мал. 2.29. Вікно виконуваної програми у Windows



Мал. 2.30. Вікно програми, що виконується в Linux

Введіть значення змінних A та B. Це мають бути цілі числа. Щоб завершити введення числа, натисніть клавішу Enter.

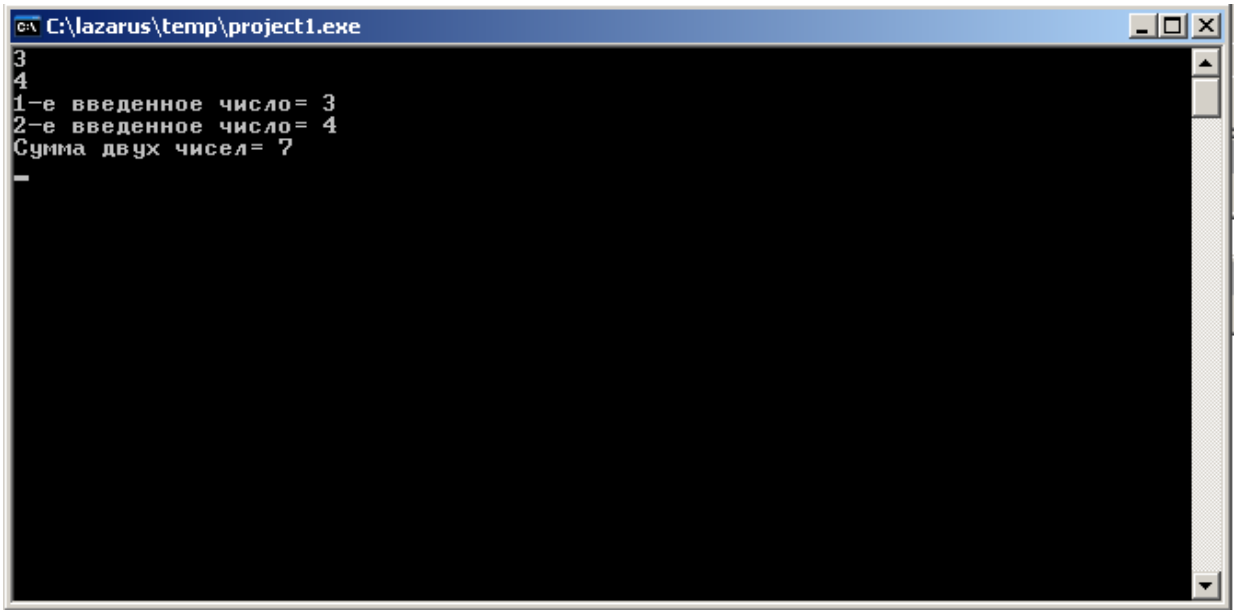
Вікно виведення програми в Linux матиме вигляд, рис. 2.31:



Мал. 2.31. Вікно виведення програми в Linux

У Windows після введення другого числа у вікні щось промайне і вікно закриється. Як зробити так, щоб вікно не зачинялося, і ми могли побачити результати виконання програми?

Для цього можна вставити в код програми оператор readln без параметрів перед завершальним оператором end з точкою. Знову натисніть клавішу F9. Цього разу після введення чисел вікно не закриється, і ви зможете побачити результати роботи програми, рис. 2.32.



Мал. 2.32. Вікно виведення програми у Windows

Щоб закрити вікно програми, натисніть Enter.

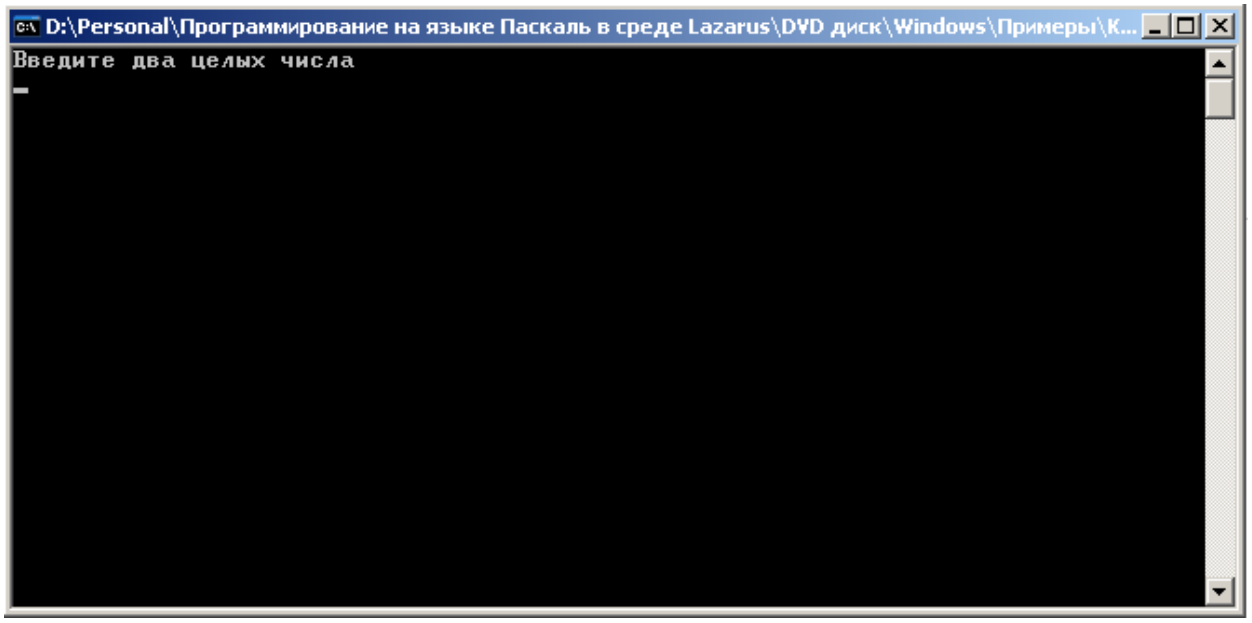
Проаналізуємо нашу програму. Наче програма працює, сума обчислюється правильно. І все ж навіть у цій найпростішій програмі помітні істотні недоліки. Недоліки з погляду організації взаємозв'язку з користувачем чи, як кажуть, інтерфейсу зв'язку з користувачем.

По-перше, після запуску нашої програми з'являється пусте вікно. Це відразу ж викликає негативні асоціації. Щось сталося з комп'ютером? Він завис?

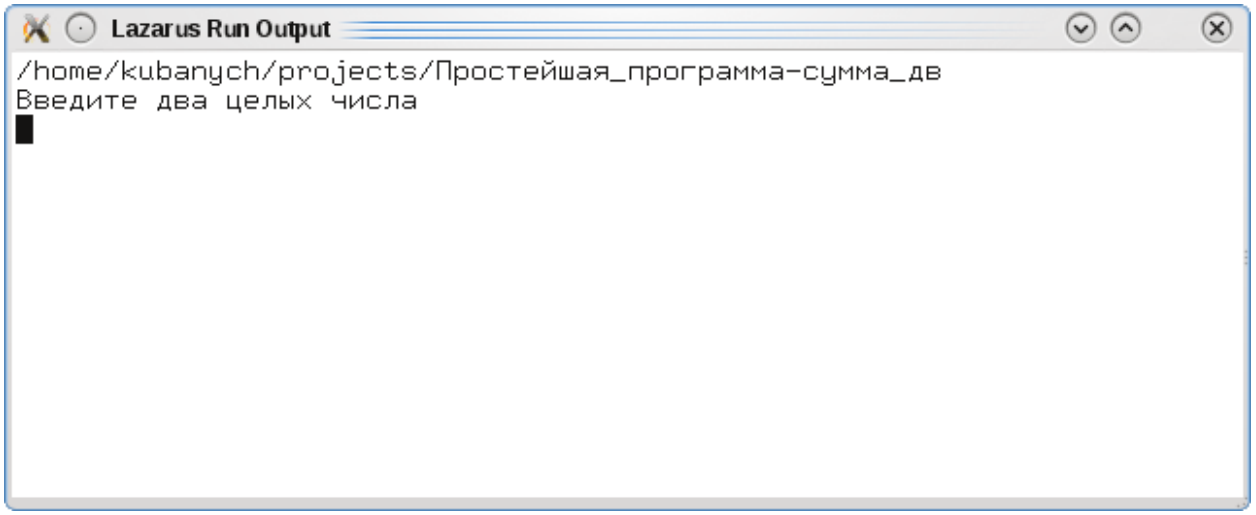
Ну, ми з вами знаємо, що потрібно ввести два числа, але користувач може і не знати про це! Як виправити це? А дуже просто! Вставте оператор

```
writeln(UTF8ToConsole('Введіть два цілих числа')); перед оператором введення readln(A, B);
```

Вікно програми виглядатиме так, рис. 2.33, 2.34:



Мал. 2.33. Вікно програми у Windows



Мал. 2.34. Вікно програми в Linux

Тепер навіть їжу зрозуміло, що треба запровадити два числа!

По-друге, (це стосується знову ж таки Windows) непогано було б дати користувачеві інструкцію як вийти їхні програми. Вікно програми буде закрито лише після натискання клавіші Enter. Натискання будь-яких інших клавіш до закриття вікна не призведе! Можна, звичайно, скористатися кнопкою закриття вікна. Але бажано, щоб програма сама закривала власне вікно. І знову, адже користувач не знає, що для виходу з програми потрібно натиснути клавішу Enter. Є два варіанти виходу із цієї ситуації:

1. В кінці програми, але перед readln вставити оператор

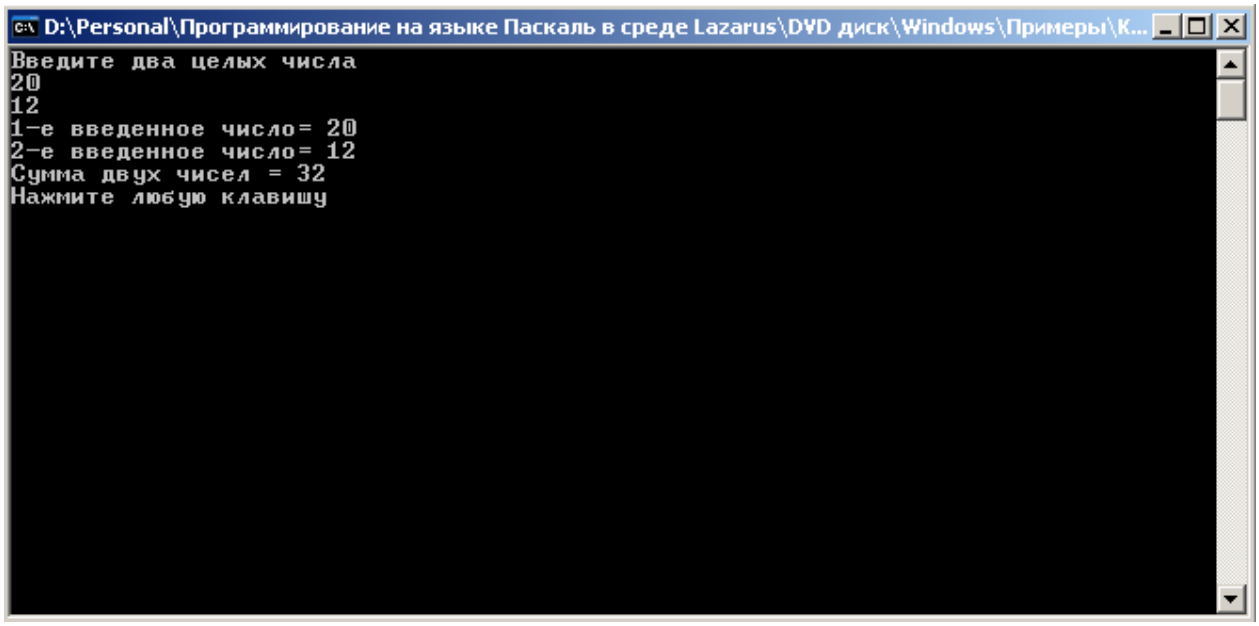
```
writeln(UTF8ToConsole ('Для виходу натисніть Enter'));
```

2. Скористайтеся функцією readkey без параметрів. Вікно програми закриватиметься при натисканні будь-якої клавіші. Цей варіант кращий, тільки для цього необхідно використовувати оголошення`uses Crt`, щоб увімкнути модуль CRT про який мова піде пізніше і в якому і є цяфункція. навіть у цьому випадку буде правильніше, якщо вставити оператор (зрозуміло, перед readkey) writeln(UTF8ToConsole('Для виходу натисніть будь-яку клавішу'));

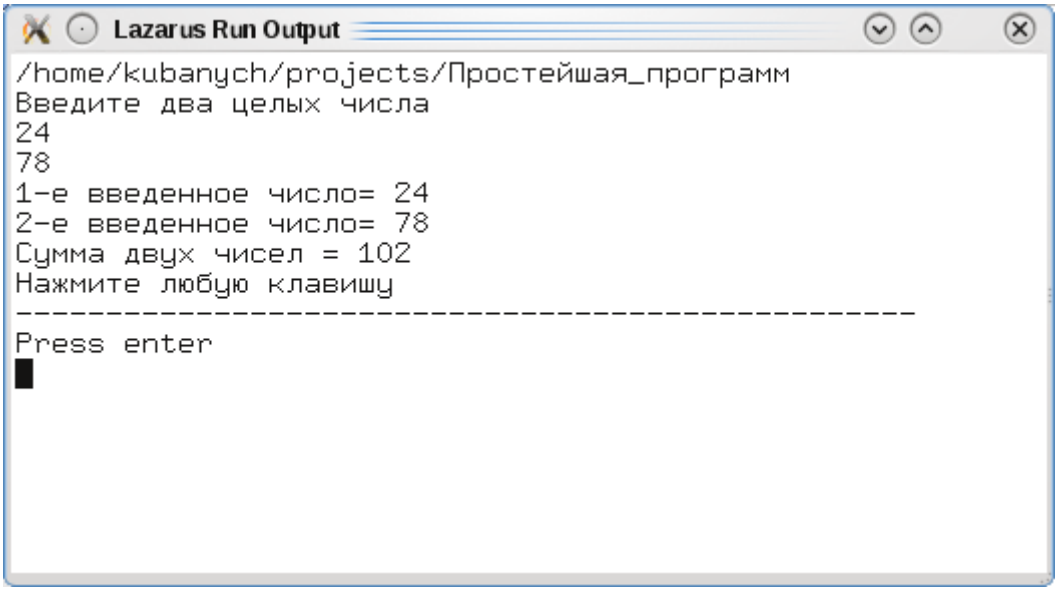
Остаточний текст програми виглядатиме так:

```
program summa;
{$mode objfpc}{$H+}
uses
  Crt, FileUtil; {Підключення модуля CRT для використання
функції readkey та модуля FileUtil для коректного
відображення російських букв
за допомогою функції UTF8ToConsole}
var
  result, A, B: integer;
begin
  writeln(UTF8ToConsole('Введіть два числа'));
  readln(A, B);
  result:=A + B;
  writeln(UTF8ToConsole('1-е введене число='), A);
  writeln(UTF8ToConsole('2-е введене число='), B);
  writeln(UTF8ToConsole('Сума двох чисел = '),
  result); writeln(UTF8ToConsole('Натисніть будь-
  яку клавішу')); readkey;
end.
```

Результати роботи програми наведено на рис. 2.35, 2.36.



Мал. 2.35. Результати роботи програми у Windows



Мал. 2.36. Результати роботи програми в Linux

Користувачі Linux зверніть увагу, що якщо ви виконуєте програму в терміналі, то ще додатково видається повідомлення "Press enter". Все ж таки оператори

```
writeln(UTF8ToConsole('Натисніть будь-яку
клавішу')); readkey;
```

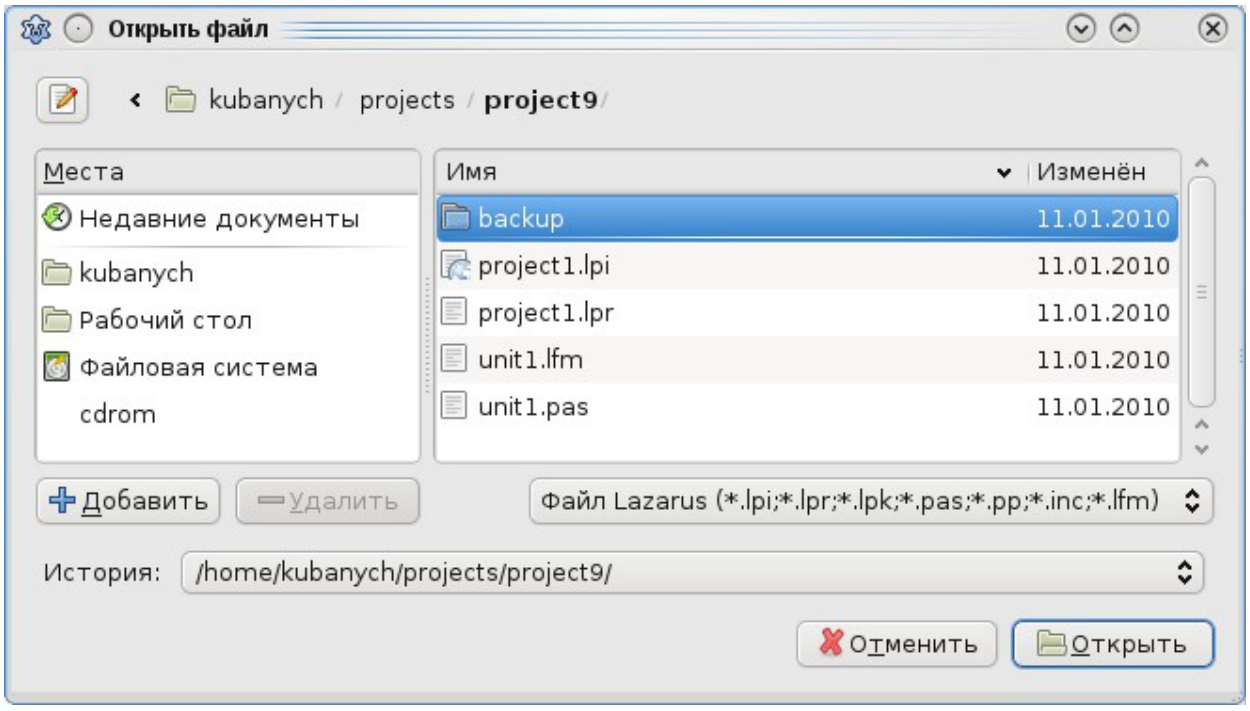
ми залишимо, тому що книга розрахована не тільки для "лінуксоїдів", але і для "віконників"! А у Windows, як ми вже бачили, ці оператори потрібні. Ну, а користувачам Linux залишаємо право вибирати, використовувати ці оператори чи ні, так само як і застосовувати функцію UTF8ToConsole().

2.1.11 Відкриття існуючого проекту

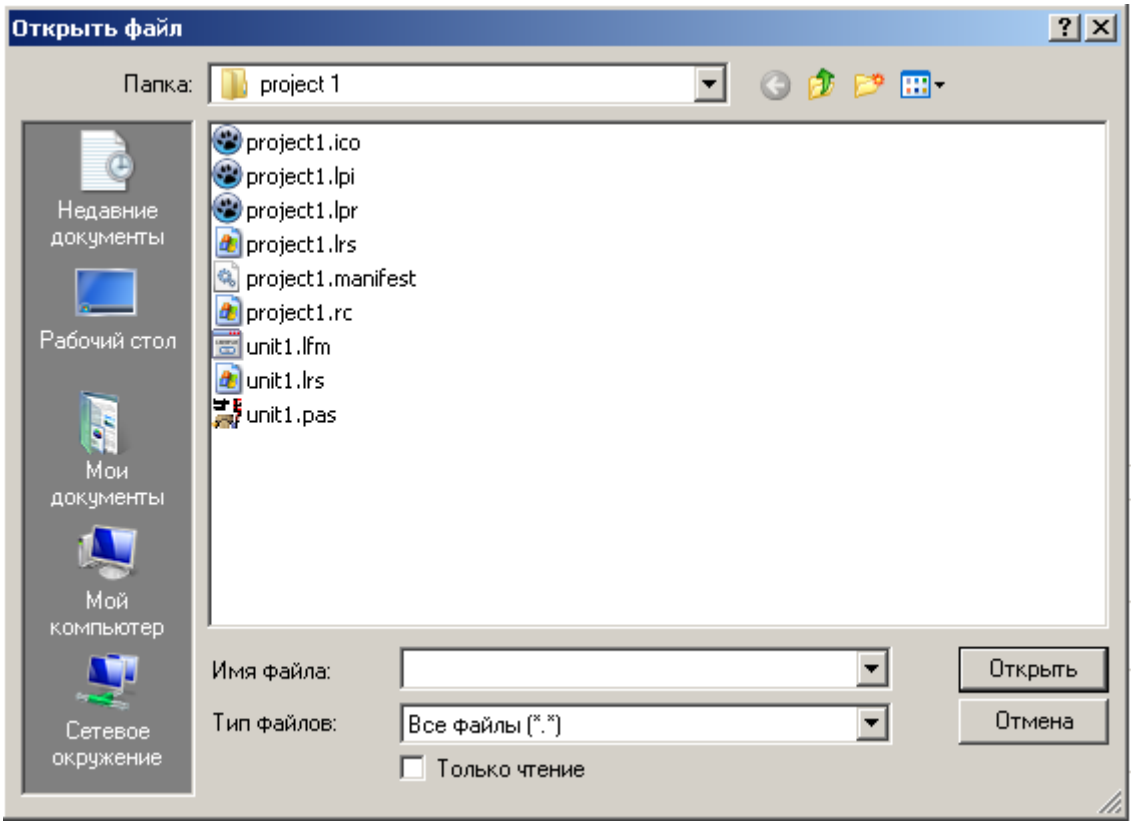
Відкрити існуючий проект із середовища Lazarus можна, скориставшись кнопкою в панелі інструментів, мал. 2.37.

Мал. 2.37. Кнопка відкриття існуючого проекту

або в меню "Файл" вибрати пункт "Відкрити..." або натиснути клавіші Ctrl+O. Буде відкрито стандартний діалог, рис. 2.38, 2.39.



Мал. 2.38. Стандартний діалог відкриття проекту в Linux



Мал. 2.39. Стандартний діалог відкриття проекту у Windows

Якщо у вас Lazarus не запущений, можна відкрити існуючий проект двічі клацнувши на ім'я файлу з розширенням *.lpi або *.lpr. Файл з розширенням lpi (Lazarus Project Information) є основним файлом проекту Lazarus, що зберігається у XML форматі.

Файл із розширенням lpr – вихідний код основної програми. Не дивлячись на специфічне для Lazarus розширення насправді це є звичайнийPascal код. У Linux, якщо замість Lazarus запускатиметься інша програма (найчастіше Konqueror або KWrite), то треба налаштувати файловий менеджер, щоб він відкривав Lazarus по подвійному клацанню на ім'я файлу.

Клацніть правою клавішею миші на ім'я файлу проекту. У контекстному меню, що відкрилося, виберіть пункт "Властивості…", рис. 2.40. Далі у вікні "Властивості", що відкрилося, натисніть на кнопку із зображенням гайкового ключа, рис. 2.41. У вікні "Змінити тип файлу" натисніть кнопку "Додати…", мал.

2.42. Відкриється вікно "Вибір програми…", рис. 2.43. У цьому вікні введіть

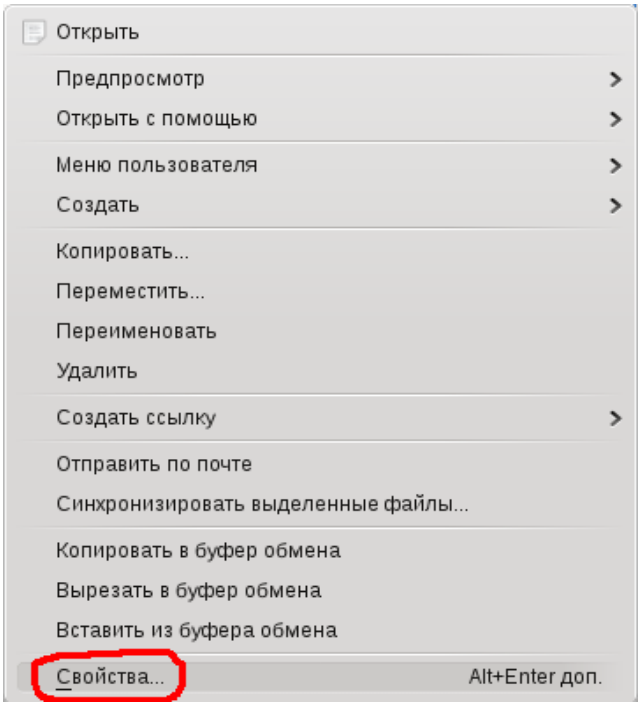
ім'я програми за допомогою до нього, наприклад

```
/usr/lib/lazarus/lazarus
```

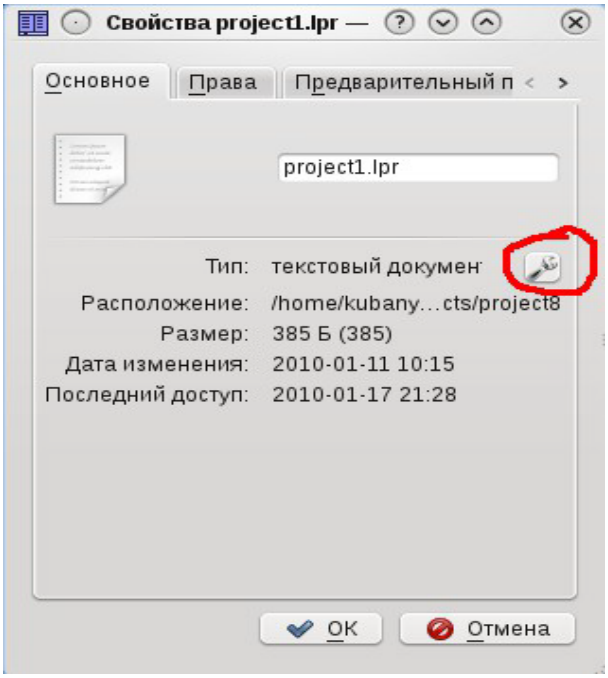
або

```
/usr/lib/lazarus/startlazarus
```

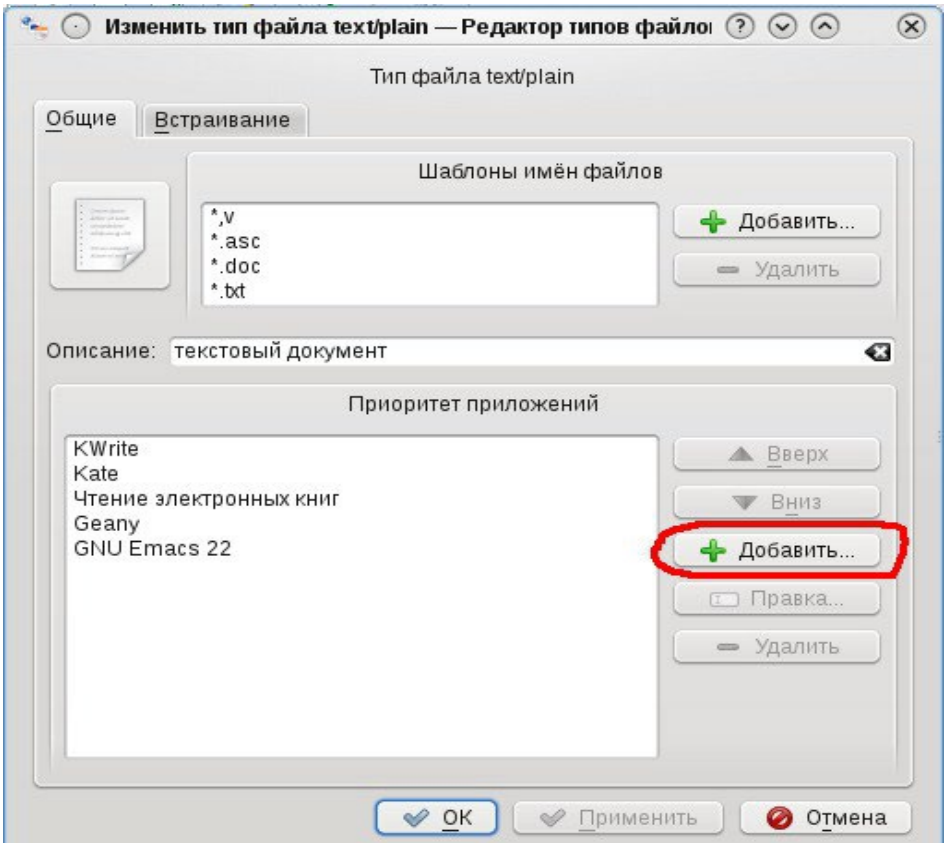
або натисніть кнопку вибору файлу та у вікні "Редактор типів файлу" знайдіть папку, де встановлений Lazarus, мал. 2.44.



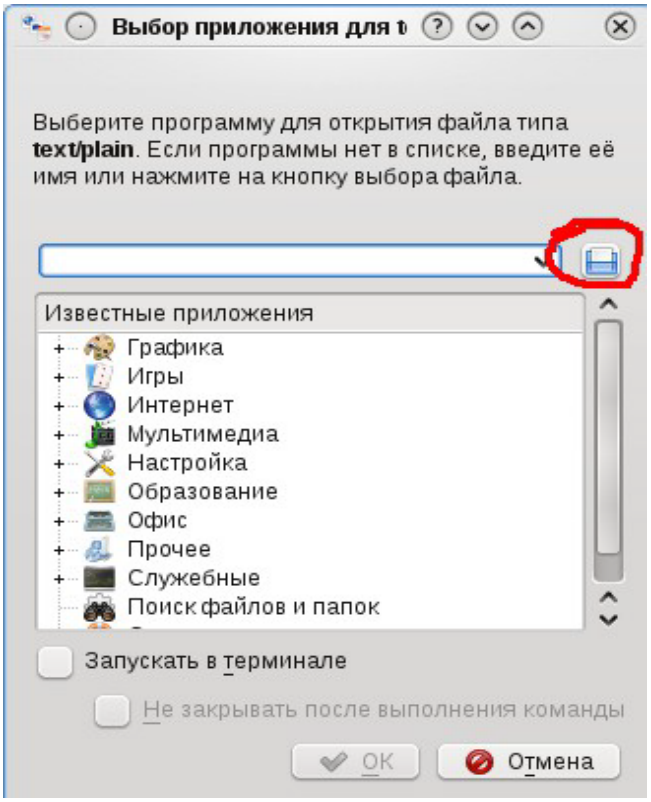
Мал. 2.40. Контекстне меню



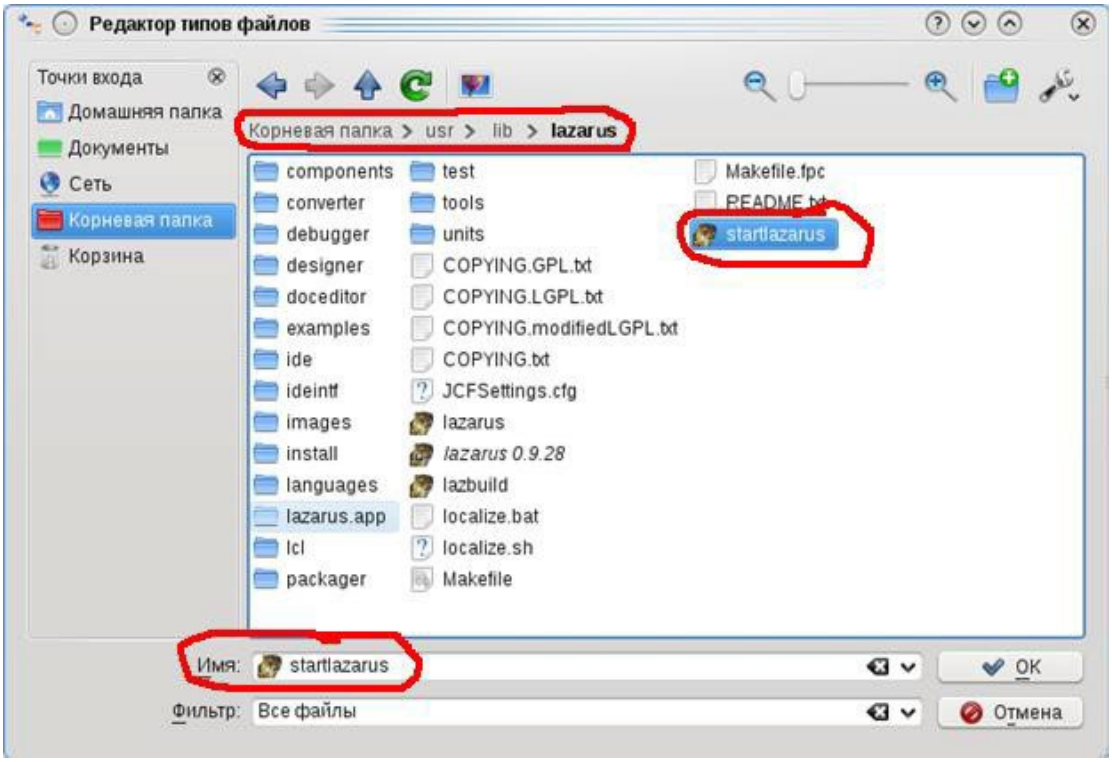
Мал. 2.41. Вікно властивостей файлу



Мал. 2.42. Зміна типу файлу



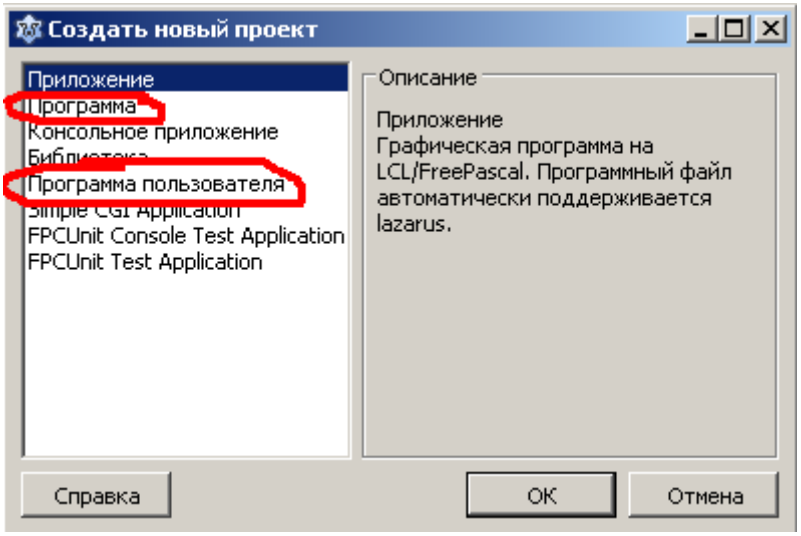
Мал. 2.43. Вікно вибору програми для відкриття файлу



Мал. 2.44. Вибір Lazarus для відкриття файлу проекту

2.1.12 Інші способи створення консольних програм

Lazarus надає інші способи створення консольних додатків. Розглянемо знову меню Проект-> Створити проект… рис. 2.45.



Мал. 2.45. Інші способи створення консольних програм

Консольну програму можна створити, вибравши пункти "Програма" та "Програма користувача".

При виборі "Програма" Lazarus зробить таку заготівлю:

```
program Project1;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}
begin
end.
```

Вихідний код програми буде автоматично підтримуватись Lazarus і буде збережений у файлі з розширенням .lpr.

При виборі "Програма користувача" Lazarus зробить таку заготовку:

```
program Project1;
{$mode objfpc}{$H+}
uses
  Classes, SysUtils
  { you can add units after this };
{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}
begin
end.
```

При цьому вихідний код програми буде збережено у файлі з розширенням .pas.

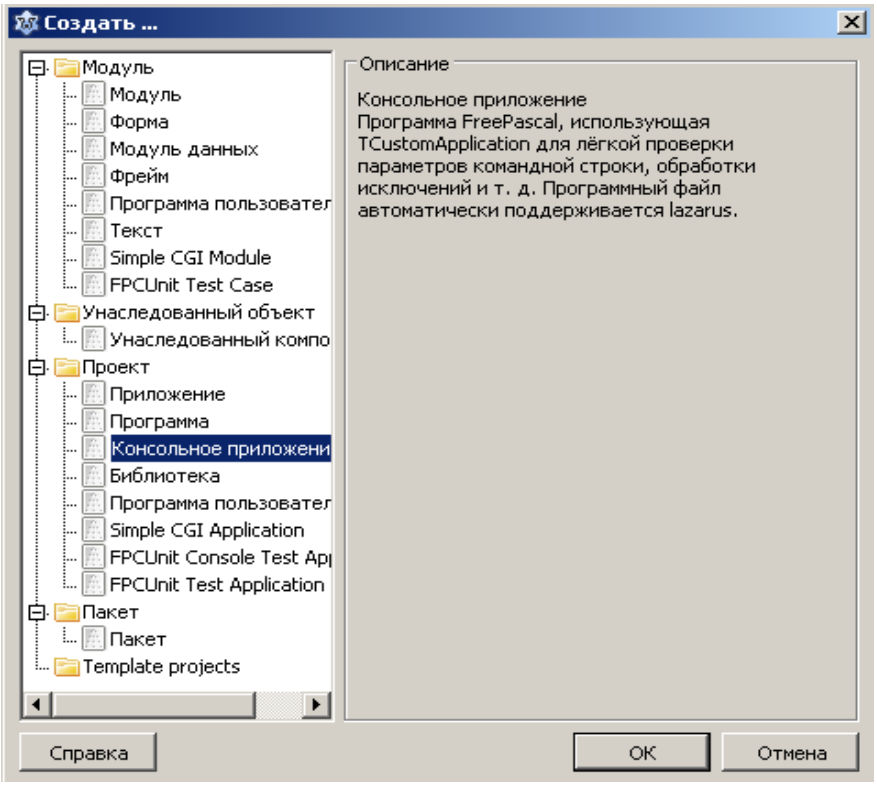
Порівнюючи ці три способи створення консольних додатків, можна зробити-

робити висновок, що "просунутим" способом буде перший спосіб (яким ми і скористалися). Щоправда, поки що ми лише "продемонстрували" свої наміри. Адже ми повністю замінили код, який нам пропонував Lazarus. Справа в тому, що заготівля програми, запропонована Lazarus, передбачає, що ми знаємо об'єктно-орієнтоване програмування (ООП) і багато чого, наприклад, обробку винятків. До цього нам ще далеко!

Але ми, вибравши пункт "Консольна програма" підтверджуємо свою рішучість "йти до кінця" – вивчити мову, вивчити ОВП і потім створювати консольні програми "як належить"!

А поки що, в нашій книзі, ми не робитимемо відмінностей у способах створення консольних додатків. Ви можете вибирати будь-який спосіб, який вам сподобається.

Насамкінець зазначимо, що створювати нові консольні проекти, а також безліч інших видів проектів та програмних об'єктів можна через меню Файл -> Створити…, рис. 2.46.



Мал. 2.46. Меню "Створити"

2.1.13 Типовий порожній проект

Як ви бачили з 2.1.10 після створення нового проекту, його потрібно налаштовувати. Зокрема, включити до проекту пакет LCL (див. рис. 2.24, 2.25), а Linux налаштувати параметри запуску (див. рис. 2.27, 2.28). При великій кількості проектів це напружує. У нашій книзі ми створюватимемо досить багато консольних додатків. Автор дуже сподівається, що ви, шановний читачу, всі приклади, наведені в книзі, скрупульозно виконуватимете на комп'ютері. Адже лише так можна навчитися програмувати! Навіть простий набір текстів програм у редакторі вихідного коду Lazarus дозволить вам набагато швидше освоїти та запам'ятати багато синтаксичних конструкцій мови Паскаль. Як було сказано в 1.1.3 одного читання та "розуміння" прикладів книги буде зовсім недостатньо!

Тому зручніше для виконання прикладів створити порожній проект з усіма налаштуваннями, про які говорилося вище і завжди, коли необхідно створювати новий проект, використовувати вже готовий порожній проект. Давайте вчинимо наступним чином:

1. Створіть консольну програму будь-яким зручним для вас способом.

2. Замість заготовки Lazarus введіть у вікні редактора вихідного коду наступний текст:

```
program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil, SysUtils;
begin
  {Вставте сюди вихідний код вашої програми}
```

```
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

3. Встановіть потрібні характеристики проекту, тобто. увімкніть у проект пакет LCL, у Linux налаштуйте параметри запуску.

4. Збережіть проект у папці Типовий порожній проект для консольних додатків. Не забувайте, що в Linux у назвах папок не допускаються пробіли, тому дайте цій папці ім'я без пробілів, наприклад:

"Типовий_порожній_проект_для_консольних_додатків".

Тепер, якщо вам необхідно створити новий проект, відкрийте цей порожній проект і збережіть його в іншій папці за допомогою меню
Файл-> Зберегти як… При цьому можна і потрібно зберегти проект під іншим ім'ям, що відображає характер завдання, що розв'язується. Ваш новий проект вже матиме необхідні нам налаштування.

2.1.14 Операції з цілими числами

Досі ми розглядали лише одну операцію з цілими числами – додавання. Природно, у Паскалі дозволені й інші операції. Розглянемо програму, що показує всі можливі операції з цілими числами:

приклад.

```
program int_operations;
uses Crt, FileUtil;
var
  A, B, C: integer;
begin
  writeln(UTF8ToConsole('Введіть два числа'));
```

```
  readln(A, B);
  writeln('A=',A,'B=',B); C:
  = A + B;
  writeln(UTF8ToConsole('Демонстрація додавання,
  C='),C); C: = A * B;
  writeln(UTF8ToConsole('Демонстрація множення, C='),C);
  C:=A div B;
  writeln(UTF8ToConsole('Демонстрація поділу націло, C = '), C);
  C:=A mod B;
  writeln(UTF8ToConsole('Залишок від поділу,
  C='),C); C:=A - B;
  writeln(UTF8ToConsole('Демонстрація віднімання,
  C='),C); writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

З цієї програми видно, що знак множення * (а чи не буква х). Тим самим уникають можливості сплутати його з літерою х.

Далі з цілими числами визначено операції:

div (від англійської divide – ділити) – розподіл націло. Нехай A = 17, B = 3, звідси 17:3 = 5 * 3 +2 і

A div B

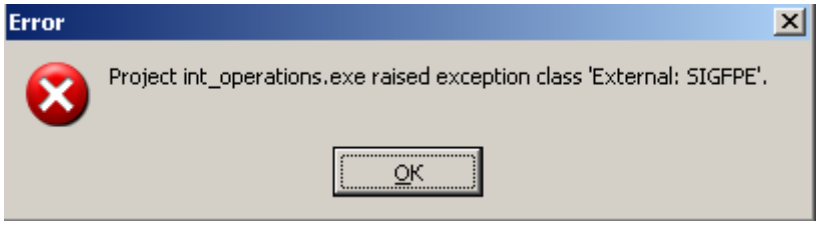
дає результат 5

mod (від англійського modulo – визначити залишок) – визначення залишку від поділу націло.

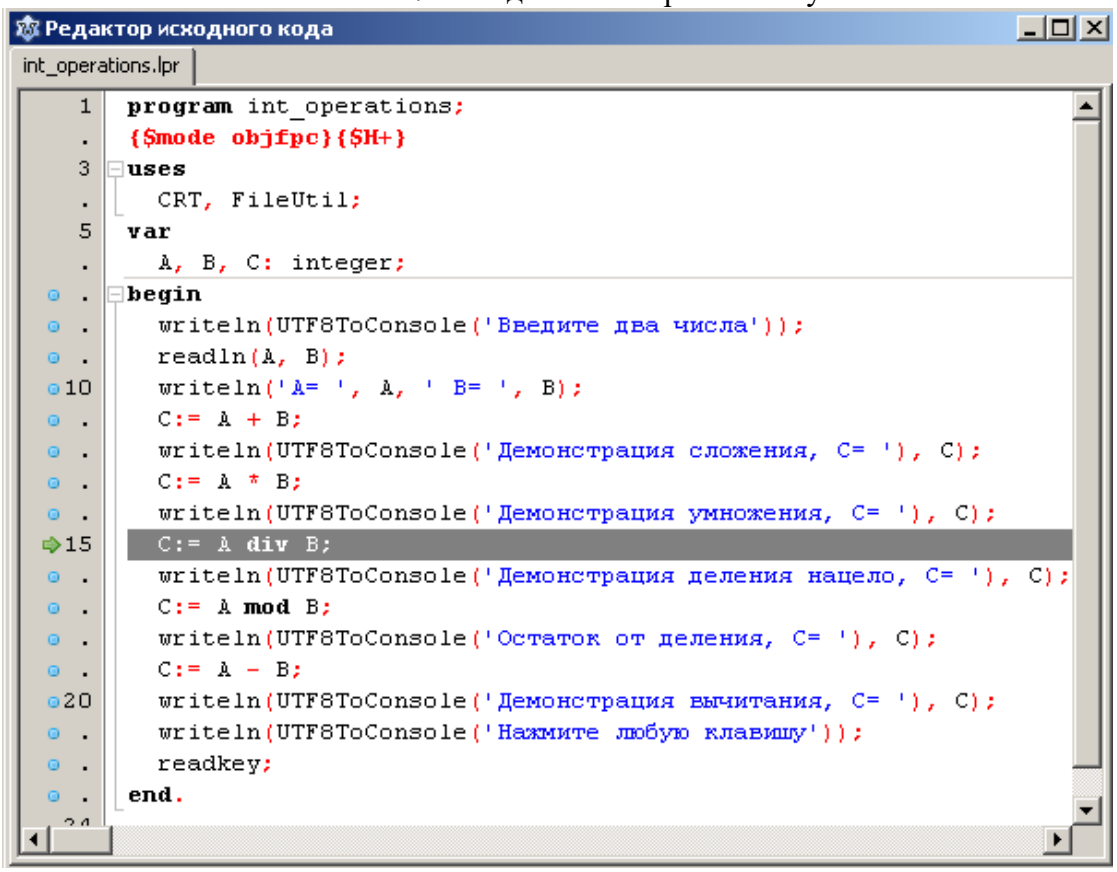
A mod B дає результат 2

Відкомпілюйте та виконайте свою програму (клавіша F9).

Все гаразд? Програма працює? У вас ще не виходило таке? (Рис. 2.47, 2.48)

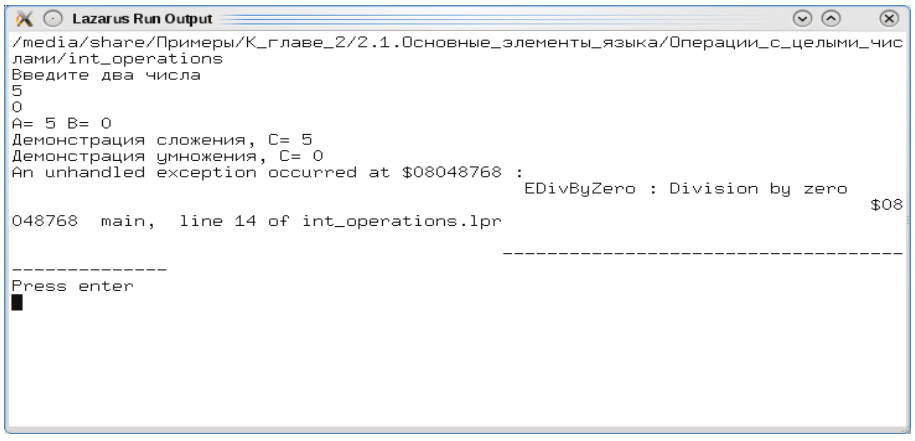


Мал. 2.47. Повідомлення про помилку



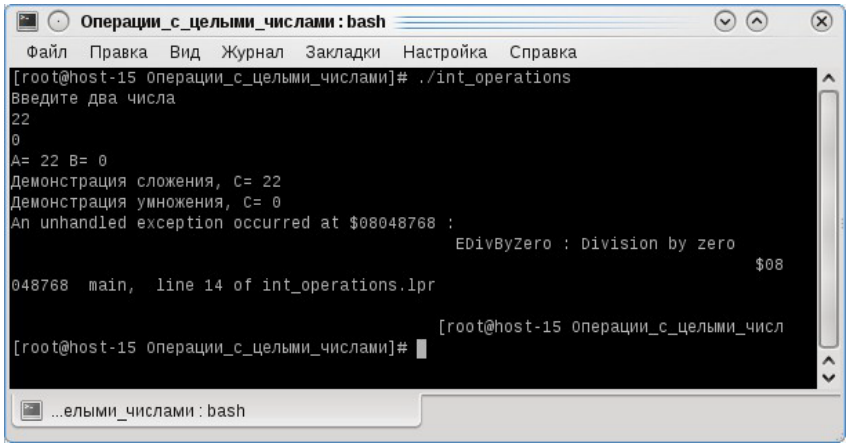
Мал. 2.48. Оператор, під час якого сталася помилка

У Linux вікно виводу буде таким:



Мал. 2.49. Повідомлення про помилку в терміналі

Або, якщо ви запускаєте програму в консолі, то вікно матиме вигляд:



Мал. 2.50. Повідомлення про помилку в консолі

Навіть якщо такого повідомлення не виходило, багато хто з читачів, напевно, вже здогадався, що ми в цій програмі не врахували. Так-так, ви абсолютно праві, у програмі є операція поділу. Адже користувач міг як друге число ввести нуль. Але ділити на нуль, як відомо, не можна!

Тут мені хотілося б знову відступити від конкретики і поміркувати на "загальні" теми.

2.1.15 Замість ліричного відступу 2

У процесі розробки програми багато програмістів-початківців зовсім не звертають уваги на такі, здавалося б, дрібниці. А дарма! Ваші програми повинні бути захищені від будь-яких мислимих і немислимих помилок або ненавмисних дій користувача. Здавалося б, у цілком очевидних ситуаціях, коли користувач за жодних обставин не повинен був би ввести число нуль, з тисячі користувачів знайдеться хоча б один, який обов'язково введе 0! Навіть якщо ви виведете яскраве, написане великим шрифтом попередження, що не можна вводити число 0. Тому є мільйон причин. Користувач міг у цей час задуматися про щось інше.

його могли в цей момент чимось відволікти, він міг просто не звернути уваги на ваше попередження або забути про нього. Зрештою він міг натиснути не на ту клавішу! Наприклад, цифра 9 на клавіатурі розташована поруч із цифрою 0. І, нарешті, обов'язково знайдуться такі користувачі, які захочуть подивитися, а що буде, якщо я все-таки введу нуль!

Ясно, що якщо у ваших програмах будуть зустрічатися такі "казуси", то вашому престижу як програміста буде завдано непоправної шкоди, особливо якщо ви пишете програми на комерційній основі, тобто. продаєте їх. Це також вплине на кількість продажів вашої програми.

Таким чином, контроль за такими "не передбаченими" діями користувача лежить на програмісті! Є таке поняття у програмуванні - писати програми, розраховані на "дурня" (fool-tolerance).

Тому любите свого користувача, поважайте його, дбайте про те, щоб йому було легко та комфортно працювати з вашою програмою (навіть є таке поняття дружність програми), але пишіть свої програми так, щоб вони були захищені від будь-яких ненавмисних, невмілих і навіть "неможливих" дій користувача.

Найчастіше такі помилки виникають при введенні користувачем якихось даних. Зі способами захисту своєї програми від таких ненавмисних дій користувача ми ознайомимося пізніше (див. розділ 2.1.25 та 6.3.7). Тут я просто загострив вашу увагу на цій проблемі, щоб ви завжди пам'ятали про це під час написання своїх програм.

2.1.16 Стандартні функції із цілими аргументами

Розглянемо програму:

```
program functions;
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil;
var
  a, b, c: integer;
begin
  a:=-2;
  b:= abs(a);
  writeln(UTF8ToConsole('Абсолютна величина числа a='),
  b); c: = sqr (b);
  writeln(UTF8ToConsole('Квадрат числа b='),
  c); c: = sqr (b + b);
  writeln(UTF8ToConsole('Квадрат числа (b + b)= '),
  c); writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Оператор b: = abs (a);привласнює змінною `b абсолютне значення числа a.`

Під абсолютним значенням числа розуміється значення цього числа, якщо відкинути знак.

```
abs(-2)= 2
abs(-10)= 10
abs(5)= 5
```

Оператор c:= sqr(b)- надає змінної c квадрат числа b, тобто. $c = b^2$, sqr (від англійської square – квадрат). Число в дужках називається аргументом функції. Як аргумент може бути вираз, наприклад, $\sqrt{b^2 - 4ac}$ запишеться на Паскалі так:

$sqrt(\text{sqr(b)} - 4 * a * c);$

Як зазначалося, в операторах write і writeln можна використовувати будь-які допустимі вирази, тобто. можна записувати так:

```
writeln('Квадрат числа (b+b)=', sqr(b+b));
```

2.1.17 Операції із речовими числами (тип real).

З речовими числами можна виконувати різні операції. Усі можливі операції ілюструються наступною програмою:

```
program real_numbers;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a, b, c: real;
begin
  a: = 17.3;
  b: = 3.4;
  c:= a * b;
  writeln(UTF8ToConsole('Умноження дійсних чисел c = '),
  c); c:= a/b;
  writeln(UTF8ToConsole('Ділення дійсних чисел c = '),
  c); c: = a + b;
  writeln(UTF8ToConsole('Складання дійсних чисел c = '),
  c); c:= a - b;
  writeln(UTF8ToConsole('Віднімання дійсних чисел c = '),
  c); writeln(UTF8ToConsole('Натисніть будь-яку клавішу'));
  readkey;
end.
```

2.1.18 Форматування висновку

При виведенні значень речового типу використовується так зване експоненційне уявлення, в якому використовується ступінь десяти. Ви це могли бачити під час виконання попереднього прикладу. Такий вид чисел на екран часто незручний. В операторах виводу можна використовувати форматування для вказівки ширини поля виводу. Вид оператора виводу в цьому випадку буде таким:

```
write(змінна_1:w:d, … змінна_n:w:d);
writeln(змінна_1:w:d, … змінна_n:w:d);
```

де w – загальна ширина поля виведення, d – кількість знаків після коми, тобто. дрібної частини числа. w та d повинні бути константами або виразами цілого типу. Щоб загальна ширина поля виводу визначалася автоматично, вказуйте w = 0. Наприклад:

```
writeln('a * b = ', c:0:2);
В цьому випадку на екран
буде виведено a * b = 58.82
замість
a * b = 5.8820000000000E+001
```

2.1.19 Одночасне використання речових та цілих чисел.

У програмі можуть зустрічатися змінні різних типів:

```
program int_real;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
```

```
  n, k: integer;
  a, b: real;
begin
  a: = 3.6;
  n: = 4;
  b:= n;
  writeln(UTF8ToConsole('Речовинна змінна b='), b);
  n:= trunc(a);
  writeln(UTF8ToConsole('Операція truncate n='),
  n); k:= round(a);
  writeln(UTF8ToConsole('Операція round k='), k);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

У програмі бачимо запис b:= n; де речової змінної b надається значення цілої змінної n. Крім того, в записах таких як b:= n + 4.6; або b: = 3 * 7.2 + n; зустрічаються речові та цілі числа, що стоять у правій частині виразу. Такі записи дозволені. Компілятор автоматично перетворює вираз у правій частині оператора присвоєння до речового типу. І навпаки, присвоєння речових значень цілим змінним просто заборонено. Тобто. якщо написати оператор присвоєння

```
n: = 3.14;
```

компілятор випустить помилку.

Для цього використовуються стандартні функції trunc та round. За допомогою функції trunc здійснюється перетворення речовинного числа в ціле шляхом відкидання всіх цифр, що стоять після десяткової точки (trun-

cate – усікати), round – дозволяє перетворити речове число в ціле шляхом округлення (round – округляти).

2.1.20 Інші стандартні функції з речовими аргументами

Таблиця 2.2

Функція	Запис на Паскалі
x^2	sqr(x)
$\sqrt{x}$s	sqrt(x)
in	sin(x)
xcos	cos(x)
x	arctan(x)
arctgx	ln(x)
ln x	exp(x)
e^x	

Ще раз нагадую, що аргументом функції (тобто те, що стоїть у дужках) може бути вираз. Наприклад,

```
s:= sin(a + b * 2/5.2 - sqr(a));
```

2.1.21 Булеви змінні

Булеві змінні (логічні змінні) – це змінні, що мають лише два значення false (брехня) та true (істина). Над булевими змінними визначено логічні операції not (логічне заперечення, "НЕ"), and (логічне "І"), or (логічне "АБО"), xor (що виключає "АБО").

Крім того, визначено такі операції відносини:

```
= одно,          <> не однаково,
менше,           > більше
```

<= менше чи одно, >= більше чи одно

Результатом цих операцій відносини є логічні false або true.

Розглянемо такий вираз:

x > 3

Залежно від значення x цей вислів буде або істинним (true), або хибним (false).

приклад.

```
program logic;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x: integer;
  flag: boolean;
begin
  x:= 4;
  flag: = x > 3;
  writeln('flag=', flag);
  flag:= x < 3;
  writeln('flag=', flag);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Допускається розміщувати праворуч і ліворуч від знаків відносин арифметичні вирази: x + 6.5 < x + 5 такі вирази називаються логічними або булевими виразами.

Розглянемо програму, де використовуються логічні функції:

```
program logic_1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x: integer;
  L1, L2, Result: boolean;
begin
  x:= 4; L1:
  = x > 3;
  L2: = x <3;
  writeln(UTF8ToConsole('Булева змінна L1='), L1);
  writeln(UTF8ToConsole('Булева змінна L2='), L2);
  Result := L1 AND L2;
  writeln(UTF8ToConsole('L1 AND L2 дорівнює '), Result);
  Result := L1 OR L2;
  writeln (UTF8ToConsole('L1 OR L2 дорівнює '), Result);
  Result := NOT Result;
  writeln (UTF8ToConsole('NOT Result одно'), Result);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

2.1.22 Умовні оператори.

Умовні оператори – це оператори, за допомогою яких можна змінювати послідовність виконання операторів програми.

2.1.22.1 Оператор if …. then

Цей оператор має вигляд:

```
if умова then
  оператор;
```

де оператор – будь-який оператор Паскаля, умова – логічний вираз.

Якщо ця умова виконується (тобто ця умова дає значення true), то буде виконано оператор, який стоїть після слова then.

Розглянемо приклад:

```
if X < 3 then
  writeln(X);
```

Тут записано: "Якщо X < 3 то вивести на екран значення X".

Операторif…then можна у вигляді структурної схеми, з допомогою якої показується хід виконання програми:

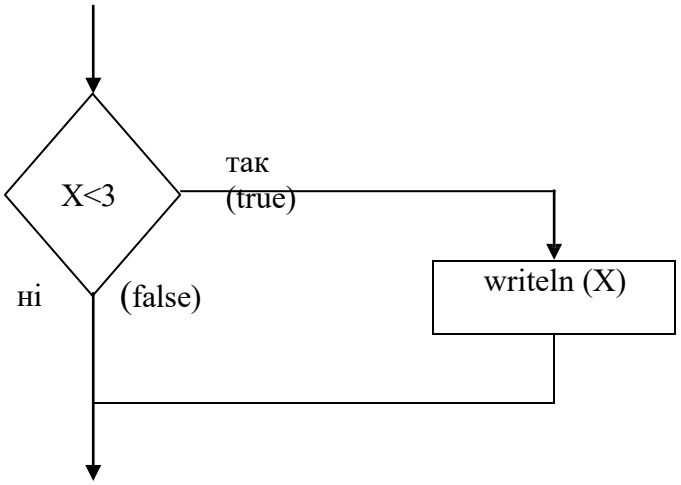


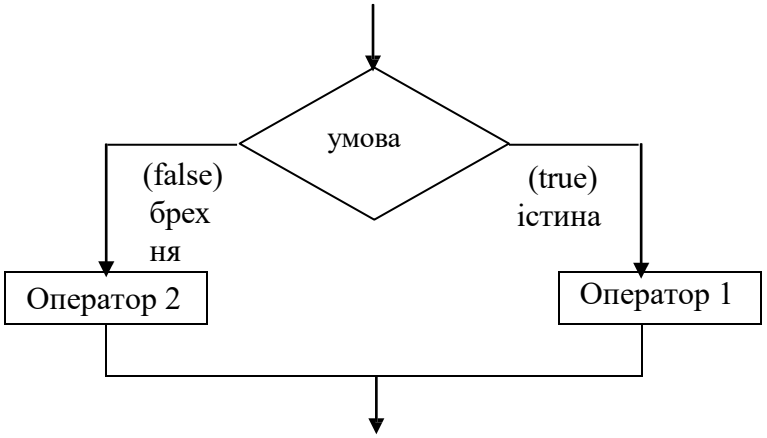
Рис.2.51. Блок-схема виконання умовного оператора if…then

2.1.22.2. Оператор if …then ... else

Цей оператор записується так:

```
if умова then
   оператор 1
else
   оператор 2;
```

Зверніть увагу, що перед else точка з комою (;) не ставиться. У цьому операторі, залежно від значення умови, виконуватиметься оператор 1, або оператор 2 , що ілюструє структурна схема.



Мал. 2.52. Блок-схема виконання умовного оператора if…then…else

З цієї структурної схеми видно, що вибирається чи один, чи інший варіант продовження програми.

Наприклад, після виконання оператора:

```
if X < 2 then
  X1:= 0
else
  X1: = 1;
```

змінної X1 буде надано значення 0, якщо X менше 2 або 1, якщо X більше чи одно 2.

приклад. Обчислити значення функції:

$$y=\begin{cases} x, \text{x} > 0 \\ 0, \ x = 0 \\ x^{2}, \text{x} < 0 \end{cases},$$

значення x введіть з клавіатури.

У наведених далі прикладах ми не складатимемо блок-схеми в таємній надії, що читач сам їх складе у разі потреби.

```
program ex1; {Варіант 1 - використання оператора if…then}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  X, Y: real;
begin
  writeln(UTF8ToConsole('Введіть значення X'));
  readln(X);
  if X > 0 then
    Y:= X;
  if X = 0 then
    Y:= 0;
  if X < 0 then
    Y:= SQR(X);
  writeln('X=', X:0:2, '; Y=', Y:0:2);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

```
program ex2; {Варіант 2 - використання оператора
if…then…else }
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil;
var
  X, Y: real;
begin
  writeln(UTF8ToConsole('Введіть значення X'));
  readln(X);
  if X > 0 then
    Y:= X
  else
  if X = 0 then
    Y:= 0
  else
    Y: = sqr (X);
  writeln('X=', X:0:2, '; Y=', Y:0:2);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Часто буває необхідно, щоб виконували або не виконували групу з кількох операторів. Для цього ці оператори поєднуються в блок, в якому перед першим оператором ставиться слово begin, а після останнього слово end. Усі оператори між begin та end утворюють так званий складовий оператор, а begin та end як би виконують роль дужок. Їх часто так і називають – операторні дужки.

Приклад:

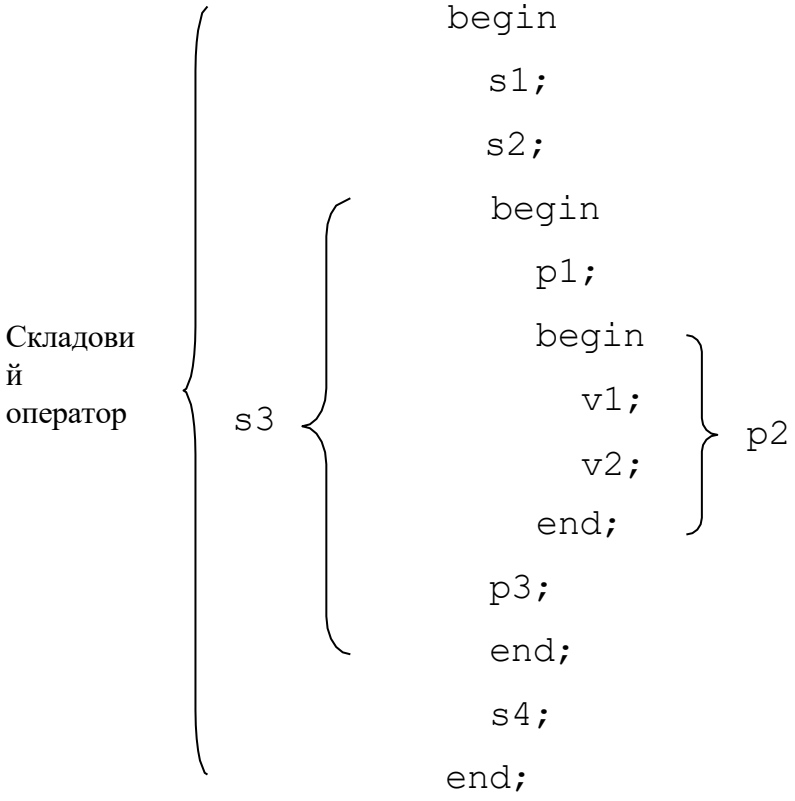
```
program demo; {Демонстрація застосування складеного
оператора}
{$mode objfpc}{$H+}
uses CRT, FileUtil;
```

```
var X: integer;
begin
  writeln(UTF8ToConsole ('Введіть значення X'));
  readln(X);
  if X < 2 then
  begin     //початок складеного оператора
     writeln(UTF8ToConsole(')Виконання програми за умовоютrue')); writeln('X =
     ', X);
  end       //  кінець складеного оператора
{ Складовий оператор вважається як би одним оператором,
тому перед else ; не ставиться}
  else
  begin     //початок складеного оператора
     writeln(UTF8ToConsole(')Виконання програми за умовою false')); writeln('X = ',
     X);
  end;      //кінець складеного оператора
  writeln(UTF8ToConsole('Натисніть будь-яку клавішу')); readkey;
end.
```

Ілюстрація у загальному вигляді: складовий оператор записується всередині службових слів begin та end:

```
  begin
    s1;
    s2;
    s3;
    s4;
  end;
```

де S1, S2, S3, S4 - оператори Паскаля, зокрема і складові оператори, тобто. Складові оператори можуть бути вкладені один в одного. Схематично це виглядає так:



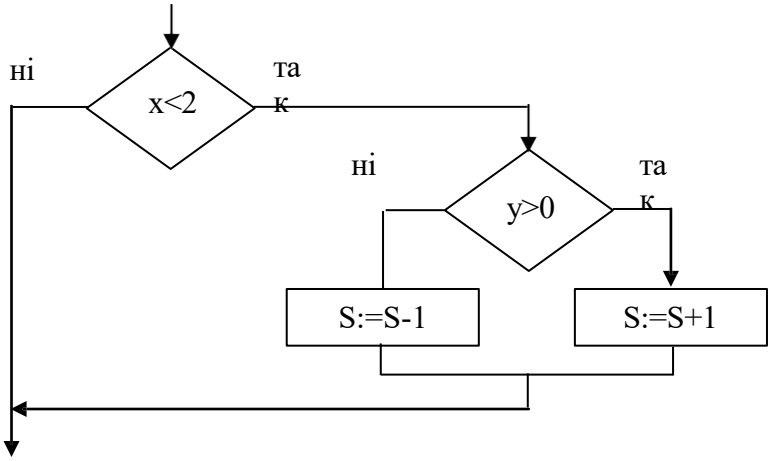
Мал. 2.53. Вкладені складові оператори

Тут s3 та p2 також складові оператори.

Умовні оператори також можуть бути вкладені один в одного.

```
if x < 2 then
  if y > 0
  then s: = s + 1
  else s: = s - 1;
```

Відповідну блок-схему наведено на рис. 2.54.



Мал. 2.54. Блок-схема виконання вкладених умовних операторів

2.1.23 Оператори циклу

Оператори циклу використовуються для організації багаторазового повторення виконання тих самих операторів. У мові Паскаль існує три типи операторів циклу.

2.1.23.1. Оператор циклу з передумовою

Цей оператор має вигляд:

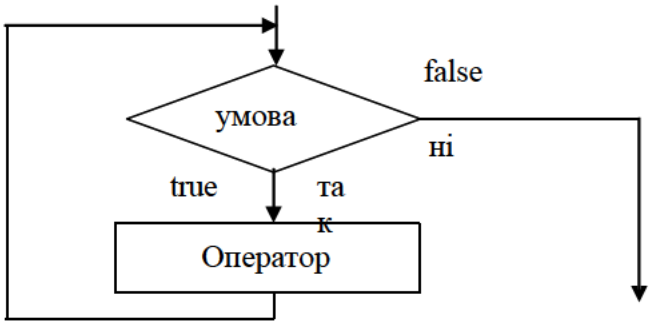
```
while умова do оператор;
```

де умова – булевський вираз, оператор – будь-який оператор Паскаля, зокрема може бути складовим оператором. Слова while і do є службовими словами, а оператор після do часто називають тілом циклу. Виконується цей оператор наступним чином: спочатку обчислюється значення булевого виразу. Якщо це значення є true, то виконується оператор після слова do і знову відбувається повернення до обчислення булевого виразу. Так повторюється, поки булеве вираз має значення true. Як тільки значення булевського виразу стане false, то відбувається вихід із циклу, тобто оператор після службового слова do вже не виконується, а буде вико-

нятися наступний після оператора циклу оператор.

У цьому операторі обчислення виразу відбувається раніше, ніж виконуватиметься оператор після do, тому і називається оператор циклу з передумовою. Може так статися, що оператор після do не буде виконаний взагалі, якщо значення умови з першого разу буде false.

Структурна схема циклу з передумовою



Мал. 2.55. Структурна схема циклу з передумовою

2.1.23.2. Оператор циклу з постумовою

Цей оператор має вигляд:

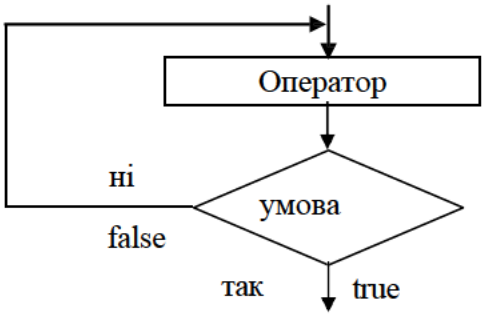
```
оператор оператора
until умова;
```

де оператор – будь-який оператор Паскаля, в тому числі і складовий, умова – булевський вираз.

`repeat і until – службові слова.`

Цей оператор виконується так: спочатку виконується оператор наступний за службовим словом repeat, потім обчислюється значення булевського виразу (умови). Якщо значення умови false, то відбувається повернення до виконання оператора і після цього знову обчислюється значення булевського виразу. Так повторюється доти, доки значення булевського виразу false. Як тільки умова стане true, виконання оператора циклу припиняється.

Структурна схема оператора



Мал. 2.56. Структурна схема циклу з постумовою

На відміну від оператора циклу while-do, тут оператор буде виконаний хоча б один раз, незалежно від значення умов.

Попередження! Щоб розглянуті вище оператори циклу виконувались кінцеве число разів, при побудові циклу необхідно передбачити, щоб серед операторів, що виконуються, обов'язково був оператор, який змінював би значення умови, таким чином, щоб коли-небудь значення умови приймало б false(для оператора while-do)або true(для оператора repeat-until).

В іншому випадку цикл повторюватиметься нескінченне число разів і програма "зациклиться". Відповідальність за правильне застосування цих операторів циклу несе програміст!

Приклад: Обчислити $S = \sum_{x=1}^{100} x$

```
program sum_1; {Варіант 1 цикл із передумовою}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var x, Sum: integer;
begin
```

```
  Sum: = 0; // у цій змінній накопичується сума
  x:= 1;
  while x <= 100 do
  begin
    Sum: = Sum + x;
    x: = x + 1;
  end;
  writeln('Sum=',Sum);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.

program sum_2; {Варіант 2 цикл із постумовою}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var x, Sum: integer;
begin
  Sum: = 0; // у цій змінній накопичується сума
  x:= 1;
  repeat
    Sum: = Sum + x;
    x: = x + 1;
  until x > 100;
  writeln('Sum=',Sum);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Приклад: Обчислитифункцію:

$$y=\begin{cases} x^2+1, \text{x}>0 \\ 0, \text{x}=0 \\ x^2-1, \text{x}<0 \end{cases}$$ для x $\in$[-10, 10] з кроком 1

Напишемо програму обчислення функції з використанням оператора if…then та оператора циклу while…do:

```
program fun_1; {Варіант 1}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  x:=-10;
  while x <= 10 do
  begin
    if x > 0 then
      y:= sqr(x) + 1;
    if x = 0 then
      y:= 0;
    if x < 0 then
      y:= sqr(x) - 1;
    writeln('x=', x, 'y=', y); x: =
    x + 1;
  end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey; end.
```

Напишемо програму обчислення цієї функції з використанням оператора if…then…else та оператора циклу while…do:

```
program fun_2; {Варіант 2}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  x:=-10;
  while x <= 10 do
  begin
    if x > 0 then
      y:= sqr(x) + 1
    else
    if x = 0 then
      y:= 0
    else
      y:= sqr(x) - 1;
    writeln('x=', x,'y=', y); x: = x
    + 1;
  end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

У програмах fun_1 та fun_2, як ви могли помітити, використовувалися складові оператори.

```
program fun_3;{Варіант 3 з використанням оператора циклу з
постусло- вом}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  x:=-10;
  repeat
  if x > 0 then y:=
    sqr(x) + 1;
  if x = 0 then
    y:= 0;
  if x < 0 then
    y:= sqr(x) - 1;
  writeln('x=', x, 'y=', y); x:
  = x + 1;
  until x > 10;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Зауважимо, що в операторі циклу з постумовою, якщо після слова repeat використовується не один, а кілька операторів, то не обов'язково використовувати операторні дужки begin та end, оскільки службові слова repeat та until самі виконують роль операторних дужок.

2.1.23.3. Оператор циклу із параметром.

Іноді заздалегідь відомо, скільки разів має бути виконана певна дія. Для завдань такого типу в мові Паскаль є оператор циклу з параметром. Цей оператор має вигляд:

```
for змінна:= вираз 1 to вираз 2 do оператор;
```

де for, to, do – службові слова; змінна- змінна цілого типу, звана індексом чи параметром циклу.

Вираз 1, вираз 2 – арифметичні вирази цілого типу, тобто. значення виразів мають бути цілими;

оператор – простий чи складовий оператор.

Для того, щоб оператор циклу виконувався хоча б один раз, значення виразу 1 має бути меншим або рівним значенню виразу 2 (на практикі значення виразу 1 завжди менше значення виразу 2). Оператор працює таким чином: спочатку змінній (параметру циклу) присвоюється значення виразу 1, потім порівнюється значення параметра циклу та значення виразу 2. Якщо параметр циклу менший від значення виразу 2, то виконується оператор після слова do. Потім параметр циклу збільшується на 1, після цього знову порівнюється значення параметра циклу та вираз 2, якщо параметр циклу менший, то знову виконується оператор після слова do. І так продовжується до тих пір, поки параметр циклу не стане більше виразу 2. Як тільки це відбувається, оператор циклу закінчується.

Приклад: давайте обчислимо знову значення функції

$$y=\begin{cases} x^2+1, & x>0 \\ 0, & x=0 \\ x^2-1, & x<0 \end{cases}$$ для x $\in$[-10, 10] з кроком 1

```
program fun_4; {оператор циклу з параметром for...to}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  for x:=-10 to 10 do
  begin
    if x > 0 then
      y:=sqr(x) + 1
    else
    if x= 0 then
      y:= 0
    else
      y:=sqr(x)-1;
    writeln('x=', x, 'y=', y); end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

2.1.23.4. Другий варіант оператора циклу з параметром

Оператор циклу з параметром може бути записаний і в такому вигляді:

```
для змінної:= вираз 1 downto вираз 2 do оператор;
```

У цьому варіанті параметр циклу (змінна) після кожного повторення не збільшується, а зменшується на 1. Значення виразу 1 більше або дорівнює (на практиці завжди більше) значення виразу 2.

Оператор циклу закінчується як тільки параметр циклу поменшає виразу 2.

Приклад:

```
program fun_5; {оператор циклу з параметром for...downto}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  for x:= 10 downto - 10 do
  begin
    if x > 0 then
      y:=sqr(x) + 1
    else
    if x = 0 then
      y:= 0
    else
      y:= sqr(x) - 1;
    writeln('x=', x, 'y=', y); end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Приклад: Обчислити $S = \sum_{x=1}^{10} x,$

```
program summa_1; {Варіант 1 використовується оператор циклу
for}
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil;
var
  x, s: integer;
begin
  s: = 0; // у цій змінній накопичується сума
  for x:= 1 to 10 do
    s:= s + x;
  writeln('x=', x, UTF8ToConsole(' сума s='), s);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
program summa_2; {Варіант 2 використовується оператор циклу
while…do}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, s: integer;
begin
  s: = 0; // у цій змінній накопичується сума
  x:= 1;
  while x <= 10 do
  begin
    s: = s + x;
    x: = x + 1;
  end;
  writeln('x=', x, UTF8ToConsole(' сума s='), s);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
```

```
end.
```

Як бачимо, під час використання оператора для запис циклу стає коротшим. Тому, якщо заздалегідь відомо число повторень циклу або його можна обчислити перед виконанням циклу, то краще використовувати цикл for.

2.1.24 Оператор вибору case

Для програмування розгалужень у алгоритмі найчастіше використовується умовний оператор if…then чи if…then…else. Однак якщо шляхів вибору багато, запис алгоритму за допомогою умовного оператора стає громіздким і важкооглядним. У таких випадках набагато зручніше використовувати оператор вибору case. Цей оператор має наступний синтаксис:

```
case <вираз> of
значення 1: оператор 1;
значення 2: оператор 2;
……
значення n: оператор n;
else
оператор;
end;
```

У цій конструкції оператори може бути складовими, <вираз> має бути порядкового типу, тобто. integer, char, boolean, перерахованого чи інтервального типу. Докладніше про типи даних ми поговоримо у 3.2.

Тип <значення> повинен збігатися з типом <вираз>, може бути одне або кілька, розділених комами, а також може являти собою деякий діапазон значень. Уся конструкція має завершуватися ключовим словом end.

Гілка else разом із оператором може бути відсутня.

Оператор працює таким чином:

1. обчислюється значення <вирази>.

2. виконується оператор, позначка якого <значення> збігається зі значенням <вираз>.

3. Якщо жодна <значення> не співпадає зі значенням <вираз>, виконується оператор після else.

Розглянемо застосування оператора вибору case з прикладу створення меню в консольних додатках.

приклад.

```
program case_menu;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  choose: integer;
begin
  {Формування меню}
  writeln(UTF8ToConsole('Виберіть потрібний режим
  роботи:')); writeln(UTF8ToConsole('Створення файлу
                                                    1'));wr
  iteln(UTF8ToConsole('Виведення вмісту файлу
                                                    2'));wri
  teln(UTF8ToConsole('Пошук записів по заданим полям
  3')); writeln(UTF8ToConsole('Додавання записів у файл
                                                    4'));wr
  iteln(UTF8ToConsole('Видалення записів із файлу 5'));
  writeln(UTF8ToConsole('Коригування записів у файлі 6'));
```

```
writeln(UTF8ToConsole('Вихід із програми                    7'));
repeat
```

```
    readln(choose);
    case choose of
    {choose – значення для вибору режиму роботи}
      1: writeln(UTF8ToConsole('Ви вибрали пункт меню 1'));
      2: writeln(UTF8ToConsole('Ви вибрали пункт меню 2'));
      3: writeln(UTF8ToConsole('Ви вибрали пункт меню 3'));
      4: writeln(UTF8ToConsole('Ви вибрали пункт меню 4'));
      5: writeln(UTF8ToConsole('Ви вибрали пункт меню 5'));
      6: writeln(UTF8ToConsole('Ви вибрали пункт меню
      6')); 7: writeln(UTF8ToConsole('Ви вибрали пункт
      меню 7' ));
    else
      writeln(UTF8ToConsole('Такого режиму
    немає')); end; {end of case}
  until choose = 7;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

2.1.25 Організація найпростішого контролю за введенням даних.

Повернемося тепер, наприклад, розібраному в розділі 2.1.14. "Операції з цілими числами", де ми обговорювали питання щодо контролю за введеними користувачем даних. Зараз у нас вже достатньо знань, щоб організувати найпростіший контроль даних, що вводяться. Для цього необхідно організувати цикл і, у разі введення помилкових даних, дати можливість користувачеві повторно запровадити необхідні дані. Програма у разі матиме вид: program int_operations_control;

```
{$mode objfpc}{$H+}
```

```
uses Crt, FileUtil;
var
  A, B, C: integer;
  zero: boolean;
begin
  zero: = true;
  writeln(UTF8ToConsole('Введіть два
  числа')); readln(A);
  repeat
    readln(B);
{Перевірка введеного числа B
    на нуль} if B = 0 then
    begin
      writeln(UTF8ToConsole('Друге число не може дорівнювати 0'));
      writeln(UTF8ToConsole('Введіть інше число не рівне
      0'));
    end
    else
    zero: = false;
  until not zero;
  writeln('A=',A,'B=',B); C:
  = A + B;
  writeln(UTF8ToConsole('Демонстрація додавання,
  C='),C); C: = A * B;
  writeln(UTF8ToConsole('Демонстрація множення,
  C='),C); C: = A div B;
  writeln(UTF8ToConsole('Демонстрація поділу націло,
  C='),C); C: = A mod B;
  writeln(UTF8ToConsole('Залишок від поділу,
  C='),C); C: = A - B;
```

```
  writeln(UTF8ToConsole('Демонстрація віднімання,
  C='),C); writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Іноді при виконанні циклу буває необхідно примусово завершити цикл, навіть якщо умова виходу з циклу не набула потрібного значення (для циклів while…do та repeat…until) або параметр циклу не досягнув потрібної величини (для циклу for). Для цього використовують інструкцію

```
break;
```

Також досить часто зустрічаються ситуації, коли необхідно розпочати цикл із початку, навіть якщо не всі оператори тіла циклу виконані. У цьому випадку використовують оператор

```
continue;
```

Вдосконалимо нашу програму. Справа в тому, що користувач може як друге число ввести не тільки число 0, а взагалі ввести неприпустимі символи. Наприклад, знаки + і - для цілих чисел допустимі, інші символи, зокрема і точка, неприпустимі. Те саме стосується і першого введеного числа.

Для організації контролю за введенням скористаємося спеціальною функцією IOResult. Для цього потрібно директивою компілятора відключити стандартний контроль вводу/виводу. Ця директива має вигляд {$I-}. Для відновлення стандартного режиму контролю введення/виведення необхідно використовувати директиву {$I+}.

Функція IOResult повертає 0, якщо операція введення/виведення завершилася успішно. Програма тепер матиме вигляд:

```
program int_operations;
{$mode objfpc}{$H+}
{$i-} // відключення стандартного режиму контролю введення
uses
  CRT, FileUtil;
var
  A, B, C: integer;
  error: boolean; {Булева змінна для контролю помилок
  введення}
 begin
  error:= false; {помилок при введенні першого числа немає}
  writeln(UTF8ToConsole('Введіть два числа'));
  {Перевірка на помилкове введення числа A}
  repeat
    readln(A);
    error:= (IOResult <> 0);
    if error then { якщо error= true, значить сталася
    помилка при вві-
де}
    writeln(UTF8ToConsole('Помилка! Введіть ціле число'));
  until not error;
  error:= false; {помилок при введенні другого числа немає}
  {Перевірка на помилкове введення числа B}
  repeat
    readln(B);
    error:= (IOResult <> 0);
    if error then
```

```
    begin
```

```
      writeln(UTF8ToConsole('Помилка! Введіть ціле
      число')); continue; { під час виконання цього
      оператора наступна перевірка числа на нуль
      зроблено нічого очікувати, виконання циклу
      почнеться початку, тобто. з оператора readln(B)
      }
    end;
    if B = 0 then
    begin
      writeln(UTF8ToConsole('Друге число не може
      дорівнювати 0')); writeln(UTF8ToConsole('Введіть інше
      число не рівне 0'));
    end
  until (B <> 0) and (not error);
  {$+} { відновлення стандартного режиму контролю
  введення/виведення }
  writeln('A=',A,'B=',B); C: =
  A + B;
  writeln(UTF8ToConsole('Демонстрація додавання, C='),
  C); C: = A * B;
  writeln(UTF8ToConsole('Демонстрація множення, C='), C);
  C: = A div B;
  writeln(UTF8ToConsole('Демонстрація поділу націло, C = '), C); C:
  = A mod B;
  writeln(UTF8ToConsole('Залишок від поділу, C='),
  C); C: = A - B;
  writeln(UTF8ToConsole('Демонстрація віднімання,
  C='),C); writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Зрозуміло, розглянутий метод контролю за введенням/висновком не є єдиним. Існують більш надійні та загальні методи контролю, про кото-

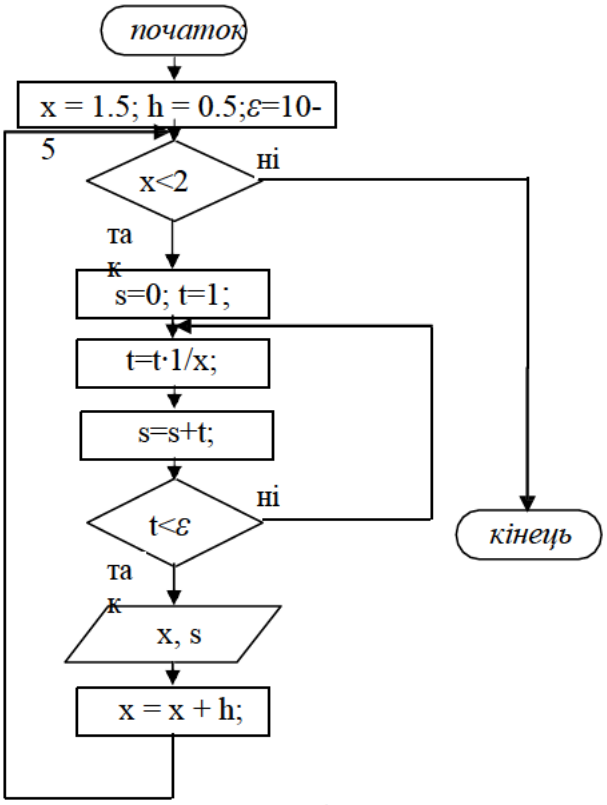
ми будемо говорити в розділі 6, зокрема в розділі, присвяченому виняткам.

2.1.26 Обчислення сум схожих рядів

Обчислити суму ряду $S = \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + \cdots + \frac{1}{x^n} + \ldots,$

$x \in [1.5, 2]$ з кроком h = 0.5, точність $\varepsilon = 10^{-5}$

Складемо блок-схему алгоритму:



Мал. 2.57. Алгоритм обчислення суми ряду

З цієї блок-схеми ми бачимо, що цикли можуть бути вкладені один в інші, так званий цикл у циклі. Внутрішній цикл обчислює суму ряду поточного значення змінної x. Зовнішній цикл змінює значення x, після цього знову починає свою роботу внутрішній цикл, який обчислює суму ряду для іншого значення x. Так відбувається доти, доки не будуть перебрані всі значення x.

Запис s=s+t слід у " динаміці " , тобто. до поточного значення суми s додається черговий член ряду t і отримане значення знову запам'ятовується у змінній s. Таким чином, у змінній s накопичується сума ряду.

Тепер що означає обчислити суму ряду з точністю ε. Це означає, що ми округляємо обчислену суму ряду з числом знаків після коми, вказаної в ε. Тобто, якщо черговий член ряду виявиться меншим за ε (взагалі кажучи, за абсолютною величиною), то додавання цього члена до суми не вплине на результат і не позначиться на підсумковій сумі, оскільки ми його округляємо із заданою точністю ε.

Всі ці міркування вірні для рядів, що сходяться, обчислення суми розбіжного ряду не має сенсу.

Розглянемо у блок-схемі фрагмент, де обчислюється черговий член ряду *t*. Багато хто тут припускається типової помилки. Для обчислення xn використовують

формулу $xn = en^{\ln(x)}$. Так можна обчислювати ступінь, але за такого способу ваша програма буде виконуватися за часом на порядок довше, ніж за способу, застосованого нами. Адже використовуючи наведену вище формулу, ви змушуєте комп'ютер обчислювати xn при кожному новому n як би заново, з нуля. Але придивіться уважно до ряду. Щоб вирахувати черговий член ряду треба

попередній член ряду помножити на $\frac{1}{x}$. І все!

Хоча на сучасних комп'ютерах різниця в часі виконання для таких простих програм практично не помітна, привчайтесь писати програми з оптимальним кодом! У разі оптимальність полягає у зменшенні кількості арифметичних операцій для обчислення чергового члена низки, отже виграш у часі виконання програми.

```
program summ;
{$mode objfpc}{$H+}
uses
```

```
  CRT, FileUtil;
var
  x, s, h, eps, t: real;
begin
  x: = 1.5;
  h: = 0.5;
  eps: = 0.0000000001;
  while x <= 2 do
  begin
    s: = 0;
    t: = 1;
    repeat
      t:= t * 1/x;
      s:= s + t;
    until t <= eps;
    writeln('X=', X:0:2, 'S=', S:0:5);
    x: = x + h;
  end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Обчислити значення функції`sinx` використовуючи його розкладання в ряд Тейлора:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots.$$

$x \in [0,1]$, значення x задані у радіанах, h=0.2, точність $\varepsilon = 10^{-7}$

Вивести також "точне" значення функції, обчислене за допомогою стандартної вбудованої функції Паскаля sin(x)

Складання блок-схеми алгоритму надається самому читачеві.

Тут складності може викликати лише процес організації обчислень визначення чергового члена низки по вже обчисленому попередньому члену. Уважно проаналізуйте ряд, увімкніть "міркування" і ви легко придумаєте, як це зробити. Рекомендую також не поспішати дивитися програму, а спробувати самому написати та налагодити її!

```
program sinus;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y, s, h, eps, t: real;
  n: integer;
begin
  x: = 0;
  h: = 0.2;
  eps:= 1e-7;
  while x <= 1 do
  begin
    s: = x;
    t: = x;
    n: = 2;
    repeat
      t:=-t * sqr(x)/(n * (n + 1));
      s: = s + t;
      n: = n + 2;
    until abs(t) <eps;
    y:= sin(x);
    writeln('x=', x:0:2,
```

```
            UTF8ToConsole('Наближене значення синуса = '),
            s:0:7, UTF8ToConsole('Точне значення синуса =
            '), y:0:7);
    x: = x + h;
  end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Як перевірити, чи правильні результати видає програма чи ні? Можна використовувати так звані модельні вихідні дані, при яких апріорі відомо, якими мають бути результати. Для синуса, наприклад, відомо, що $\sin 0^\circ = 0$, $\sin \frac{\pi}{2} = 1$. Якщо у цих точках ваша програма видає правильні результати, можна з певною часткою впевненості вважати, що ваша програма працює правильно.

Є й інший спосіб – порівняти результати, що видаються вашою програмою з іншою програмою, яка вирішує те саме завдання.

У нашому випадку ми використовували вбудовану функцію sin(x). Важливо розуміти, що цю функцію теж писали люди, програмісти, які розробили Free Pascal. Швидше за все, вони також використовували розкладання функції sinx у ряд. І в нас немає жодних підстав не довіряти їм! Тому вважатимемо, що вбудована функція sin(x) "точна" і якщо наша програма видає ті ж результати, то можна вважати, що програма працює правильно.

2.2. Реалізація деяких алгоритмів розділу 1.

Настав час для реалізації алгоритмів розібраних у розділі 1, у розділах 1.1 та 1.3, крім програми розв'язання квадратного рівняння, яке, сподіваюся, ви вже давно самі написали.

2.2.1 Програма вирішення задачі про поїзди та муху

```
program mukha;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var d, v, v1, v2, x, y, s, t: real;
    F: boolean;
begin
  {Блок визначення вихідних даних}
  {Можна замінити введенням їх із
   клавіатури. Тоді програма стане
   більш універсальною,
   у тому сенсі, що можна задавати різні відстані та швидкості d:= 600;
  v: = 200;
  v1: = 40; v2: = 60;
  y:= d;
  s: = 0;
  F: = false; // спочатку йдемо правою гілкою алгоритму
  while y > 1e-2 do
  begin
    if not F then
```

```
    begin // це права гілка алгоритму
      F: = true;
      t:= y/(v + v1);
    end
    else
    begin // це ліва гілка
      F: = false;
      t:= y/(v + v2);
    end;
    x:= t * v;
    s:= s + x;
    y:= y - t * (v1 + v2);
    writeln('x=', x:0:4, 's=   ', s: 0:2);
  end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

2.2.2 Програма обчислення певного інтегралу

Напишемо програму обчислення інтегралу

$$\int_0^{\frac{\pi}{2}} \sin x dx$$

за формулою Сімпсона шляхом подвійного перерахунку. Нагадаємо, що блок-схему обчислення інтегралу ми розглядали у 1.3.2.

```
program integral;
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil;
var
  a, b, h, x, s, s1, eps: real;
  n, k: integer;
begin
 {задаємо інтервал, на якому обчислюється
  інтеграл} a: = 0;
  b: = pi/2;
  k: = 0; // за першому обчисленні інтеграла k=0
  eps:= 1e-5; // задана точність обчислення інтегралу
  n: = 4; // Початкова кількість точок розбиття інтервалу
  (a, b)
  h:=(b - a)/n; // крок обчислення підінтегральної функції
  while true do
  begin
    x:= a;
    x: = x + h;
    s: = 0;
    while x <(b - h) do
    begin
      s: = s + sin (x) + 2 * sin (x +
      h); x: = x + 2 * h;
    end;
    s: = 2 * s;
    s:= (h/3) * (sin(a) + 2 * sin(b) + s);
    if k = 0
    then
    begin
      k:= 1;
```

```
      s1: = s;
      h:= h/2;
      continue;
    end
    else
    if abs(s1 - s) > eps then
    begin
      s1: = s;
      h:= h/2;
      continue;
    end
    else
      break;
  end;
  writeln(UTF8ToConsole('Значення інтеграла s='),
  s:0:4); writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

При заданих межах інтегрування значення інтеграла s= 1.0000 Перевіримо правильність одержаних результатів прямим обчисленням інтеграла.

$$\int_0^{\frac{\pi}{2}} \sin x dx = -\cos\Big|_0^{\frac{\pi}{2}} = -(\cos(\frac{\pi}{2}) - \cos(0)) = (0 - 1) - 1$$

Змінимо верхню межу інтегрування на b=π , для цього у програмі змініть оператор

```
b: = pi/2;
```

на

```
b: = pi;
```

Здійсніть новий прогін програми. Отримаємо відповідь s= 2.0000

Знову перевіримо прямим обчисленням інтеграла:

$$\int_0^{\pi} \sin x dx = -\cos\Big|_0^{\pi} = -(\cos(\pi) - \cos(0)) = -(-1 - 1) = 2$$

І, нарешті, спробуємо зробити верхню межу інтегрування b= $\frac{\pi}{4}$

Програма видасть відповідь s=0.2929, знову перевіримо:

$$\int_0^{\frac{\pi}{4}} \sin x dx = -\cos\Big|_0^{\frac{\pi}{4}} = -(\cos(\frac{\pi}{4}) - \cos(0)) = -(\frac{\sqrt{2}}{2} - 1) = 0.70710 - 1 = 0.2929$$

Таким чином ми можемо сміливо стверджувати, що наша програма правильно обчислює цей інтеграл.

Розділ 3 Більш складні елементи мови

3.1. Загальна структура Паскаль – програми

Загальна структура Паскаль – програми має вигляд:

```
program < Ім'я програми >;
label
   <Мітки>;
const
   <Константи>; type
   <Нові типи даних>; var
   <Опис змінних>;
   <процедури>;
   < Функції >;
begin
   <Тіло головної програми>;
end.
```

З основними елементами цієї структури ми знайомі. Нами залишилися не розглянутими <Мітки>,<Процедури>,<Функції>.

Мітки потрібні для організації переходу на якийсь оператор. Для цього оператор позначається міткою. Оператор переходу має вигляд:

```
goto <Мітка>;
```

та дозволяє передати керування оператору з міткою <Мітка>.

Програма, що містить мітки, зазвичай погано читається і, як правило, є не структурованою. Цілком можна писати програми взагалі не використовуючи мітки. Отже, забули про мітки та поїхали далі!

3.1.1 Процедури та функції

При розробці великих програм майже завжди з'являються фрагменти коду, що часто повторюються. Щоб не повторювати ці фрагменти в різних частинах програми, їх можна записати один раз, присвоїти їм якісь імена та використовувати їх у потрібних місцях програми. Такі іменовані фрагменти коду називають підпрограмами.

Підпрограми поділяються на процедури та функції. Текст процедури або функції записується у розділі описів, після опису змінних. Надалі для того, щоб використовувати цю процедуру або функцію, достатньо вказати її ім'я. У деяких випадках процедура (функція) використовує деякі значення, які мають передаватися з головної програми або інших процедур (функцій). Ці значення називають параметрами. Параметри вказуються у заголовку процедури (функції) у дужках. Вказується ім'я змінної та через двокрапку тип змінної. Якщо змінних одного типу кілька, то вони поділяються комами. Параметри різних типів поділяються крапкою з комою.

Відмінність функції від процедури в тому, що функція обов'язково повинна повернути деяке обчислене значення програмі, що викликає.

3.1.1.1 Структура процедури

```
procedure <ім'я
процедури>[(параметри)]; label
   <Мітки>;
```

```
const
   <Константи>; type
   <Нові типи даних>; var
   <   Опис   локальних змінних >;
   <   Опис   внутрішніх процедур >;
   <   Опис   внутрішніх функцій >;
begin
   <Тіло процедури>; end;
```

Параметри, зазначені в заголовку, називаються формальними параметрами. Для виклику процедури необхідно просто вказати її ім'я та параметри (якщо вони є). Параметри, вказані під час виклику процедури, називаються фактичними параметрами.

3.1.1.2. Структура функції

```
function <ім'я функції> [(параметри)]: тип
функції; label
   <Мітки>;
const
   <Константи>; type
   <Нові типи даних>; var
   <   Опис   локальних змінних >;
   <   Опис   внутрішніх процедур >;
   <   Опис   внутрішніх функцій >;
```

```
begin
   <Тіло функції>;
end;
```

У заголовку функції вказується її ім'я, параметри (якщо вони є) та тип значення, яку функція має повернути. Для виклику функції необхідно вказати її ім'я та фактичні параметри (за наявності). Причому ім'я функції можна вказувати у виразах!

У тілі функції повинен бути хоча б один оператор, який надає імені функції деяке значення. В іншому випадку значення функції залишається невизначеним, що може призвести до непередбачуваних наслідків. При обчисленні виразів ім'я функції заміщається обчисленим значенням.

Останнім часом входить у моду надавати значення всередині функції не імені функції, а системної змінної Result.

Функція або процедура може викликати інші функції.
або процедури, причому може викликати і саму себе! Рівень вкладеності у своїй не обмежений. Хоча на практиці намагаються уникати надто глибоко вкладених функцій (процедур), оскільки це збільшує ймовірність появи помилок, які важко знайти, і, крім того, погіршує читабельність і розуміння логіки роботи програми.

3.1.1.3 Глобальні та локальні змінні

Зазвичай будь-яка програма, функція чи процедура використовує якісь змінні, константи, функції та процедури. У програмі будь-який об'єкт перед її використанням має бути описаний у розділі описів. При цьому кожна змінна, а також константи, функції або процедури має свою область видимості, де вони можуть використовуватися. Якщо об'єкт описано в підпрограмі, то доступ до неї з програми, що викликає, неможливий. Цей об'єкт є локальним стосовно підпрограми, де він описаний, тобто. "не ві-

дим" у викликаючій програмі. Такі змінні, константи, функції та процедури називаються локальними. У той же час, об'єкти, описані у викликаючій програмі, доступні підпрограмам, що викликаються, тобто "видимі" ім. Такі об'єкти називаються глобальними.

Розглянемо приклад, щоб глибше зрозуміти сказане вище:

```
program console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x: integer; // Глобальна змінна
procedure local_global; // процедура без параметрів
begin
  x:= 25; // Глобальна змінна x доступна процедурі
  writeln(UTF8ToConsole('Значення змінної x'));
  writeln(UTF8ToConsole('всередині процедури = '), x);
end; // кінець процедури
begin // початок головної програми
  x: = 1;
  writeln(UTF8ToConsole('Глобальна змінна x'));
  writeln(UTF8ToConsole('до виклику процедури = '),
  x); local_global; // виклик процедури
  writeln(UTF8ToConsole('Глобальна змінна x'));
  writeln(UTF8ToConsole('після виклику процедури = '),
  x); writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Як бачимо, процедура змінила значення глобальної змінної. Слід пам'ятати, що змінна, що є локальної у цій підпрограмі, стає глобальною всім підпрограм нижчого рівня. Розглянемо приклад:

```
program console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x: integer; // Глобальна змінна
procedure local_global; // процедура без параметрів
var
  y: integer; // Локальна змінна
procedure A; // Процедура всередині процедури local_global
begin
  y:= 100; // Всередині процедури A змінна y є глобальною
  writeln(UTF8ToConsole('У процедурі A y='), y);
end; // кінець вкладеної процедури
begin
  x:= 25; y:= 0;
  A; // виклик процедури нижчого рівня
  writeln(UTF8ToConsole('Після виклику процедури A y='),
y); end; // кінець процедури
begin // початок головної програми
  x:= 1;
  writeln(UTF8ToConsole('Глобальна змінна x'));
  writeln(UTF8ToConsole('до виклику процедури дорівнює'),
  x);
```

```
  local_global; // виклик процедури
  // y: = 1; Якщо видалити коментар, компілятор видасть
  помилку,
  // A; т.к. локальні об'єкти невидимі для головної
  програми writeln(UTF8ToConsole('Глобальна змінна
  x')); writeln(UTF8ToConsole('після виклику процедури
  дорівнює'), x); writeln(UTF8ToConsole('Натисніть
  будь-яку клавішу')); readkey;
end.
```

Змінна y є локальною у процедурі local_global, у той час вона доступна процедурі A, тобто. для неї вона є глобальною.

Зверніть увагу, що локальні об'єкти недоступні програмі, що викликає. Таким чином, з головної програми не можна викликати процедуру A та не можна змінити значення змінної y. Головна програма просто не знає про їхнє існування! Процедура local_global виступає для головної програми "чорною скринькою". Для обміну даними між програмою, що викликає, і підпрограмою не варто використовувати глобальні змінні. Для цього використовується механізм передачі параметрів, про який йтиметься у наступному розділі.

Використання глобальних змінних може призвести до важких помилок, оскільки, коли підпрограма змінює значення глобальної змінної, головна програма про це може і "не знати". Підпрограма має бути незалежною і самодостатньою ("чорна скринька"!) у тому сенсі, що повинна вирішувати своє завдання виключно "своїми силами", лише за необхідності отримуючи від викликаючої програми дані через параметри і через параметри передаючи результати своєї роботи. Якщо це функція, вона повинна повертати єдине значення через своє ім'я.

Чому локальні змінні "невидимі" програмі, що викликає? Річ у тім, що локальним змінним пам'ять розподіляється інакше, ніж

Світовим змінним. Пам'ять для глобальних змінних виділяється статично компілятором в області пам'яті, яка називається сегментом даних або статичною пам'яттю. Глобальні змінні "живуть" у статичній пам'яті від запуску головної програми та аж до її закриття. Тому вони доступні для всіх підпрограм. Для локальних змінних створюється спеціальна область пам'яті, яка називається стеком. Параметри підпрограми також розміщуються у стеку. Після завершення роботи підпрограми пам'ять відведена для неї автоматично звільняється і може бути розподілена вже для іншої підпрограми.

Є ще один тип змінних, які по суті є локальними, але пам'ять виділяється в сегменті даних. Це так звані статичні змінні. Той факт, що вони знаходяться в сегменті даних, означає, що такі змінні після виходу з підпрограми не знищуються. Таким чином, якщо підпрограма буде викликана повторно, значення статичної змінної збережеться і її можна буде використовувати.

У Паскалі статичною змінною є так звана типізована константа. Її опис має вигляд:

```
const
  Константа: Тип = значення;
```

На відміну від звичайної константи у цьому, що вказується тип і для таких констант виділяється пам'ять. Значення типізованої константи в підпрограмі можна змінювати (тільки не в Delphi! Там типізована константа є "справжньою" константою).

приклад.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
```

```
procedure static_var; // процедура
const x: integer = 0;
begin
  writeln(UTF8ToConsole('До зміни x='), x); x:=
  x + 25;
  writeln(UTF8ToConsole('Після зміни x='), x);
end; // кінець процедури
begin
  writeln(UTF8ToConsole('Зміна статичної змінної x'));
  writeln(UTF8ToConsole('всередині процедури після першого
  виклику')); static_var;
  writeln(UTF8ToConsole('Зміна статичної змінної x'));
  writeln(UTF8ToConsole('всередині процедури після другого
  виклику')); static_var;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Після першого виклику процедури значення змінної x дорівнюватиме 25. Після другого виклику x=50, тобто. статична змінна зберегла значення 25 після першого виклику процедури.

Імена локальних змінних у підпрограмі можуть збігатися з глобальними іменами програми, що викликає. У цьому випадку при вході в підпрограму доступна лише локальна змінна, а глобальна стає недоступною, наприклад:

```
program console_app;
{$mode objfpc}{$H+}
uses
```

```
  CRT, FileUtil;
var
x: integer; // Глобальна змінна
procedure local_global; // процедура без параметрів
var
  x: integer; // Локальна змінна з тим самим ім'ям
begin
  x:= 25;
  writeln(UTF8ToConsole('Локальна змінна x='), x);
end; // кінець процедури
begin // початок головної програми
  x:= 1;
  writeln(UTF8ToConsole('Значення глобальної змінної
  x)); writeln(UTF8ToConsole(' до виклику процедури'),
  x); local_global; // виклик процедури
  writeln(UTF8ToConsole('Значення глобальної змінної
  x)); writeln(UTF8ToConsole(' після виклику
  процедури'), x); writeln(UTF8ToConsole('Натисніть
  будь-яку клавішу')); readkey;
end.
```

Ми бачимо, що значення глобальної змінної не змінилося. Локальна змінна “перекрила” глобальну та оператор присвоювання

x:= 25;

у процедурі привласнило значення 25 зовсім інший змінної, хоч і з тим самим ім'ям x.

Таким чином, однойменні глобальні та локальні змінні – це різні змінні. Будь-яке звернення до таких змінних у тілі підпрограми трактується як звернення до локальних змінних, тобто глобальні змінні в цьому випадку просто недоступні.

Приклад використання функції.

Обчислити значення y $= \sin x$ шляхом розкладання sin x в ряд Тейлора з точністю $\varepsilon = 10^{-3}$, $x \in [0,1]$, $\Delta x = 0.1$ Обчислення оформити як функції. Обчислене значення sin x для кожного x порівняти з "точним" значенням путим застосування стандартної функції Паскаля sin(x).

Нагадаю, що ми вже писали цю програму (див. 2.1.26), але без застосування функцій. Розкладання sin x у ряд Тейлора має вигляд:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

функцію, яка обчислює *sinx* по даному ряду, назвемо `No_standard_sin(x)`.

```
program sinus_fun;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y, y1, dx:real;// глобальні змінні, доступні функції
function No_standard_sin(x:real): real;
var
  eps, s, t: real; // локальні змінні, недоступні
  n: integer;     // викликаючу програму
begin
  s: = x;
  t: = x;
```

```
  n: = 2;
  eps:= 1e-7;
  repeat
    t: = -t * sqr (x) / (n * (n +
    1)); s: = s + t;
    n: = n + 2;
  until abs(t) <eps;
  No_standard_sin:=s; // Повернення значення функції
  //Result:= s; // можна і так
end;
begin
  x: = 0;
  dx: = 0.2;
  writeln(UTF8ToConsole('Значення синуса'));
  writeln(UTF8ToConsole('моя функції, стандартна
  функції')); while x <= 1 do
  begin
    y:= No_standard_sin(x);// виклик моєї
    функції y1:= sin(x); // Виклик стандартної
    функції writeln('', y:0:7,'',    y1:0:7);
    x: = x + dx;
  end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

приклад.

Згадаймо приклад з 2.2.2. обчислення інтеграла за формулою Сімпсона. Перепишемо програму, причому оформимо обчислення підінтегральної функції.

ції як функції, а обчислення інтеграла як процедури:

```
program integral;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a, b, eps: real;

функція Fx(x: real): real; // Підінтегральна функція
{Якщо потрібно обчислити інтеграл для іншої
функції, достатньо поміняти лише один оператор}
begin
  Fx: = sin (x); // Для іншої функції змінюємо тут
end;

{Процедура обчислення інтеграла за формулою Сімпсона
 методом подвійного перерахунку}
procedure Simpson(a, b, eps: real);
var
  x, h, s, s1: real;
  n, k: integer;
begin
  k: = 0; // за першому обчисленні інтеграла k=0
  n: = 4; // Початкове число точок розбиття
  інтервалу (a, b) h: = (b - a) / n; // крок
  обчислення підінтегральної функції while true do
  begin
    x:= a;
```

```
    x: = x + h;
    s: = 0;
    while x <(bh) do
    begin
      s: = s + Fx (x) + 2 * Fx (x +
      h); x: = x + 2 * h;
    end;
    s: = 2 * s;
    s:= (h/3) * (Fx(a) + 2 * Fx(b) + s);
    if k = 0
    then
    begin
      k:= 1;
      s1: = s;
      h:= h/2;
      continue;
    end
    else
    if abs(s1 - s) > eps then
    begin
      s1: = s;
      h:= h/2;
      continue;
    end
    else
      break;
  end;
  writeln(UTF8ToConsole('Значення інтеграла s='),
s:0:4); end; // кінець процедури
```

```
begin // початок основної програми
 {задаємо інтервал на якому обчислюється інтеграл}
  a: = 0;
  b: = pi/4;
  eps:= 1e-5; // задана точність обчислення інтеграла
  Simpson (a, b, eps); // виклик процедури обчислення
  інтеграла writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

3.1.1.4 Методи передачі параметрів

Передача параметрів за значенням

При такому способі передачі параметрів у функцію або процедуру передається копія змінної. Всередині функції або процедури можна змінювати значення переданих параметрів, однак у програмі, що викликає, значення параметрів залишаються незмінними.

Розглянемо приклад:

```
program parameters;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: real;
  n: integer;
{Оголошення процедури з параметром}
procedure example(x, y: real; n: integer);
begin
```

```
x: = 1.5;
```

```
  y:= 2.8;
  n: = 10;
end;
begin
  x:= 1;
  y:= 1;
  n: = 1;
  writeln('x=', x:0:2, 'y=', y:0:2, 'n=', n); example(x,
  y, n);
  writeln('x=', x:0:2, 'y=', y:0:2, 'n=', n);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Як бачите, після виклику процедури змінних x, y, n не змінилися.

У попередніх прикладах ми використовували передачу параметрів за значенням.

Передача параметрів за посиланням

Інша ситуація під час передачі параметрів за посиланням. У цьому випадку зміна параметра всередині функції (процедури) спричиняє зміну значення змінної в програмі, що викликає. Для передачі параметра посилання потрібно перед ім'ям параметра в заголовку вказати ключове слово var. Розглянемо попередній приклад, але передамо параметри x та n за посиланням.

```
program parameters;
{$mode objfpc}{$H+}
uses
```

```
  CRT, FileUtil;
var
  x, y: real;
  n: integer;
{Оголошення процедури з параметром}
procedure example(var x: real; y: real; var n: integer);
begin
  x: = 1.5;
  y:= 2.8;
  n: = 10;
end;
begin
  x:= 1;
  y:= 1;
  n: = 1;
  writeln('x=', x:0:2, 'y=', y:0:2, 'n=', n); example(x,
  y, n);
  writeln('x=', x:0:2, 'y=', y:0:2, 'n=', n);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Тут значення змінних x і n змінилися!

Передача параметрів-констант

Якщо при передачі параметрів за значенням всередині функції (процедури) їх значення можна змінювати, то при передачі параметрів як констант всередині функції або процедури взагалі неможливо змінити. При спробі їх з-

зміни компілятор видасть помилку. Для передачі параметра як константи потрібно перед ім'ям параметра встановити ключове слово const.

```
program parameters;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: real;
  n: integer;
{Оголошення процедури з параметром}
procedure example(var x: real; y: real;
                  const n: integer);
begin
  x: = 1.5;
  y:= 2.8;
  {Спроба змінити параметр-константу}
  //n: = 10; //тут, якщо забрати коментар, компілятор вкаже на помилку
end;
begin
  x:= 1;
  y:= 1;
  n: = 1;
  writeln('x=', x:0:2, 'y=', y:0:2, 'n=', n); example(x,
  y, n);
  writeln('x=', x:0:2, 'y=', y:0:2, 'n=', n);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

3.1.1.5 Процедури завершення

Для дострокового завершення функції чи процедури, і навіть основний програми застосовуються спеціальні процедури exit і halt. Якщо exit виконується у функції (процедурі), то її виконання негайно припиняється, навіть якщо не всі оператори були виконані і керування передається програмі, що викликає (функції, процедурі). Далі виконуватиметься наступний за викликом цієї функції (процедури) оператор. Якщо exit виконується в основній програмі, то робота програми завершується і керування передається операційній системі.

Процедура halt одразу завершує роботу програми та передає керування операційною системою. Найчастіше halt використовують для аварійного завершення програми.

3.2. Ще раз про типи даних

Досі при складанні програм ми використовували всього три типи даних – цілий, речовий та логічний, до того ж використовували не всі їхні можливості. Однак у Паскалі є дуже широкий набір типів даних, причому мова передбачає створення типів даних користувача. Тут ми дамо класифікацію типів даних у Паскалі та розглянемо деякі з них.

3.2.1 Класифікація типів даних

Всі типи в Паскалі поділяються на стандартні та користувальницькі типи. Типи користувача створюються на основі стандартних. При описі типів користувача використовується ключове слово type.

До стандартних типів відносяться:
- цілий;
- речовий;
- символьний;
- логічний.

Крім того, всі типи можна розділити на категорії:
- прості;
- строкові;
- дати та часу;
- структуровані;
- покажчики;
- об'єкти;
- класи;
- варіанти.

Прості типи, своєю чергою, поділяються на порядкові і речові. Порядковий тип характеризується тим, що кожному його значенню можна поставити у відповідність ціле число – його порядковий номер у сукупності значень. Наприклад, для цілого типу саме значення числа є його порядковим номером. Для логічного типу значенням true відповідає 1, а значенню false відповідає 0.

3.2.1.1 Цілий тип

Окрім вже знайомого нам типу integer, у Паскалі є й інші цілочисленні типи. Вони різняться діапазоном представлення цілого числа в пам'яті комп'ютера (розміром пам'яті, що виділяється) і способом представлення числа (зі знаком або без знака). У таблиці 3.1 наведено найважливіші характеристики цілочисленних типів даних (нагадую, що ми розглядаємо мову стосовно компілятора Free Pascal).

Таблиця 3.1

Тип	Діапазон значень	Розмір пам'яті (байти)	Формат
Byte	0..255	1	без знаку
ShortInt	-128..+127	1	зі знаком
Word	0..65535	2	без знаку
SmallInt	-32768..+32767	2	зі знаком
Integer	-2147483648..+2147483647	4	зі знаком
LongInt	-2147483648..+2147483647	4	зі знаком
LongWord	0..4294967295	4	без знаку
Cardinal	0..4294967295	4	без знаку
Int64	-263..+263-1	8	зі знаком

Як бачите, кількість цілих типів досить велика, проте найчастіше використовуються типи integer і cardinal. Ці два типи забезпечують максимальну продуктивність на 32-бітових платформах. Зауважимо, що вказаний діапазон, відповідний LongInt, вірний лише режимів компіляції OBJFPC і DELPHI. В інших випадках (в т.ч. і за умовчанням) Integer відповідає SmallInt (2 байти).

3.2.1.2. Інтервальний тип

Інтервальний тип визначається на основі порядкового типу та дозволяє обмежити діапазон допустимих значень у вигляді деякого інтервалу виду:

```
Мінімальне значення. Максимальне значення
```

У цьому символи ".." між мінімальним і максимальним значеннями вважаються одним символом, тобто. прогалини між цими точками неприпустимі. Природно, максимальне значення має бути більшим за мінімальне. Опис змінної інтервального типу має вигляд:

```
var <Змінна>:Мінімальне значення. Максимальне значення;
```

Наприклад:

```
var day: 1..31;
    month: 1..12;
    year: 2000..2008;
```

3.2.1.3. Перелічуваний тип

У цьому типі, як випливає з назви, значення задаються простим перерахуванням через кому, причому весь список полягає в дужках, наприклад:

```
var education: (student, bachelor, graduate, doctor);
    course: (first, second, third, fourth, fifth);
```

Порядковий номер списку починається з 0, таким чином сту- дент має порядковий номер 0, bachelor порядковий номер 1, graduate номер 2, doctor номер 3.

3.2.1.4. Безліч

Безліч Object Pascal є групою елементів, з якими можна порівнювати інші елементи з метою визначення входять ці елементи до складу множини чи ні. Багато надають програмістові можливість досить просто представити колекцію символів, чисел або інших типів даних, що перераховуються. Оголошення безлічі має вигляд:

```
Set of <Базовий
тип> Наприклад:
var
  num: set of 1..10;
  num1: set of 20..100;
```

Тут num - безліч, що складається з десяти цілих чисел.

num1 – множина, що складається з цілих чисел від 20 до 100. Значенням перемінної типу множина є набір значень або інтервалів порядкового типу, укладених у квадратні дужки. Така форма визначення множини називається конструктором множини.

Присвоєння значення змінним:

```
num:=[1, 3, 5, 7, 9];
num1:=[21..75, 81, 82..95];
```

Щоб визначити, чи належить змінна безлічі, використовується оператор in:

```
if 81 in num1 then
 <Якісь дії>
```

Можна безліч заздалегідь не визначати, а відразу використовувати конструктор множини в операторі in:

```
if range in [1..50, 75..100] then
 <Якісь дії>
```

3.2.1.5. Логічний тип

Логічний тип має лише два значення true (істина, так) і false (брехня, ні). Причому логічному true відповідає порядкове число 1, а false 0. Таким чином true "більше" false!

Можливі логічні типи представлені у таблиці 3.2.

Таблиця 3.2

Тип	Розмір пам'яті (байти)
Boolean	1
ByteBool	1
WordBool	2
LongBool	4

Рекомендується використовувати тип boolean, інші типи введені для сумісності з іншими мовами.

3.2.1.6. Речовий тип

Речові числа представлені в пам'яті комп'ютера у формі з плаваючою точкою і дозволяють проводити обчислення з великою точністю та значно більшим діапазоном значень чисел, у тому числі й дробових.

Основні речові типи представлені у таблиці 3.3.

Таблиця 3.3

Тип	Діапазон значень	Число значущих розрядів	Розмір пам'яті (Байти)
Real48	$\pm 2.9 \cdot 10^{39}$.. $\pm 1.7 \cdot 10^{38}$	11 – 12	6
Real	$\pm 5.0 \cdot 10^{324}$.. $\pm 1.7 \cdot 10^{308}$	15 – 16	8
Single	$\pm 1.5 \cdot 10^{45}$.. $\pm 3.4 \cdot 10^{38}$	7 – 8	4
Double	$\pm 5.0 \cdot 10^{324}$.. $\pm 1.7 \cdot 10^{308}$	15 – 16	8
Extended	$\pm 3.6 \cdot 10^{4932}$.. $\pm 1.1 \cdot 10^{4932}$	19 - 20	10
Comp	-263 .. 263	19 - 20	8
Currency	-922337203685477.5808 +922337203685477.5807	19 - 20	8

Максимальну продуктивність та точність забезпечує тип Extended. Тип Currency мінімізує помилки округлення та його доцільно застосовувати для грошових розрахунків. Тип Comp насправді ціле 64-х розрядне число, але воно обробляється як і речові типи, тобто. у виразах повністю сумісний із речовими типами.

3.2.1.7. Вказівники

Покажчик це змінна особливого типу, де містяться не самі дані, а адресу пам'яті, де ці дані зберігаються. Точніше адреса першого байта цих даних. Таким чином, покажчик хіба що посилається на дані за допомогою свого значення (адреси). Прикладом вказівника у звичайному житті може бути номер телефону. Саме собою це число нічого не означає, але якщо ви наберете цей номер у своєму мобільному телефоні, ви "потрапите" до потрібного абонента.

Покажчики бувають типізовані та нетипізовані. При оголошенні типізованого покажчика завжди вказується тип даних, куди посилається покажчик. Опис вказівника має вигляд:

```
var ім'я змінної: ^тип;
```

Наприклад:

```
var px: ^integer; // покажчик на дані цілого
    типу py: ^ real; // покажчик дані
    речовинного типу pname: ^string; //
    покажчик даних типу рядок pphone:
    ^string[7];
```

Компілятор суворо стежить за правильністю використання покажчика та типом даних, куди посилається цей покажчик.

Для нетипізованих покажчиків тип даних не вказується. У цьому випадку програміст сам повинен дбати про правильне використання покажчика та типів даних. У його описі використовується ключове слово pointer. Приклад опису нетипізованого покажчика:

```
var p: pointer;
```

Докладніше ми вивчатимемо покажчики у розділі 4.

Надалі ми розглянемо ще типи даних, зокрема у наступному розділі познайомимося з типами даних, що дозволяють обробляти символьну інформацію.

3.3. Обробка символьної інформації у Паскалі

3.3.1 Символьні та рядкові типи даних.

Перші застосування ЕОМ були переважно на вирішення про обчислювальних завдань, тобто. задач, що виникають у математиці, у науково – технічних задачах, де потрібне розв'язання різних рівнянь, обчислення значень функцій тощо. Але застосування комп'ютерів для обчислювальних завдань складає всього 20-25% порівняно з їх застосуванням в інших галузях. До таких завдань належать завдання, що виникають у лінгвістиці, логіці,

психології, теорії ігор тощо.

Комп'ютери широко застосовуються для набору різних текстів та документів, перекладу текстів. Комп'ютери вигадують музику, пишуть вірші, грають у шахи. Звичайно, при вирішенні таких завдань обчислення виробляються, але не в такому обсязі, як при вирішенні обчислювальних завдань. Здебільшого у небагатьох завданнях комп'ютер оперує із символьною інформацією.

Необхідно розуміти, що вся інформація, що зберігається у пам'яті комп'ютера, представлена у вигляді двійкових чисел. Вся річ у тому, що під цими двійковими числами розуміється, чи це справді якісь числа чи щось інше. Так ось під "інше" мають на увазі символи або сукупність символів. Тобто. кожен символ кодується як двійкових чисел. Тільки обробляються ці цифри зовсім інакше. Програміст знає, коли він працює з числами, коли із символами і тому передбачає для кожного випадку відповідні способи роботи з цими числами. А в пам'яті комп'ютера двійкове уявлення символу або групи символів може збігатися з якимось "справжнім" числом.

Примітно, що можливість застосування комп'ютерів для вирішення нечислових завдань розуміли ще тоді, коли й комп'ютерів взагалі не було! Ось що писала знаменита Ада Лавлейс ще в 1843 році: "Багато хто не знає в математиці люди думають, що оскільки призначення аналітичної машини Беббіджа - видавати результати в чисельному вигляді, то природа процесів, що відбуваються в ній, повинна бути арифметичною і чисельною, а не алге. впорядковувати і комбінувати числові значення так само, як і літери або будь-які інші символи загального характеру.

У 1963 р. американська організація зі стандартизації American Standards Association (ASA) запропонувала представлення символів, тобто. цифр, букв та інших знаків спеціальний семибітний код, який став називатися кодовою

таблицею ASCII (American Standard Code for Information Interchange). Номер, який має символ у таблиці ASCII, називається кодом цього символу. Символ можна уявити, вказавши його в лапках, а можна використовувати значок #, за яким слідує код символу. Наприклад, літера 'A', в таблиці ASCII має номер 65, тобто його код дорівнює 65, тоді можна вказати # 65 і це означатиме літеру 'A'.

Однак ця кодова таблиця містила окрім цифр та знаків лише літери англійського алфавіту. Тому був прийнятий стандарт на 8-бітну таблицю ASCII, в якій перші 128 символів залишалися ті ж, що й у 7-бітовій таблиці, а символи з 128 по 255 відводилися для не англійських символів. Пізніше Microsoft розширила таблицю ASCII і була перейменована і стала називатися ANSI (American National Standards Institute). У таблиці 3.4. наведено першу половину (з кодами 0…127) цього стандарту.

Як бачимо, перша половина таблиці містить усі літери латинського алфавіту, цифри від 0 до 9, і навіть всі найуживаніші знаки, такі як знак +, -, /, *, дужки тощо.

Друга половина символів із кодами 128…255 змінюється для різних національних алфавітів. З появою національних локалізацій для другої половини таблиці ASCII було запроваджено поняття «кодова сторінка» (code page, CP). Для кодування російських літер у MS DOS стали застосовувати кодування CP866, раніше відоме як альтернативне кодування ВЦ Академії Наук СРСР.

У Windows для представлення кирилиці використовується кодова сторінка CP-1251. Стандартні Windows-шрифти Arial Cyr, Courier New Cyr та Times New Roman Cyr для представлення символів кирилиці (без літер 'ѐ' і 'Ё') використовують останні 64 коди (від 192 до 256): 'А'...'Я' кодуються значеннями 192...22, 22.

На консолі Windows використовується кодування CP866. Цим і пояснюються проблеми під час виведення російських букв на екран у консольних додатках.

Кодування символів відповідно до стандарту ANSI

Таблиця 3.4

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
25	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
28	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	⬬

Існують інші стандарти кодування символів. Зокрема, для представлення букв деяких мов, таких як китайська, японська, корейська та ін. 8 розрядів не вистачає. Тому розроблено спеціальний стандарт Unicode.

Юнікод, або Унікод (Unicode) — стандарт кодування символів, що дозволяє подати знаки практично всіх письмових мов.

Стандарт запропонований у 1991 році некомерційною організацією «Консорціум Юнікоду» (Unicode Consortium, Unicode Inc.). Застосування цього стандарту дозволяє закодувати дуже велику кількість символів із різних писемностей: у документах Unicode можуть сусідити китайські ієрогліфи, математичні символи, літери грецького алфавіту, латиниці та кирилиці, при цьому стають непотрібними кодові сторінки.

Стандарт складається з двох основних розділів: універсальний набір символів (UCS, Universal Character Set) та сімейство кодувань (UTF, Unicode Transformation Format).

Універсальний набір символів задає однозначну відповідність символів кодам — елементам кодового простору, які мають негативні цілі числа.

Сімейство кодувань визначає машинне представлення послідовності кодів UCS.

Коди у стандарті Юнікод поділені на декілька областей. Область з кодами від U+0000 до U+007F містить символи набору ASCII з відповідними кодами. Далі розташовані області знаків різних писемностей, знаки пунктуації та технічні символи. Частина кодів зарезервована для використання у майбутньому.

У Lazarus за замовчуванням використовується кодування UTF-8. У UTF-8 всі символи поділені на кілька груп. Символи з кодами менше 128 кодуються одним байтом, перший біт якого дорівнює нулю, а наступні 7 біт точно відповідають першим 128 символам 7-бітної таблиці ASCII, наступні 1920 символів – кодуються двома байтами. Наступні символи кодуються трьома та чотирма байтами.

Для нас важливим є той факт, що символи кирилиці кодуються в UTF-8 точно двома байтами.

Як зазначалося в 2.1.8. Lazarus є середовищем із графічним інтерфейсом для швидкої розробки програм і базується на оригінальній кросплатформовій бібліотеці візуальних компонентів LCL (Lazarus Component Library). Розробниками Lazarus запропоновано використовувати UTF-8 в LCL як універсальне кодування на всіх платформах. Тому LCL містить, крім візуальних компонентів, функції перетворення UTF-8 на кодування, з яким працює консоль на кожній з платформ. Функція UTF8ToConsole() оголошена у модулі FileUtil, що є частиною LCL. Ось чому ми повинні були додавати до наших консольних проектів бібліотеку LCL.

3.3.1.1. Тип Char

Для позначення типу "символ" у Паскалі використається зарезервоване слово char. Для зберігання змінної типу " символ " потрібен один байт пам'яті, тобто. значенням змінної типу char є символ.

Нагадую, хоча комп'ютер обробляє символи, проте, насправді він оперує з числами, а точніше з кодами цих символів з кодової таблиці. Таким чином, один символ може бути "більшим", ніж інший або "меншим". Це залежить від розташування символів у таблиці. Наприклад, символ (літера) 'B' "більше" літери 'A', оскільки код 'B' (номер у кодовій таблиці) дорівнює 66, а код літери 'A' дорівнює 65 (літери англійського алфавіту, див. табл. 3.4.). З огляду на це, над символьними змінними визначено операції відносини: =, <, >, <>, <=, >=.

3.3.1.2. Функції для роботи із символами

Для символьних змінних існують такі функції:

```
chr(x) – повертає значення символу за його кодом;
ord(ch) – повертає код символу ch;
```

pred(ch) – повертає попередній символ із кодової таблиці; succ(ch) – повертає наступний символ кодової таблиці; upcase(ch) – перетворює малу літеру на заголовну. Працює тільки

для літер англійського алфавіту.

3.3.1.3. Тип String

Рядком називається деяка послідовність символів: 'AABBCC', 'Вася-це кіт'.

Рядок може складатися з кількох символів, з одного символу, а може бути зовсім порожнім. Максимальна довжина рядкової змінної залежить від директиви компілятора $H. Якщо включена директива {$H-}, то максимальна довжина рядка 255 байтів, якщо включена директива {$H+}, то довжина рядка практично необмежена і може досягати до ~2 Гб.

Тип рядкової змінної зазначається зарезервованим словом string. Якщо заздалегідь відома довжина рядка, наприклад 30 символів, можна вказати string[30].

Рядкову змінну можна розглядати як одне ціле, а можна як масив символів, причому нумерація символів починається з 0.

Приклад опису рядків та символів.

```
var Symbol: char;
    Message: string;
    Name: string[30];
```

Які операції допустимі для рядків та символів? Тільки операція складення. При цьому в результаті виходить рядок.

Приклад:

```
program ch_str;
```

```
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  Symbol: Char;
  Mum, Str: string;
begin
  Symbol:='A'; // це буква А латинського алфавіту
  Str: = 'Ш';
  Str: = Str + Symbol;
  Mum: = 'МАМА';
  Mum: = Mum + Str;
  {тепер Mum = 'МАМАША'}
  writeln(UTF8ToConsole(Mum));
  Str:= 'ТАТКА = ПЕРЕДКИ';
  Mum: = Mum + '+' + Str;
  {тепер Mum = 'МАМАША + БАТОЧКА = ПЕРЕДКИ'}
  writeln(UTF8ToConsole(Mum));writeln(UTF8ToConso
  le('Натисніть будь-яку клавішу')); readkey;
end.
```

Зверніть увагу, якщо ви в операторі

```
Symbol:='A';
```

вкажете російську букву 'А', то компілятор випустить помилку!

Як уже зазначалося, у Lazarus та FPC використовується кодування UTF8. У цьому кодуванні для представлення кириличних символів використовуються два байти. Таким чином, змінної типу char не можна присвоїти значення кирил-

особи. Якщо ви все ж таки хочете використовувати символи кирилиці в символьних змінних, то використовуйте тип TUTF8Char, як у наступному прикладі.

Приклад:

```
program ch_str;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, LCLType;
var
  Symbol: TUTF8Char; // новий тип, введений у FPC
  Mum, Str: string;
begin
  Symbol:='А'; // це російська літера А, займає у пам'яті
  два байти
  Str: = 'Ш';
  Str: = Str + Symbol;
  Mum: = 'МАМА';
  Mum: = Mum + Str;
  {тепер Mum = 'МАМАША'}
  writeln(UTF8ToConsole(Mum));
  Str:= 'ТАТКА = ПЕРЕДКИ';
  Mum: = Mum + '+' + Str;
  {тепер Mum = 'МАМАША + БАТОЧКА = ПЕРЕДКИ'}
  writeln(UTF8ToConsole(Mum));writeln(UTF8ToConso
  le('Натисніть будь-яку клавішу')); readkey;
end.
```

Зауважте, що у рядкових змінних (типу string) під символи латиниці (коди яких менше 128 у таблиці ASCII) відводиться один байт, а під символи кирилиці відводиться два байти!

Наступна програма імітує процес авторизації користувача для

входу до системи. Для цього користувач має ввести правильний пароль. Якщо після трьох спроб правильний пароль не буде введено, користувачеві буде заборонено доступ до системи.

```
program password;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, LConvEncoding;
var
  answ, passw: string;
  n: integer;
begin
 passw:= 'абвг';
 n: = 1;
 repeat
   writeln(UTF8ToConsole('Введіть пароль'));
   readln(answ);
   {$IFDEF WINDOWS}
     answ:= CP866ToUTF8(answ); // Перетворення введеної
                               // рядки до UTF8
   {$ENDIF}
   if answ <> passw then
   begin
     if n = 1 then
     begin
       writeln(UTF8ToConsole('Ви не
       користувач')); inc(n);
     end
     else
```

```
    begin
      if n = 3 then
      begin
        writeln(UTF8ToConsole('Вам відмовлено у
        доступі')); break;
      end
      else
      begin
        writeln(UTF8ToConsole('Ви не користувач'));
        writeln(UTF8ToConsole('Ви'), n,
        UTF8ToConsole('-раз ввели неправильний
        пароль')); writeln(UTF8ToConsole('Після 3-ї
        спроби вам'));
        writeln(UTF8ToConsole('відмовлено у
        доступі')); inc(n);
      end
    end
  end
  else
  writeln(UTF8ToConsole('Здрастуйте, ви увійшли в
 систему!')); until answ = passw;
 writeln(UTF8ToConsole('Натисніть будь-яку
 клавішу')); readkey;
end.
```

У цій програмі ми використовували функцію CP866ToUTF8() для перетворення введеного з клавіатури рядка до UTF-8. Вочевидь, це стосується лише користувачів Windows, оскільки ми вже неодноразово зазначали, що у консолі Windows використовується кодування CP866. Функція CP866ToUTF8() описана в модулі LConvEncoding.

Зверніть увагу на директиви компілятора для умовної компіляції

```
{$IFDEF WINDOWS}
   answ:= CP866ToUTF8(answ);
{$ENDIF}
```

Директива {$IFDEF <умова>} наказує компілятору компілювати оператори, що знаходяться після неї і до {$ENDIF} якщо <умова> виконується. В іншому випадку ці оператори пропускаються. У даному випадку оператор

```
answ:= CP866ToUTF8(answ);
```

буде включено до компіляції в операційній системі Windows і буде пропущено в Linux. Повна форма цієї директиви:

```
{$IFDEF <умова>}
   <оператори> // компілюються, якщо <умова> виконано
{$ELSE}
   <оператори> // компілюються, якщо <умова> не виконано
{$ENDIF}
```

3.3.1.4. Строкові процедури та функції

Функція Concat- виконує конкатенацію послідовності рядків, тобто об'єднання кількох рядків.

Опис: `Concat (S1 [, S2, ..., Sn]: string): string;`

`S1, S2, ..., Sn - формальні параметри, мають тип string. Квадратні дужки вказують, що параметри S2, Sn можуть бути відсутніми. Результат функції - рядок символів, тобто. має також тип string.`

Приклад:

```
program function_concat;
{$mode objfpc}{$H+}
uses
  Crt, FileUtil;
var
  a, c: string;
begin
  a:= 'Коля + Оля';
  writeln(UTF8ToConsole('Перший рядок: '),
          UTF8ToConsole(a));
  c:= '=кохання';
  writeln(UTF8ToConsole('Другий рядок:'),
          UTF8ToConsole(c));
  c: = Concat (a, c);
  writeln(UTF8ToConsole('Результуючий рядок: '),
          UTF8ToConsole(c));writeln(UTF8ToConsole
  ('Натисніть будь-яку клавішу')); readkey;
end.
```

Функція Length – повертає динамічну довжину рядка. Опис:

Length(S: string): integer;

Приклад:

```
program function_length;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  S: string;
```

```
  i: integer;
begin
  S:= 'Іванов';
  i:= Length(S);
  writeln(UTF8ToConsole('Рядок S:'), UTF8ToConsole(S));
  writeln(UTF8ToConsole('Довжина цього рядка = '), i);
  S:= '';
  i:= Length(S);
  writeln(UTF8ToConsole('Рядок S:'),
  UTF8ToConsole(S)); writeln(UTF8ToConsole('Тепер
  довжина рядка стала = '), i);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Слід пам'ятати, що функція Length() повертає кількість байтів, що займає рядок! У цьому легко переконатись, трохи видозмінивши попередню програму.

```
program function_length;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  S, S1: string;
  i: integer;
begin
  S:= 'Іванов';
  i:= Length(S);
  writeln(UTF8ToConsole('Рядок S:'), UTF8ToConsole(S));
  writeln(UTF8ToConsole('Довжина цього рядка = '), i);
```

```
  S1: = 'Іванів';
  i:= Length(S1);
  writeln(UTF8ToConsole('Рядок S1: '),
  UTF8ToConsole(S1)); writeln(UTF8ToConsole('Довжина
  цього рядка = '), i); writeln(UTF8ToConsole('Натисніть
  будь-яку клавішу')); readkey;
end.
```

Довжина рядка S дорівнює 6, довжина рядка S1 дорівнює 12! Це не помилка, просто функція Length()повертає не кількість символів у рядку, а кількість байтів, яку займає рядок у пам'яті. Для латиниці це виходить те саме, що у UTF-8 символи латиниці кодуються одним байтом, а кирилиці вдвічі більше, оскільки російські літери кодуються двома байтами. Якщо ви хочете отримати кількість символів у рядку з кирилицею, можна скористатися функцією UTF8Length(). Ця функція оголошена в модулі LCLProc.

Замініть у попередньому прикладі оператор

```
i:= Length(S1);
```

на оператор

```
i:= UTF8Length(S1);
```

і додайте модуль LCLProc в оголошення uses і запустіть програму. Тепер довжина рядка S1 також дорівнювала 6.

Спробуйте замінити оператор

```
i:= Length(S);
```

на оператор

```
i:= UTF8Length(S);
```

Ви побачите, що довжина рядка S як і дорівнює 6. Тобто. функція UTF8Length() для рядків з латиницею працює ідентично функції Length(). Звідси можна дійти невтішного висновку, що зручніше застосовувати функцію

UTF8Length(), особливо якщо вам заздалегідь невідомий вміст рядка.

Процедура SetLength – встановлює довжину рядка. Опис:

SetLength(S: string, n: integer);

S – рядок, n – нова довжина рядка. Якщо рядок складається з кирилиці, то необхідно n множити на 2. А якщо рядок містить символи та кирилиці та латиниці, то доведеться вираховувати, скільки символів кирилиці міститься в рядку. Так що без особливої необхідності застосовувати цю функцію не варто.

Функція Copy – повертає підрядок рядка.
Опис:

```
Copy(S: string, index, count: integer): string;
```

S- вихідний рядок,
index - номер символу, починаючи з якого виділяється підрядок з S, count - кількість символів у підрядку.
У модулі LCLProc є аналог цієї функції UTF8Copy().

Приклад:

```
program function_copy;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, LCLProc;
var
  a, c: string;
begin
```

```
  a:= 'Head&Shoulders';
  writeln(UTF8ToConsole('Початковий рядок:'),
          UTF8ToConsole(a));
  c:= Copy(a, 6, 9);
  c:= UTF8Copy(a, 6, 9); // можна і так
  writeln(UTF8ToConsole('Результуючий рядок: '),
          UTF8ToConsole(c));
  a:= 'Меню:Оточення';
  writeln(UTF8ToConsole('Початковий рядок:'),
          UTF8ToConsole(a));
  //c:= Copy(a, 6, 9); // Це неправильно!
  //c:= Copy(a, 10, 18); // але можна так
  c:= UTF8Copy(a, 6, 9); // найкраще так
  writeln(UTF8ToConsole('Результуючий рядок: '),
          UTF8ToConsole(c));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Процедура Delete- видаляє з рядка підрядок Опис:

```
Delete (var S: string, index: integer, count: integer);
  S – вихідний рядок, index – номер символу, з якого
  видаляє-
```

ся підрядок з S, count-кількість символів у підрядку. є аналогічна процедура UTF8Delete().

Приклад:

```
program procedure_delete;
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil, LCLProc;
var
  a: string;
begin
  a:= '123XYZ';
  writeln(UTF8ToConsole('Початковий рядок: '),
          UTF8ToConsole(a));
  UTF8Delete(a, 4, 3);
  writeln(UTF8ToConsole('Результуючий рядок: '),
          UTF8ToConsole(a));
  a:= '123АБВ';
   writeln(UTF8ToConsole('Початковий рядок: '),
          UTF8ToConsole(a));
  UTF8Delete(a, 4, 3);
  writeln(UTF8ToConsole('Результуючий рядок: '),
          UTF8ToConsole(a));writeln(UTF8ToConsole
  ('Натисніть будь-яку клавішу')); readkey;
end.
```

Процедура Insert – додає підрядок у рядок. Опис:

```
Insert(Source: string, S: string, index: integer);
```

Source - вихідний рядок

S- рядок, що додається

index - номер символу рядка Source, починаючи з якого додається підрядок S.

Приклад:

```
program procedure_insert;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, LCLProc;
var
  S: string;
begin
  S:= '1236789';
  writeln(UTF8ToConsole('Початковий рядок: '),
          UTF8ToConsole(S));
  Insert('45', S, 4);
  writeln(UTF8ToConsole('Результуючий рядок: '),
          UTF8ToConsole(S));
  S:= 'абвеежзиклмн';
  writeln(UTF8ToConsole('Початковий
  рядок:'),
          UTF8ToConsole(S));
  UTF8Insert('гд', S, 4);
  writeln(UTF8ToConsole('Результуючий рядок: '),
          UTF8ToConsole(S));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Функція Pos- здійснює пошук підрядки у рядку. Опис:

```
Pos(Substr, S: string): Byte;
```

Substr - підрядок, пошук якого проводиться в рядку S. Функція Pos повертає номер символу, починаючи з якого підрядок Substr знайде-

нав S. Якщо такого підрядка немає, то повертається 0. У модулі LCLProc є аналогічна функція UTF8Pos().

Приклад:

```
program function_pos_1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  S: string;
  i: integer;
begin
  S:= 'МАМАША'; i:=
  Pos('ША', S);
  writeln(UTF8ToConsole('Рядок: '), UTF8ToConsole(S));
  writeln(UTF8ToConsole('Номер символу, з якого'));
  writeln(UTF8ToConsole('починається підрядок "ША"='),
  i); writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Приклад: Дано рядок символів. Визначити, чи входить до цього рядка підрядок 'ША'. Рядок ввести з клавіатури.

```
program function_pos_2;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, LConvEncoding, LCLProc;
var
```

```
  S: string;
  Substr: string;
  i: integer;
begin
  writeln(UTF8ToConsole('Введіть рядок
  символів')); readln(S);
  {$IFDEF WINDOWS}
    S:= CP866ToUTF8(S);
  {$ENDIF}
  Substr:= 'ША';
  i:= UTF8Pos(Substr, S);
  if i = 0 then
    writeln(UTF8ToConsole('У даному рядку') +
           UTF8ToConsole(S) + UTF8ToConsole('підрядок'),
           UTF8ToConsole(Substr) +
           UTF8ToConsole('відсутня'))
  else
    writeln(UTF8ToConsole('підрядок') +
            UTF8ToConsole(Substr) +
    UTF8ToConsole(' входить до цього рядка з номера символу
  '), i); writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу'));
  readkey;
end.
```

Процедура Str – перетворює чисельне значення його рядкове представлення. Опис:

```
  Str(X[: width[: decimals]], S: string);
  X- числова змінна (типу integer або
  real)
  width - Довжина рядка S; decimals - кількість знаків
```

```
   після за- п'ятої.
```

Приклад:

```
program function_str;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  S: string;
  X: real;
begin
  X: = 123.565;
  Str(X:6:2, S);
{тепер S='123.56'}
  writeln(S);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Процедура Val перетворює рядкове значення на його чисельне подання.

Опис:

```
Val(S: string, v, code: integer);
```

S- вихідний рядок символів

v-числова змінна – результат перетворення

S

code- ціла змінна

Якщо неможливо перетворити S на число, то після виконання процедури Val code 0, якщо перетворення відбулося вдало, то code=0.

Приклад:

```
program function_val;
uses
  CRT, FileUtil;
```

```
var
  S: string;
  c: integer;
  X: real;
begin
  S:='234.12';
  Val(S, X, c);
  writeln(X);
  writeln(X:0:2);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Процедуру зручно використовувати для організації контролю при введенні чисел даних. Для цього спочатку необхідно вводити числа в символьному форматі, потім перетворювати число процедурою Val і перевіряти значення code.

Функція LowerCase перетворює великі літери на малі (не працює з кирилицею). Опис:

```
LowerCase(const S: String): String;
```

Функція Uppercase перетворює малі літери в великі (не працює з кирилицею). Опис:

```
Uppercase (const S: String): String;
```

Зручніше для цих цілей використовувати функції UTF8LowerCase() і UTF8UpperCase(), які підтримують кирилицю.

```
program project1;
```

```
{$mode objfpc}{$H+}
uses
  Crt, FileUtil, LCLProc, LConvEncoding;
var
  str: string;
begin
  writeln(UTF8ToConsole('Введіть рядок'));
  readln(str);
  {$IFDEF WINDOWS}
    str:= CP866ToUtf8(str);
  {$ENDIF}
  str:= UTF8UpperCase(str);
  writeln(UTF8ToConsole('Рядок у верхньому
  регістрі: '),
          UTF8ToConsole(str));st
  r:= UTF8LowerCase(str);
  writeln(UTF8ToConsole('Рядок у нижньому
          регістрі: '), UTF8ToConsole(str));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Розглянемо коротко ще кілька функцій роботи з рядками:

- CompareStr(const S1, S2: string): Integer - виконує порівняння двох рядків, роблячи різницю між малими і великими літерами; не враховує місцеву мову. Значення, що повертається менше нуля, якщо S1 < S2, дорівнює нулю, якщо S1 = S2, і більше нуля, якщо S1 > S2.
- CompareText(const S1, S2: string): Integer - виконує порівняння двох рядків, не роблячи відмінностей між

малими і великими бук-

вами; не враховує місцеву мову. Значення, що повертається менше нуля, якщо S1 < S2, дорівнює нулю, якщо S1 = S2, і більше нуля, якщо S1 > S2.

- IntToStr(Value: Integer): string — перетворює ціле число Value у рядок.
- StrToInt(const S: string): Integer — перетворює рядок на ціле число. Якщо рядок не може бути перетворений на ціле число, функція генерує виняткову ситуацію класу EConvertError (обробка виключних ситуацій розглядається в розділі 6).
- DateToStr(const DateTime: TDateTime): string — перетворює числове значення дати в рядок.
- StrToDate(const S: string): TDateTime — перетворює рядок зі значенням дати на числовий формат дати та часу.
- Trim(const S: string): string — повертає частину рядка S без лідируючих і завершальних пробілів та символів, що управляють.
- AnsiToUTF8(const S: string): UTF8String — перетворює рядок на формат UTF-8.
- UTF8Decode(const S: UTF8String): WideString — перетворює рядок формату UTF-8 на рядок формату Unicode.
- UTF8Encode(const WS: WideString): UTF8String — перетворює рядок Unicode на рядок формату UTF-8.
- UTF8ToAnsi(const S: UTF8String): string — перетворює рядок формату UTF-8 на стандартний рядок.

3.4. Масиви

У деяких випадках доводиться стикатися з ситуацією, коли має використовуватися багато змінних одного типу.

Уявімо програму, яка дозволяє обчислити певні метеорологічні дані і в якій завжди буде не менше 365 змінних. Таким чином можна надавати окремим змінним метеорологічні дані щодо окремих днів всього року.

В принципі, зараз немає жодної проблеми, щоб оголосити тип real у 365 змінних, використовуючи запис.

```
var day1, day2, day3 і т.д. day365:real;
```

Однак у Паскалі не можна писати і т.д. Потрібно перерахувати усі змінні. Для цього потрібна, принаймні, сторінка. Якщо тепер потрібно підрахувати середнє арифметичне на один день, потрібно записати:

```
sred:= (day1+day2+i т.д. day365)
```

і знову як бути з "і т.д."?

Для таких випадків Паскаль надає можливість введення великої кількості змінних того самого типу, використовуючи прості вирази.

Ця можливість у Паскалі реалізується за допомогою так званих масивів. Використовується службове слово array, яке вказує на низку змінних одного і того ж типу та мають одне ім'я.

Наприклад:

```
var
  day: array[1..365] of real;
```

За допомогою цього опису визначається масив day що складається з 365 елементів (змінних) речовинного типу. Кожен елемент масиву може використовуватись як окрема змінна. Можна записати:

```
day[1]:= 1.25;
```

```
day[2]:= 0.34;
```

Іншими словами, кожен елемент масиву визначається ім'ям масиву та номером елемента у квадратних дужках. Говорять ще індекс масиву. Причому як індекс можна використовувати арифметичний вираз цілого типу.

В описі масиву після імені у квадратних дужках вказується мінімальне значення індексу, потім дві точки без пробілів та максимальне значення індексу. У нашому випадку масив day складається з 365 елементів, нумерація йде від 1 до 365 тобто. крок нумерації за промовчанням завжди дорівнює 1.

При використанні масивів запис програми стає набагато коротшим і наочнішим.

Нехай, наприклад, на початку всі елементи масиву day повинні бути прирівняні 0.

Замість писати

```
day[1]:= 0;
day[2]:= 0;
```

і т.д.

```
day[365]:= 0;
```

ми можемо вчинити так:

```
var
  day: array[1..365] of real;
  k: integer;
begin
   for k:= 1 to 365 do
     day[k]:= 0;
```

приклад.

Нехай є 2 масиви типу string. В одному масиві містяться прізвища людей, а в іншому номери їхніх домашніх телефонів. Написати програму,

яка у циклі за номером телефону виводить на екран прізвище цього абонента.

```
program phone_1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: array[1..50] of string[30];
  tel: array[1..50] of string[3];
  k: integer;
  phone: string[3];
begin
  for k:=1 to 50 do
  begin
    writeln(UTF8ToConsole('Введіть прізвище'));
    readln(name[k]);
    writeln(UTF8ToConsole('Введіть номер
    телефону')); readln(tel [k]);
  {Видаляємо провідні та ведені
    ("хвостові") прогалини} tel[k]:=
    Trim(tel[k]);
  end;
  writeln(UTF8ToConsole('Введення даних закінчено'));
  writeln(UTF8ToConsole('Для пошуку абонента введіть'));
  writeln(UTF8ToConsole('номер його телефону'));
  writeln(UTF8ToConsole('Для виходу з програми'));
  writeln(UTF8ToConsole('введіть ''***'''));
  phone:= '';
  while phone <> '***' do
```

```
  begin
    writeln(UTF8ToConsole('Введіть номер телефону'));
    readln(phone);
    for k:= 1 to 50 do
      if tel[k] = phone then
        writeln(UTF8ToConsole('Прізвище цього абонента'),
                name [k]);
  end;
 end.
```

Недолік цієї програми в тому, що якщо вказаний номер телефону в масиві не існує, то жодних повідомлень щодо цього на екран не виводиться. Крім того, після того, як знайдено абонента, наприклад для k=2, програма ще 48 разів прокрутить цикл. Як модифікувати програму?

```
program phone_2;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: array[1..50] of string[30];
  tel: array[1..50] of string[7];
  k: integer;
  phone: string[7];
  found: boolean;
begin
  for k:= 1 to 50 do
  begin
    writeln(UTF8ToConsole('Введіть прізвище'));
```

```
    readln (name [k]);
    writeln(UTF8ToConsole('Введіть номер
    телефону')); readln(tel [k]);
    {Видаляємо провідні та ведені ("хвостові")
    прогалини} tel[k]:= Trim(tel[k]);
  end;
  writeln(UTF8ToConsole('Введення даних закінчено'));
  writeln(UTF8ToConsole('Для пошуку абонента введіть'));
  writeln(UTF8ToConsole('номер його телефону'));
  writeln(UTF8ToConsole('Для виходу з програми'));
  writeln(UTF8ToConsole('введіть ''***'''));
  phone:= '';
  while phone <> '***' do
  begin
    found: = false;
    writeln(UTF8ToConsole('Введіть номер телефону'));
    readln(phone);
    for k:= 1 to 50 do
      if tel[k]= phone then
      begin
        writeln(UTF8ToConsole('Прізвище цього абонента'),
                name[k]);
        found: = true;
        break;
      end;
    if not found then
      writeln(UTF8ToConsole('Абонента з таким номером
    немає')); end;
 end.
```

Тут ми ввели булеву змінну found, якій присвоюємо значення true, якщо знайдено абонента із зазначеним номером. Потім ми "достроково" виходимо із циклу for командою break. У зовнішньому циклі, якщо абонента не знайдено, виводимо відповідне повідомлення на екран.

Якщо ви кілька разів запускали програму, то могли бачити, що кожного разу доводиться наново вводити прізвища та телефони. Вводити їх при кожному новому прогоні програми незручно. Єдиний вихід – зберегти введені дані у файлі. Але, оскільки ми ще використання файлів у Паскалі не вивчали, введемо ознаку закінчення введення. Домовимося, якщо замість чергового прізвища ми введемо рядок '***', це означає, що введення даних закінчено. Тепер ми можемо вводити лише кілька прізвищ та телефонів лише для того, щоб переконатися у працездатності програми.

```
program phone_3;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: array[1..50] of string[30];
  tel: array[1..50] of string[7];
  fam: string[30];
  k, n: integer;
  phone: string[7];
  found: boolean;
begin
  n: = 0;
  writeln(UTF8ToConsole('Введіть прізвище'));
  writeln(UTF8ToConsole('Щоб закінчити введення введіть
  "***"')); for k:= 1 to 50 do
```

```
begin
  readln (fam);
  if fam = '***' then break;
  name[k]:= fam;
  n: = n + 1;
  writeln(UTF8ToConsole('Введіть номер телефону'));
  readln(tel [k]);
  {Видаляємо провідні та ведені ("хвостові")
  прогалини} tel[k]:= Trim(tel[k]);
  writeln(UTF8ToConsole('Введіть прізвище'));
end;
if n = 0 then exit;
writeln(UTF8ToConsole('Введення даних
закінчено'));
writeln(UTF8ToConsole('Для пошуку абонента введіть'));
writeln(UTF8ToConsole('номер його телефону'));
writeln(UTF8ToConsole('Для виходу з програми'));
writeln(UTF8ToConsole('введіть ''***'''));
phone:= '';
while phone <> '***' do
begin
  found: = false;
  writeln(UTF8ToConsole('Введіть номер телефону'));
  readln(phone);
  for k:= 1 to n do
    if tel[k] = phone then
    begin
      writeln(UTF8ToConsole('Прізвище цього абонента'),
              name[k]);
      found: = true;
```

```
        break;
      end;
    if not found then
      writeln(UTF8ToConsole('Абонента з таким номером
 немає')); end;
end.
```

І знову не зупиняємось на досягнутому, шукаємо недоліки у програмі, покращуємо та вдосконалюємо її. Власне так і роблять усі розробники програм. Вони розповсюджують свої програмні продукти версіями, покращуючи та вдосконалюючи свою програму від версії до версії.

У нашому випадку ми навчилися вводити менше даних, ніж зарезервували в масивах name і tel, але компілятор розподілить пам'ять для всього об'єму масивів. Значить, частина пам'яті залишиться невикористаною. Наявне нераціональне використання пам'яті. Хоча для сучасних комп'ютерів проблеми з обсягами доступної пам'яті не такі гострі, як у недавньому минулому, все ж таки жоден професійний програміст не піде на резервування пам'яті, так би мовити, "про запас", використовує рівно стільки пам'яті, скільки йому потрібно. Для виправлення цього недоліку нашій програмі скористаємося так званими динамічними масивами.

3.4.1 Динамічні масиви

Динамічний масив – це масив, пам'ять якого виділяється динамічно під час виконання програми. А на момент компіляції його реальний розмір невідомий. Компілятор просто виділяє пам'ять для покажчика на цей масив (4 байти).

При описі динамічного масиву кордону не вказуються, а вказується лише його тип, наприклад:

```
var
  name array of string[30];
  tel array of string[7];
```

Перед використанням такого масиву необхідно встановити розмір масиву за допомогою процедури SetLength (ім'я масиву, розмір), наприклад:

```
SetLength(name, 50);
```

У цьому випадку для масиву name буде виділено (динамічно!) Пам'ять для розміщення 50 елементів.

Можна використовувати багатовимірні динамічні масиви. Ось як, наприклад, описується 3-мірний динамічний масив:

```
a: array of array of array of integer;
```

Виділимо під цей масив пам'ять:

```
SetLength(a, 50, 50, 50);
```

Після використання динамічних масивів пам'ять, розподілена для них, буде автоматично звільнена. Але можна і "вручну" звільнити пам'ять, яку займає динамічний масив. Для цього достатньо надати масиву значення nil.

```
a:= nil;
```

Оскільки для реалізації динамічних масивів використовується механізм покажчиків, застосування оператора присвоювання

```
b: = a;
```

де a і b динамічні масиви не призведе до створення нового масиву b, а будуть створені два покажчики, що посилаються на ті самі ділянки пам'яті. Тобто. зміна елемента масиву b призведе до зміни відповідного елемента масиву a, оскільки фактично це те саме дане, тільки з різними іменами (у нашому випадку a і b).

Для того щоб створити інший масив ідентичний за змістом необ-
ходимо використовувати процедуру Copy(), наприклад:

```
b:= Copy(a);// повна копія
масиву a Можна скопіювати
частину масиву:
b:= Copy(a, 5, 5);
```

У новий масив b буде скопійовано 5 елементів масиву a, починаючи з п'ятого елемента.

Необхідно пам'ятати, що нумерація елементів динамічних масивах починається з нуля.

При передачі динамічних масивів як параметри функції і процедури дізнатися реальний розмір масиву можна за допомогою функції high(ім'я масиву), але можна просто передати реальний розмір масиву через додатковий параметр.

Тепер ми можемо написати нашу багатостраждальну програму. У ній ми використовували динамічні масиви, а також реалізували введення даних у вигляді процедури та функцію пошуку. У функції визначення фактичного розміру масиву використовували функцію high(), а процедуру передали фактичний розмір масиву через параметр.

```
program phone_4;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: array of string[30];
  tel: array of string[7];
  n: integer;
  phone: string[7];
{Ця функція здійснює пошук абонента за номером телефону}
function find_data (var name: array of string[30];
                    var tel: array of string[7];
```

```
                          phone:string[7]): boolean;
var
  k: integer;
begin
  find_data: = false;
  for k:= 0 to high(tel) do
  if (tel[k] = phone) and (phone <> '') then
  begin
    writeln(UTF8ToConsole('Прізвище цього
            абонента'), name[k]);
    find_data: = true;
    break;
  end;
end;
{Процедура введення прізвищ та номерів телефонів}
procedure data_input(var name: array of string[30];
                     var tel: array of string[7];
                     var n: integer);
var
  k: integer;
  fam: string[30];
begin
  writeln(UTF8ToConsole('Введіть прізвище'));
  writeln(UTF8ToConsole('Щоб закінчити введення введіть
  "***"')); for k:= 0 to n - 1 do
  begin
    // запам'ятаємо поточний індекс
    змінної nn:= k; // За його
    значенням ми потім
    // Встановимо фактичний розмір масивів
```

```
    readln (fam);
    if fam = '***' then break;
    name[k]:= fam;
    writeln(UTF8ToConsole('Введіть номер телефону'));
    readln(tel [k]);
    {Видаляємо провідні та ведені ("хвостові")
    прогалини} tel[k]:= Trim(tel[k]);
    writeln(UTF8ToConsole('Введіть прізвище'));
  end;
  writeln(UTF8ToConsole('Введення даних
закінчено')); end;
begin
  n: = 50;
  {спочатку встановлюємо максимальний розмір
  масивів} SetLength(name, 50);
  SetLength(tel, 50);
  data_input(name, tel, n); {введення прізвищ та номерів
  телефонів}
  {тепер встановлюємо фактичний розмір масивів}
  SetLength(name, n);
  SetLength(tel, n);
  if n = 0 then exit;
  writeln(UTF8ToConsole('Для пошуку абонента введіть'));
  writeln(UTF8ToConsole('номер його телефону'));
  writeln(UTF8ToConsole('Для виходу з програми'));
  writeln(UTF8ToConsole('введіть ''***'''));
  phone:= '';
  while phone <> '***' do
  begin
    writeln(UTF8ToConsole('Введіть номер телефону'));
```

```
    readln(phone);
    if (not find_data(name, tel, phone)) and (phone <>
'***') then
      writeln(UTF8ToConsole('Абонента з таким номером
  немає')); end;
end.
```

Як ви вважаєте, чи є в програмі ще недоліки? Взагалі кажучи, можна, звичайно, причепитися і до стовпа, але, заради справедливості, зазначу, що в програмі є ще суттєві недоліки.

По-перше, що буде, якщо під час введення будуть введені однакові прізвища та/або телефони? Буде виведено те прізвище та/або той телефон, які були введені першими. Але в масиві дані, що дублюють, залишаться. Отже, попередньо необхідно переглянути вже сформований масив і лише, якщо прізвище та/або телефон не збігаються, лише тоді додавати чергового абонента до масиву. Другий значний недолік – під час введення телефону не здійснюється жодного контролю. У номері телефону можуть бути лише цифри, ну може бути ще знак тире.

Однак розбір цієї програми займає майже половину книги. Тому спробуйте самі покопатися. Ідеї я вам підкинув. Впевнений, що ви справитеся з цим завданням!

3.4.2 Програма розв'язання системи лінійних рівнянь алгебри методом Гауса

Напишемо програму, що реалізує алгоритм Гауса з вибором головного елемента для вирішення системи лінійних рівнянь алгебри. Сам алгоритм був нами розглянутий у 1.3.4.

Як уже повелося у нас із вами, перш ніж дивитися програму, напишіть його самі по блок-схемі, наведеній у розділі 1.3.4. Використовуючи масиви, реа-

лізувати цей алгоритм для вас не складе труднощів. І лише після цього зіставте свою програму з наведеною у книзі. Думайте, аналізуйте, порівнюйте коди, шукайте найкращі рішення, нещадно критикуйте мене! Цілком можливо, що ви напишете програму краще. Краще не в сенсі результатів (вони повинні збігатися!), а з точки зору реалізації. Можливо, ви напишете ефективнішу програму чи реалізуєте окремі фрагменти алгоритму простіше, ну т.д. І майте на увазі, що й самі алгоритми, які вирішують те саме завдання, можна складати по-різному! Тож спробуйте і алгоритм придумати свій.

Для порівняння я наведу три варіанти програми, написані трьома студентами. Перший, назвемо його "поганим програмістом", реалізував алгоритм, як то кажуть у "лоб". Як записано у блок-схемі, так і реалізовано у програмі. На захисті своєї програми він зізнався мені, що так і не зміг написати цю програму без оператора goto. Крім того, його програма не вміла визначати чи існує рішення чи ні, а також не було організовано введення коефіцієнтів розширеної матриці. Його програма вміла вирішувати лише систему з трьох рівнянь із трьома невідомими.

Другий студент, назвемо його "середнім програмістом", зумів написати програму без goto, але також діяв у "чоло". Щоправда, його програма вже "вміла" вводити коефіцієнти розширеної матриці.

І, нарешті, третій студент, назвемо його "хорошим програмістом", зумів написати дуже витончену програму. Його програма виявилася набагато коротшою за кількістю операторів у тексті програми. У програмі він використовував динамічні масиви, що дозволило реалізувати алгоритм методу Гауса для будь-якого числа рівнянь. Крім того, частина коду, де безпосередньо реалізується алгоритм методу Гауса, організувала у вигляді процедури.

Як модельний приклад обрана система:

$$\left.\begin{aligned} 2x_1 + 6x_2 - x_3 &= -12 \\ 5x_1 - x_2 + 2x_3 &= 29 \\ -3x_1 - 4x_2 + x_3 &= 5 \end{aligned}\right\}$$ її рішенням є $x_1 = 3;\ x_2 = -2;\ x_3 = 6$

3.4.1.1. Варіант 1 – з goto

```
program Gauss_console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
label
  L1, L2, L3, L4, L5, L6, L7;
var
  a: array[1..3, 1..3] of real;
  b: array[1..3] of real;
  x: array[1..3] of real;
  i, j, k, p, n: integer;
  m, S, t: real;
begin
  {Визначення коефіцієнтів розширеної матриці} n:
  = 3;
  a [1,1]: = 2; a [1,2]: = 6; a[1,3]:=-1; b[1]:=-12;
  a [2,1]: = 5; a[2,2]:=-1; a [2,3]: = 2; b[2]:=29;
  a[3,1]:=-3; a[3,2]:=-4; a [3,3]: = 1; b[3]:=5;


  {Основна частина програми}
  k: = 1;
  L1: i: = k + 1;
  if (a[k, k] = 0) then
  begin
    {перестановка рівнянь}
    p: = k; // В алгоритмі використовується буква l, але
вона схожа на 1
    // Тому використовуємо ідентифікатор p
    L6: if abs(a[i, k]) > abs(a[p, k]) then p:= i;
    if not( i = n) then
    begin
      i:= i + 1;
      goto L6;
    end;
    if p = k then i: = k + 1
```

```
  else
  begin
    j: = k;
    L7: t: = a [k, j];
    a [k, j]: = a [p, j];
    a [p, j]: = t;
    якщо не (j = n) then
    begin
      j: = j + 1;
      goto L7;
    end;
    t: = b [k];
    b[k]:= b[p];
    b [p]: = t;
  end;
end; // кінець блоку перестановки рівнянь L2:
m: = a [i, k] / a [k, k];
a [i, k]: = 0;
j: = k + 1;
L3: a [i, j]: = a [i, j] - m * a [k,
j]; якщо не (j = n) then
begin
  j: = j + 1;
  goto L3;
end;
b [i]: = b [i] - m * b
[k]; if not(i = n) then
begin
  i:= i + 1;
  goto L2;
end;
if not( k= n - 1) then
begin
  k: = k + 1;
  goto L1;
end;
x [n]: = b [n] / a [n,
n]; i:= n - 1;
L4: j: = i +
1; S: = 0;
L5: S: = S - a [i, j] * x
[j]; якщо не (j = n) then
begin
  j: = j + 1;
```

```
    goto L5;
  end;
  x[i]:=(b[i] + S)/a[i, i]; if
  not(i = 1) then
  begin
    i:= i - 1;
    goto L4;
  end;
  for i:= 1 to n do
  writeln('x', i, '=', x[i]:0:4);
  writeln(UTF8ToConsole('Натисніть будь-яку клавішу'));
  readkey;
end.
```

3.4.1.2. Варіант 2 – без goto

```
program Gauss_console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a: array[1..3, 1..3] of real;
  b: array[1..3] of real;
  x: array[1..3] of real;
  i, j, k, p, n: integer;
  m, S, t: real;
begin
  {Введення коефіцієнтів розширеної
  матриці} n: = 3;
  for i:=1 to n do
  begin
    for j:=1 to n do
    begin
      writeln(UTF8ToConsole('Введіть a'), i, j);
      readln (a [i, j]);
    end;
    writeln(UTF8ToConsole('Введіть b'), i);
    readln(b[i]);
  end;


  {Основна частина
  програми} k: = 1;
```

```
while true do
begin
  i: = k + 1;
  if (a[k, k] = 0) then
  begin
    {перестановка рівнянь}
    p: = k; // В алгоритмі використовується буква l, але
    вона схожа на 1
    // Тому використовуємо ідентифікатор p
    while true do
    begin
      if abs(a[i, k]) > abs(a[p, k]) then p:=i;
      if i = n then break;
      i:= i + 1;
      continue;
    end;
    if p = k then i: = k +
    1 else
    begin
      j: = k;
      while true do
      begin
        t: = a [k, j];
        a [k, j]: = a [p,
        j]; a [p, j]: = t;
        if j = n then break;
        j: = j + 1;
        continue;
      end;
      t: = b [k];
      b[k]:= b[p];
      b [p]: = t;
    end;
  end; // кінець блоку перестановки рівнянь
  while true do
  begin
    m: = a [i, k] / a [k,
    k]; a [i, k]: = 0;
    j: = k + 1;
    while true do
    begin
      a [i, j]: = a [i, j] - m * a [k,
      j]; if j = n then break;
      j: = j + 1;
```

```
      continue;
    end;
    b [i]: = b [i] - m * b
    [k]; if i = n then
    break; i:= i + 1;
    continue;
  end;
  if k = n - 1 then break;
  k: = k + 1;
  continue;
end;
{Перевірка існування
рішення} if a[n, n] <> 0
then begin
  x [n]: = b [n] / a [n,
  n]; i:= n - 1;
  while true do
  begin
    j:= i + 1;
    S: = 0;
    while true do
    begin
      S: = S - a [i, j] * x
      [j]; if j = n then
      break; j: = j + 1;
      continue;
    end;
    x[i]:=(b[i] + S)/a[i, i]; if
    i = 1 then break;
    i:= i - 1;
    continue;
  end;
  for i:= 1 to n do
  writeln('x', i, '=', x[i]:0:4);
end
else
if b[n] = 0 then
  writeln(UTF8ToConsole('Система рівнянь'
  +
          'не має рішення.'))
else
  writeln(UTF8ToConsole('Система рівнянь'+
        'має безліч рішень.'));
writeln(UTF8ToConsole('Натисніть будь-яку клавішу'));
```

```
  readkey;end
.
```

3.4.1.3. Варіант 3 – найкраща реалізація

```
program Gauss_console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a: array of array of real; {матриця коефіцієнтів системи,
 двовимірний динамічний масив}
  vector: array of real; {перетворений
  одновимірний динамічний масив}
  b: array of real;
  x: array of real;
  i, j, k, n: integer;
procedure gauss(var vector: array of real;
                var b: array of real;
                var x: array of real;
                var n: integer);
var
  a: array of array of real; {матриця коефіцієнтів системи,
 двовимірний динамічний масив}
  i, j, k, p, r: integer;
  m, s, t: real;
begin
  SetLength(a, n, n); // Встановлення фактичного розміру
  масиву
   {Перетворення одновимірного масиву на
  двовимірний} k:=1;
  for  i:=0  n-1  do
  for  j:=0  n-1  do
  begin
    a[i,j]:= vector[k];
    k:=k+1;
  end;
  for k:=0 n-2 do begin
    for i:=k+1 n-1 do begin
      if (a[k,k]=0) then
      begin
        {перестановка рівнянь}
```

```
      p:=k; // В алгоритмі використовується буква l, але
      вона схожа на 1
           // Тому використовуємо
      ідентифікатор p for r: = i to n-1 do
      begin
        if abs(a[r,k]) > abs(a[p,k]) then p:=r;
      end;
      if p<>k then
      begin
        for j:= до n-1 до
        початку
          t: = a [k, j];
          a[k,j]:=a[p,j];
          a[p,j]:=t;
        end; t: = b
        [k];
        b[k]:=b[p];
        b[p]:=t;
      end;
    end; // кінець блоку перестановки рівнянь
    m:=a[i,k]/a[k,k];
    a[i,k]:=0;
    for j:=k+1 n-1 do
    begin
      a[i,j]:=a[i,j]-m*a[k,j];
    end;
    b[i]:= b[i]-m*b[k];
  end;
end;
{Перевірка існування
рішення} if a[n-1,n-1] <>
0 then begin
  x[n-1]:=b[n-1]/a[n-1,n-
  1];for i:=n-2 downto 0 do
  begin
    s:=0;
    for j:=i+1 n-1 do
    begin
      s:=sa[i,j]*x[j];
    end;
    x[i]:=(b[i] + s)/a[i,i];
  end;
  writeln('');
  writeln(UTF8ToConsole('Рішення:'));
```

```
    writeln('');
    for i:=0 to n-1 do
    writeln('x', i+1, '=', x[i]:0:4);
  end
  else
  if b[n-1] = 0 then
    writeln(UTF8ToConsole('Система не має
    рішення.'))
  else
    writeln(UTF8ToConsole('Система
    рівнянь'+ 'має безліч рішень.'));
  writeln('');
  {звільнення пам'яті,
  розподіленої для динамічного масиву a:=nil;
end;
  {Початок основної
програми} begin
  {Введення коефіцієнтів розширеної матриці}
  writeln(UTF8ToConsole('Введіть кількість невідомих'));
  readln(n);
  {Установка реальних розмірів динамічних
  масивів} SetLength(a, n, n);
  SetLength(vector, n*n);
  SetLength(b, n);
  SetLength(x, n);
   {у динамічних масивах індекси починаються
  з нуля} for i:=0 to n-1 do
  begin
    for j:=0 to n-1 do
    begin
      writeln(UTF8ToConsole('Введіть a'), i+1, j+1);
      readln(a[i,j]);
    end;
    writeln(UTF8ToConsole('Введіть b'), i+1);
    readln(b[i]);
  end;
  {Перетворення двовимірного масиву в
  одновимірний} k:=1;
  for  i:=0  n-1  do
  for  j:=0  n-1  do
  begin
    vector[k]:=a[i,j];
```

```
    k:=k+1;
  end;
  {Виклик процедури розв'язання системи
  лінійних рівнянь алгебри методом Гауса}
  gauss(vector, b, x, n);
  {звільнення пам'яті,
  розподіленої для динамічних масивів a:=nil;
  vector:=nil;
  x:=nil;
  b:=nil;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
 end.
```

Зверніть увагу, що масив "a" в головній програмі та масив "a" у процедурі - це різні масиви, у тому сенсі, що вони будуть при виконанні програми займати абсолютно різні ділянки пам'яті. Хоча за змістом завдання це та сама матриця коефіцієнтів, тому їм присвоєно однакове ім'я. Перед передачею матриці коефіцієнтів у процедуру двовимірний масив перетворюється на одновимірний, який і передається, а всередині процедури одновимірний масив перетворюється назад на двовимірний масив з ім'ям "a".

В принципі, можна реалізувати алгоритм і з одновимірним масивом, головне не заплутатися в індексах. Пропоную вам самим написати таку програму. Організуйте введення коефіцієнтів системи одразу в одновимірний динамічний масив та реалізуйте з ним алгоритм методу Гауса.

Загальним недоліком всіх трьох програм є відсутність контролю за введення коефіцієнтів системи. Однак це зроблено навмисно, щоб не відволікатися від суті завдання та не захаращувати програми зайвими деталями. Програми і так виходять чималими. Ви можете самостійно вставити перевірку під час введення, використовуючи способи, викладені в 2.1.25. Більш просунуті та професійні способи контролю ми розглядатимемо в розділі 6.

3.5. Модулі у Паскалі

Модуль це така програмна одиниця, яка може і найчастіше компілюється окремо та незалежно від головної програми. У модулі можуть міститися визначення та реалізації функцій і процедур, що часто використовуються, а також константи, змінні, оголошені типи. Це дозволяє поділити роботу між програмістами при розробці великих та складних програм. Наприклад, один програміст може розробляти один модуль, другий програміст інший модуль, а третій головну програму.

Розподілом роботи, у тому числі які функції та процедури повинні утримуватися в тому чи іншому модулі, що вони повинні робити, які результати вони повинні видавати, їхній інтерфейс, тобто. список та типи формальних параметрів, як правило, займається керівник проекту.

При цьому кожен програміст пише та налагоджує свій модуль незалежно від інших.

Текст модуля мовою програмування називається вихідним модулем. Після компіляції створюється про об'єктний модуль. Об'єктний модуль це такий модуль, який вже переведений на внутрішню машинну мову. На етапі компонування (складання) необхідні модулі включаються до програми (компонуються) для забезпечення правильних викликів функцій та процедур. Наприклад, головна програма викликає певну функцію з модуля А, в модулі відбувається виклик процедури з модуля С і т.д. Такий процес називається дозволом зв'язків. У результаті збирається виконувана програма.

3.5.1 Структура модуля

Структура модуля має вигляд:

```
unit <Ім'я модуля>;
interface          //     розділ інтерфейсу
  <розділ відкритих описів>
implementation //        розділ реалізації
  <розділ закритих описів> initialization //
Розділ ініціалізації finalization          //розділ
завершення end.
```

Модуль починається зі службового слова unit, за яким слідує ім'я модуля. Якщо цей модуль використовує інші модулі, після слова interface необхідно помістити службове слово uses і список модулів, що використовуються.

Інтерфейсний розділ модуля починається зі службового слова interface. У цьому розділі можна визначати константи, типи даних, змінні, процедури та функції, доступні для всіх програм і модулів, які використовують цей модуль. Глобальні змінні, розміщені в інтерфейсній секції, можуть бути використані в основній програмі.

Розділ реалізації модуля розпочинається службовим словом implementation. У секції реалізації можуть бути свої описи, невидимі для програм і модулів, які використовують цей модуль. Описані в секції інтерфейсу константи, типи даних, змінні, процедури та функції є видимими у секції реалізації.

Ті процедури та функції, які описані в інтерфейсній секції, описуються ще раз у секції реалізації, причому їх заголовок має бути точно таким самим, як той, який зазначений у секції інтерфейсу.

У секції ініціалізації розміщуються оператори, що виконуються лише один раз при зверненні до цього модуля основної програми. Ці опера-

ри зазвичай використовуються для підготовчих операцій. Наприклад, у секції ініціалізації можуть ініціалізуватись змінні, відкриватися потрібні файли. Під час виконання програми, яка використовує певний модуль, секція ініціалізації цього модуля викликається перед запуском основного тіла програми. При використанні кількох модулів їх секції ініціалізації викликаються в порядку, зазначеному в uses. Секція ініціалізації є необов'язковою і може взагалі бути відсутнім.

Розділ finalization також є необов'язковим. У ньому виконуються різні дії перед закриттям програми, наприклад, закриття файлів баз даних, звільнення динамічно розподіленої пам'яті тощо.

З метою зменшення розміру основної програми, покращення читабельності готові підпрограми рекомендується оформляти у вигляді модуля. Згодом у вас з'явиться власна бібліотека модулів.

Ім'я бібліотечного модуля повинне співпадати з ім'ям файлу, під яким зберігається текст модуля на диску. Вихідний текст бібліотечного модуля має розширення *.pas (<ім'я модуля>.pas).

Як приклад оформимо програму обчислення синуса з розділу 3.1.1.3 як модуля.

У меню Файл виберіть Створити модуль. У вікні редактора вихідного коду з'явиться заготовка коду для модуля. Очистіть вікно редактора та введіть текст модуля:

```
unit my_module;
interface
   function No_standard_sin(x:real):real;
implementation
   {повторюємо заголовок функції
   точно таким, що і в розділі
   interface}
   function No_standard_sin(x:real):real;
   var eps, s, t: real;
```

```
    n: integer;
begin
  s: = x; t: = x; n: = 2; eps:= 1e-
  7; repeat
    t: = -t * (sqr (x) / (n * (n +
    1))); s: = s + t;
    n: = n + 2;
  until abs(t)<eps;
  No_standard_sin:=s;
end;
end.
```

Збережіть модуль обов'язково з тим самим ім'ям, що вказаний у заголовку модуля, тобто. my_module.pas у якійсь папці. Відкомпілюйте модуль натиснувши клавіші Ctrl+F9 або меню Запуск-> Зібрати. Для кожного модуля створюються два файли: двійковий файл опису модуля з розширенням (.ppu) та об'єктний з розширенням (.o). Таким чином, у вашій папці створяться два файли my_module.ppu та my_module.o

Створіть консольну програму та введіть текст програми:

```
program project_with_my_module;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil,SysUtils, my_module; // вказуємо ім'я
  модуля
var x, y, dx: real;
begin
  x: = 0;
  dx: = 0.1;
  while x <= 1 do
    begin
```

```
      y:= No_standard_sin(x);
      writeln('x=', x:0:1, 'y=', y:0:7,
              'sin(x)= ', sin(x):0:7); x:
      = x + dx;
    end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Збережіть проект у тій же папці, де ви зберегли модуль або скопіюйте файли my_module.ppu та my_module.o до папки з поточним проектом. Натисніть клавішу F9. Почнеться компіляція та складання (компонування) програми. На етапі складання ваш модуль буде автоматично доданий у програму, що виконується. Після виконання програми ви отримаєте ті самі результати, що й у розділі 3.1.1.3.

Якщо ви хочете, щоб ваш модуль був доступний і в інших програмах, то найпростіше зробити так:

1. Створіть папку, наприклад, з ім'ям my-units
2. Перенесіть файли my_module.ppu та my_module.o в цю папку;
3. Скопіюйте саму папку my-units всистемний каталог "C:\lazarus\fpc\2.2.4\units\i386-win32" якщо ви працюєте у Windows або в каталог "/usr/lib/fpc/2.2.4/units/i386-linux", якщо ви працюєте в Linux.

Тепер ваш модуль буде автоматично підключатися до ваших програм (зрозуміло, треба прописати імена модулів в оголошенні uses). Чому саме цей каталог? Тому що компілятор, якщо не знайде модуль у папці з вашим проектом, він буде шукати його за умовчанням у цій системній папці.

Якщо ж ви розмістили папку my-units в іншому місці, вам доведеться прописувати шлях до цієї папки, або в інспекторі проекту, або безпосередньо в оголошенні uses ось таким чином:

```
uses my_module in '<шлях до папки з вихідним кодом модуля>'
```

Інший спосіб – це додати шлях до модуля у файлі fpc.cfg. Файл знаходиться в папці \lazarus\fpc\2.2.4\bin\i386-win32 у Windows або в папці /etc у Linux. Знайдіть у цьому файлі рядок

searchpath for units and other system dependent things

І додайте запис вигляду:

```
-Fшлях до модуля (саме без пробілу!), Наприклад
```

```
-Fus:\my-projects\my-units\
```
(Windows)

```
-Fu/home/user/my-projects/my-units
```
(Linux)

Але цей файл системний, "лізти" в нього без особливої необхідності не рекомендую.

У запропонованому мною способі "нічого не треба робити"! Потрібно просто скопіювати папку my-units у системний каталог. Шлях до нього вже записаний у файлі fpc.cfg. Тільки щоб не "засмічувати" системний каталог, вам потрібно всі свої модулі, хоч би скільки їх було, поміщати тільки в одну папку, у нашому випадку ми будемо розміщувати всі наші спільні модулі в папку my-units.

3.5.2 Системні модулі

Системні модулі Паскаля містять низку корисних процедур, функцій та констант, які дають змогу використовувати майже всю міць комп'ютерів. Розглянемо коротко деякі системні модулі:

System – містить стандартні функції та процедури

"класичного" Паскаля, наприклад, обчислення функції cos, sin, ln, exp та ін.

System автоматично доступний для всіх програм. Фактично ми широко користувалися цим модулем.

DOS містить процедури і функції, що дозволяють використовувати засоби операційної системи MS-DOS.

CRT – набір процедур для роботи з екраном та клавіатурою.

Graph містить великий набір програм, який дозволяє використовувати графічні можливості комп'ютера.

В даний час цей модуль застарів і практично не використовується. Логіка його роботи слабо сумісна з сучасними системами і проблем з ним безліч. Початківці часто знаходять якісь приклади з його використанням та розчаровуються - частина не працює взагалі, у Linux виникають проблеми з бібліотеками, правами доступу тощо.

Для використання модуля достатньо в програмі після рядка заголовка вставити рядок:

```
uses ім'я модуля;
```

Якщо використовується кілька модулів, пишеться так:

```
uses ім'я модуля 1 ім'я модуля 2, ….., ім'я модуля N;
```

Переглянути той чи інший системний модуль можна, натиснувши клавішу Ctrl і одночасно підвівши покажчик миші до імені модуля. Коли ім'я модуль заміниться на гіперпосилання, натисніть ліву клавішу миші.

Зокрема, якщо ви подивіться модулі SysUtils, FileUtil, LCLProc та ін., то побачите там оголошення та реалізації тих функцій та процедур, якими ми широко користувалися.

Рекомендую частіше заглядати в ці та інші системні модулі. Ви

знайдете для себе чимало цікавого та повчального. Заодно погляньте, як пишуть програми професіонали високого рівня.

3.5.2.1. Модуль CRT

Модуль CRT, як уже говорилося, містить низку процедур та функції для роботи з екраном у текстовому режимі, клавіатурою та динаміком. Зауважу, що цей модуль залишено для сумісності з Turbo Pascal. В даний час програмісти частіше використовують інші засоби.

Розглянемо екран комп'ютера у текстовому режимі. Зазвичай на ньому розміщується 25 рядків тексту та 80 символів у кожному рядку. Для того, щоб повністю володіти екраном, потрібно знати координати курсору, змінювати колір символів, тексту та фону. Для цього існують такі процедури:

- `GoToXY(i,j)- позиціонує курсор у і рядок і j стовпець екрану.`
- `write(s)- виводить рядок s починаючи з поточної позиції курсору.`
- процедура TextColor (Color) - встановлює колір тексту, що виводиться.

Існують такі константи кольорів:

```
Black – чорний;
Blue – синій;
Green – зелений;
Cyan –
бірюзовий; Red –
червоний;
Magenta – малиновий;
Brown – коричневий;
LightGray – світло-
сірий; DarkGray –
темно-сірий; LightBlue
```

```
– яскраво-блакитний;
LightGreen – яскраво-зелений;
```

```
LightCyan - яскраво-
бірюзовий; LightRed –
яскраво-червоний;
LightMagenta – яскраво-
малиновий; Yellow – жовтий;
White – білий;
Blink - мерехтіння (миготіння);
```

- Процедура TextBackGround (Color) - встановлює колір тла.
- Процедура ClrScr – очищає екран, одночасно забарвлюючи його у колір встановленого фону.
- Процедура Window(x1, y1, x2, y2) - визначає на екрані вікно, де x1, y1 - координати лівого верхнього кута, а x2, y2 - координати правого нижнього кута вікна

Функція KeyPressed – повертає значення true, якщо клавіша натиснута та false – в іншому випадку.

Функція readkey – зчитує символ із клавіатури.

приклад.

Написати програму, яка очищає екран та встановлює синій фон. Потім текст виводить жовтим кольором.

```
program project1;
uses
  CRT, FileUtil;
begin
  TextBackGround(Blue);
  ClrScr;
  GoToXY(6, 6);
  TextColor(Yellow);
```

```
  writeln(UTF8ToConsole('Привіт, Васю!'));
  writeln(UTF8ToConsole('Натисніть будь-яку клавішу'));
  readkey;
end.
```

Подальші "ігри" з квітами та вікнами надаю вам самому, шановний читачу, а зараз давайте напишемо більш серйозну програму.

приклад.

У компанії з продажу комп'ютерів є менеджери, прізвища яких, зберігаються у масиві Name. В іншому масиві є дані про кількість проданих комп'ютерів кожним менеджером за місяць. Написати програму, яка виводить таблицю, в якій такі графи: прізвище, кількість проданих комп'ютерів, сума виручки, сума премії. Підсумковий рядок має містити загальну кількість проданих компанією комп'ютерів за місяць, загальну суму виручки, суму преміальних.

Вартість комп'ютера прийняти рівною 1000 $, розмір премії за кожен проданий комп'ютер 25 $

```
program manager;
uses
  CRT, FileUtil, SysUtils;
var
  name: array of string[18];
  kol: array of integer;
  sum, cost, prem, i, k, n: integer;
  progprem: integer;
begin
  writeln(UTF8ToConsole('Введіть кількість менеджерів'));
  readln(n);
  SetLength(name, n);
```

```
SetLength(kol, n);
for k:= 0 to n - 1 do
begin
  writeln(UTF8ToConsole('Введіть прізвище'));
  readln(name[k]);
  writeln(UTF8ToConsole('Введіть кількість
  комп'ютерів')); readln(kol[k]);
end;
ClrScr;
GoToXY(6,1);
write(UTF8ToConsole("Відомості про реалізацію
комп'ютерів")); GoToXY (14,2);
write(UTF8ToConsole('за січень 2010 р.'));
GoToXY(1,3);
write('------------------------------');
GoToXY(1,4);
write(UTF8ToConsole('ПрізвищеКількість Виторг Премія'));
GoToXY(1,5);
write('------------------------------');
cost:= 1000;
progprem: = 25;
for k:= 0 to n - 1 do
begin
  sum: = cost * kol [k];
  prem:= progprem *
  kol[k]; writeln;
  writeln(name[k]);
  GoToXY(24, k + 6);
  write(kol[k]);
```

```
    GoToXY(32, k + 6);
    write(sum);
    GoToXY(40, k + 6);
    write(prem);
  end;sum:
  = 0;
  i:= 0;
  for k:= 0 to n - 1 do
  begin
    i:= i + kol[k];
    sum: = i * cost; {Сума виручки}
    prem: = i * progprem;
  end;
  GoToXY(1, n + 6);
  write('------------------------------');
  GoToXY(17, n + 7);
  writeln(UTF8ToConsole('Разом:'));
  GoToXY(24, n + 7);
  write(i);
  GoToXY(32, n + 7);
  write(sum);
  GoToXY(40, n + 7);
  write(prem);
  GoToXY(1, n + 9);
  write(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

3.6. Файли

Файл – деяка ділянка на диску, де зберігаються дані. Кожному файлу надається унікальне ім'я, завдяки якому стає можливим звертатися до різних даних на диску. Файли складаються з окремих елементів одного типу. Розмір файлу залежить від обсягу фізичного пристрою, на якому він зберігається.

3.6.1 Тип даних – запис

Запис - сукупність даних, що мають свої імена та тип, об'єднаних одним іменем. Запис – комбінований тип даних. Кожен запис складається з окремих полів, які можуть мати різні типи.

Загальний вигляд оголошення запису:

```
type
   <ім'я запису> = record
   <ім'я1>: <тип1>;
   <ім'я2>: <тип2>;
   ..............
   <ім'яN>: <типN>;
  end;
```

Після цього можна оголосити змінні з типом запису:

```
var <ім'я змінної>: <ім'я запису>;
```

приклад. Нехай є таблиця наступної структури:

Прізвище	Ім'я	По-батькові	Група	Рік народження

Тоді можна оголосити запис такого вигляду:

```
type
```

```
    student = record
    family, name, otch, group: string[20];
    birthday: integer;
  end;
  var
    a, b: student;
```

Звернення до елементів запису виконується через уточнене (складене) ім'я.

```
a.family:= 'Іванів';
b.group:= 'ПОВТАС-1/08';
```

Щоб не подовжувати надто текст програми, у Паскалі передбачено оператора приєднання with. Це дає можливість визначити шматок програми, всередині якого можна просто вказувати необхідні поля запису, не вказуючи щоразу при цьому ім'я самого запису.

Синтаксис оператора:

```
with <список імен записів> do
begin
  … {в цьому блоці при зверненні до полів запису не обов'язково каж-
вказувати ім'я запису}
  ……
end;
```

Приклад:

```
type
   student = record
   family, name, otch, group: string[20];
end;
```

```
var
  a, b: student;
begin
  with b do
  begin
    family: = 'Іванів';
    group:= 'ПОВТАС-1/08';
  end;
end.
```

3.6.2 Файлові типи

У програмі роботи з файлами вводяться файлові змінні. Змінні файлові типи необхідні тим програмам, яким потрібно читати дані з диска або записувати дані на диск. Синтаксис опису файлової змінної:

`var <ім'я файлової змінної>: <тип файлу>; Розрізняють три типи файлів:`

- текстовий файл;
- типізований файл;
- нетипізованийфайл

Текстовий файл складається з послідовності будь-яких символів змінної довжини, поділених на рядки. Кожен рядок закінчується спеціальною ознакою "кінець рядка" - eoln (end of line).

Опис текстового файлу має вигляд:

`var <ім'я файлової змінної>: TextFile;`

Типізований файл складається із елементів одного типу. Однак програміст може створити свій тип даних – запис з різнорідними типами

лей даних. Таким чином, взагалі кажучи, у типізованому файлі можна зберігати дані різних типів, але лише у структурованому вигляді – як записів.

Опис типізованого файлу має вигляд:

```
var <ім'я файлової змінної>: File of <тип компонентів>;
```

І, нарешті, у нетипізованих файлах зберігаються дані будь-якого типу та структури. Опис нетипізованого файлу має вигляд:

```
var <ім'я файлової змінної>: File;
```

приклад.

```
var
  name: TextFile; // Текстовий файл
  kol: File of integer; // Типізований файл
  buf: File; // Нетипізований файл
```

Файл name складається з послідовності рядків будь-яких символів змінної довжини, файл kol з цілих чисел, buf файл містить дані будь-якого типу і структури, тобто. інтерпретуються незалежно від типу та структури даних, що містяться в ньому.

Для всіх типів файлів визначається спеціальна ознака "кінець файлу" - eof (end of file).

3.6.3 Процедури для роботи з файлами

3.6.3.1. Загальні процедури роботи з файлами всіх типів

Для зв'язку файлової змінної із файлом у зовнішній пам'яті використовується процедура:

```
AssignFile(f, fname);
```

де f - файлова змінна, fname - змінна типу string, що містить ім'я файлу. Процедура AssignFile – пов'язує ім'я зовнішнього файлу на диску із змінною файловою.

Загальний вигляд імені файлу:

<диск>:\<ім'я каталогу>\<ім'я підкаталогу>\...\<ім'я файлу> (Windows)

/<ім'я каталогу>/<ім'я підкаталогу>/.../<ім'я файлу> (Linux)

Якщо вказано лише ім'я файлу, то приймається поточний каталог та поточний логічний пристрій. Тільки після виконання AssignFile можна звертатися до інших процедур обробки файлів.

```
AssignFile(name, 'Name.dat')
AssignFile(kol, 'C:\Work\Kol.dat')
AssignFile(data, '/home/user/data.dat')
```

Процедура `Rewrite` - створює і відкриває новий файл. Опис:

```
Rewrite(f)
```

f – є файловою змінною. Перед використанням Rewrite змінна f повинна бути пов'язана з дисковим файлом за допомогою AssignFile. Якщо файл із такою назвою вже існує, він видаляється, але в його місці створюється новий порожній файл.

Процедура Reset- відкриває існуючий файл для читання або запису. Опис:

```
Reset(f)
```

де f – файлова змінна. Перед використанням Reset змінна f повинна бути пов'язана з файлом на диску за допомогою AssignFile. Після створення або відкриття будь-якого файлу створюється спеціальний указ-

тель, за допомогою якого відстежується розташування поточного елемента у файлі. Після прочитання чергового елемента файлу покажчик автоматично переміщується до наступного елемента. Після запису вказівник автоматично переміщається до елемента, який щойно записаний у файл на диску.

Процедура CloseFile (f) – закриває відкритий файл.

Функція Eof() у випадку, якщо кінець файлу досягнуто, повертає значення true і false , якщо кінець файлу ще не досягнутий.

Процедура `Rename(f, newname);` дозволяє перейменувати файл; `newname – змінна типу string, що містить нове ім'я файлу; Процедура Erase(f); - видаляє файл.`

Функція IOResult – дозволяє визначити результат останньої операції вводу/виводу. Повертає 0, якщо введення/виведення пройшло успішно. Якщо операція введення/виводу пройшла невдало, повертає код помилки.

Функція FileExists(fname) – дозволяє визначити, чи існує файл fname на диску. Тут fname – змінна типу string, у ній задається ім'я файлу на зовнішньому пристрої, найчастіше на диску. Дозволяється вказувати ім'я файлу разом із шляхом до файлу, наприклад

`FileExists('C:\Work\Data\Student.dat') - повертає true, якщо файл існує і false, якщо файл з цим ім'ям у вказаній папці не існує.`

3.6.3.2. Процедури для роботи з текстовими файлами

Текстові файли є файлами із послідовним доступом. Доступ до кожного рядка можливий лише послідовно, починаючи з першого. Якщо потрібно, наприклад, прочитати рядок n, спочатку потрібно прочитати всі n-1 рядків.

Процедура Reset(f) відкриває текстовий файл лише для читання.

Процедура Rewrite(f) створює новий файл (якщо файл з тим самим ім'ям)

вже існував, то він знищується) і відкривається лише для запису.

Процедура Append(f) відкриває файл для запису та дозволяє додавати нові елементи до кінця файлу.

Процедура Read(f, v)- зчитує змінну v черговий елемент текстового файлу, відповідного поточному положенню покажчика. Якщо проводиться зчитування відразу в кілька змінних, то вони поділяються комами, наприклад:

```
Read(f, v1, v2, v3);
```

При цьому після зчитування значення елемента файлу в останню змінну списку (v3) покажчик залишається на місці. Допускається використання різних типів змінних.

Процедура Readln(f, v)- зчитує в змінну v черговий елемент текстового файлу, що відповідає поточному положенню покажчика. Якщо проводиться зчитування відразу в кілька змінних, то вони поділяються комами, наприклад:

```
Readln(f, v1, v2, v3);
```

При цьому після зчитування значення елемента файлу в останню змінну списку (v3), елементи, що залишилися, аж до символу eoln пропускаються, а покажчик зсувається на наступний рядок. У цьому полягає відмінність Readln від Read.

Тому, якщо в текстовому файлі є кілька рядків, процедурою Read можна прочитати лише один рядок, а процедурою Readln кілька рядків, але для цього треба процедуру Readln викликати стільки разів, скільки необхідно.

Виклик процедури у вигляді Readln(f) дозволяє пропустити рядок у файлі, не зчитуючи його вміст.

При використанні процедур Read і Readln відбувається автоматичне перетворення прочитаних символів у типи змінних, які

вказані у списку введення. Під час читання змінних типу char виконується читання одного символу та надання ліченого значення змінній. Під час читання з файлу значення рядкової змінної, довжина якої явно задана в її оголошенні, зчитується стільки символів, скільки зазначено в оголошенні, але не більше, ніж у поточному рядку.

При читанні з файлу значення рядкової змінної, довжина якої явно не задана в оголошенні змінної, значенням змінної стає частина поточного рядка, що залишилася після останнього читання.

Якщо однією інструкцією Readln здійснюється введення кількох, наприклад, двох змінних, то перша змінна міститиме стільки символів, скільки зазначено в її оголошенні або, якщо довжина не вказана, весь рядок файлу. Друга змінна буде містити символи, що залишилися, поточного рядка або, якщо таких символів немає, не міститиме жодного символу (довжина рядка дорівнює нулю).

Під час читання числових значень алгоритм роботи процедур введення змінюється. Починаючи з поточного положення вказівника виділяються значущі символи до першого пробілу, знаку табуляції або ознаки кінця рядка. Отримана послідовність символів вважається символьним поданням числа і робиться спроба перетворення його у внутрішнє уявлення числа (цілого або речового, залежно від типу змінної, що вводиться). Якщо перетворення пройшло успішно, відповідній змінній надається значення цього числа. Інакше фіксується помилка, тобто. це означає, що у символьному поданні числа зустрічається неприпустимий символ. Такі помилки можна опрацювати програмно. Способи фіксації та обробки помилок введення/виводу ми розглянемо у розділі 3.6.3.5.

Процедура Write(f, v)- записує значення змінної v файл. Тут f, як і раніше, файлова змінна. Якщо виконується запис значень відразу кількох змінних, вони поділяються комами, наприклад:

```
Write(f, v1, v2, v3);
```

Процедура Writeln(f, v)- записує значення змінної v файл і додає символ кінця рядка eoln. Якщо виконується запис значень відразу кількох змінних, вони поділяються комами, наприклад:

```
Writeln(f, v1, v2, v3);
```

У текстових файлах при записі відбувається автоматичне перетворення типів змінних у списку виводу на їх символьне уявлення.

Процедура Writeln(f) дозволяє записати у файл порожній рядок.

Щоб перевірити, чи не є поточний елемент символом кінця рядка eoln, можна використовувати функції EoLn(f) і SeekEoLn(f). Функція EoLn(f) перевіряє лише поточну позицію покажчика на ознаку кінця рядка, а SeekEoLn(f) спочатку пропускає пробіли та символи табуляції і потім перевіряє на ознаку кінця рядка.

Так само функція SeekEof(f) спочатку пропускає прогалини та символи табуляції і лише потім перевіряє файл на ознаку кінця.

Розглянемо програму з розділу 3.5.2.1. З цією програмою було незручно працювати, т.к. постійна інформація (прізвища, кількість проданих комп'ютерів) не зберігалася і при кожному запуску програми доводилося заново вводити їх. Проблема вирішується, якщо постійну інформацію зберігати на диску.

Напишемо програму створення файлів на диску і додавання в них дано них:

```
program create_files;
{$mode objfpc}
uses
  {У модулі SysUtils міститься функція
  FileExists} CRT, FileUtil, SysUtils;
var
  name: TextFile; // Файлова змінна
```

```
  kol: TextFile; // Файлова змінна
  comp, k, n:integer;
  fam: string[18];
begin
  AssignFile(name, 'Name.txt'); AssignFile(kol,
  'Kol.txt');
  {Перевірка файлів. Якщо файли
  існують, вони відкриваються для
  дозапису. Якщо ні, то створюються
  нові порожні файли} if not
  FileExists('Name.txt') then
    Rewrite(name)
  else
    Append(name);
  if not FileExists('Kol.txt') then
    Rewrite(kol)
  else
    Append(kol);
  writeln(UTF8ToConsole('Введіть кількість менеджерів));
  readln(n);
  for k:=1 to n do
  begin
    writeln(UTF8ToConsole('Введіть прізвище'));
    readln(fam);
    Writeln(name, fam); // запис у файл
    writeln(UTF8ToConsole('Введіть кількість
    комп'ютерів')); readln(comp);
    Writeln(kol, comp); // запис у файл
  end;
  writeln(UTF8ToConsole('Інформація на диск записана'));
```

```
  CloseFile(name);
  CloseFile(kol);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Напишемо програму обробки файлів:

```
program manager;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: TextFile;
  kol: TextFile;
  sum, cost, prem, k: integer;
  comp, sumc, sumv, sump, sum1: integer;
  fam: string[18];
begin
  {За потреби вкажіть повний шлях
  до файлів або скопіюйте ці
  файли до папки
  з даним проектом або збережіть сам проект у
  папці, де створено проект create_files.lpr}
  if (not FileExists('Name.txt')) or
     (not FileExists('Kol.txt')) then
  begin
    writeln(UTF8ToConsole('Файли не існують'));
    writeln(UTF8ToConsole('Спочатку створіть їх'));
    writeln(UTF8ToConsole('Натисніть будь-яку клавішу'));
```

```
  readkey;
  exit;
end;
begin
  AssignFile(name, 'Name.txt'); AssignFile(kol,
  'Kol.txt');
  // файли відкриваються для читання
  Reset(name);
  Reset(kol);
end;
ClrScr;
GoToXY(6, 1);
write(UTF8ToConsole("Відомості про реалізацію
комп'ютерів")); GoToXY (14, 2);
write(UTF8ToConsole('за січень 2010
p.')); GoToXY(1, 3);
write('------------------------------');
GoToXY(1, 4);
write(UTF8ToConsole(' Прізвище Кількість Виручка
Премія')); GoToXY(1, 5);
write('------------------------------');
cost:= 1000; // Вартість комп'ютера
prem: = 25;  // розмір премії
sumc: = 0;   // Загальна кількість комп'ютерів
sumv: = 0;   // Загальна сума виторгу
sump: = 0;   // Загальна сума премії
k: = 0;      // Відстежує поточний рядок на екрані
while not Eof(name) do
```

```
begin
  Readln(name, fam);
  Readln(kol, comp);
  sum:=comp*cost;   // сума виручки для одного менеджера
  sum1:=comp*prem;  // сума преміальних для одного менеджера
  sumc:=sumc+comp;  // всього продано комп'ютерів
  sumv:=sumv+sum;   // виторг від продажу всіх комп'ютерів
  sump:=sump+sum1;  // загальна сума преміальних
  writeln;
  writeln(fam);
  GoToXY(24, k + 6);
  write
  (comp);
  GoToXY(32, k + 6);
  write(sum);
  GoToXY(40, k + 6);
  write(sum1);
  k: = k + 1;
end;
GoToXY(1, k + 6);
write('------------------------------');
GoToXY(17, k + 7);
write(UTF8ToConsole('Разом:'))
;GoToXY(24, k + 7);
write(sumc);GoToXY(32
, k + 7);
write (sumv);
GoToXY(40, k + 7);
write(sump);GoToX
Y(1, k + 9);
```

```
  writeln(UTF8ToConsole('Натисніть будь-яку клавішу'));
  CloseFile(name);
  CloseFile(kol);
  readkey;
end.
```

Чому ми поділили ці програми на дві самостійні? Щоб відокремити процес введення безпосередньо від обробки. Так часто роблять на практиці. Адже введенням даних може займатися, наприклад, оператор. Причому дані мають запроваджуватися, за ідеєю, щодня. А програма видачі зведених даних запускається раз на місяць.

3.6.3.3. Процедури для роботи з типовими файлами

Процедура Read(f, v)- зчитує змінну v черговий елемент файла. Тип змінної v повинен відповідати файловій змінній f.

Процедура Write(f, v)- записує змінну v файл. f-файлова змінна, v-змінна того ж типу, що і елемент файлу f.

Для типізованих файлів застосування процедур Writeln та Readln заборонено.

Типізовані файли на відміну текстових є файлами прямого доступу, тобто. дозволяють звернутися "відразу" до потрібного елемента за його номером. Таким чином, у типізованих файлах усі його елементи мають свій номер. Нумерація елементів файлу починається із 0.

Для переміщення типизованим файлом застосовується процедура Seek. Формат виклику процедури:

```
  Seek(f, n);
```

Де f – файлова змінна, n – номер елемента у файлі. Процедура Seek просто встановлює покажчик елемент із номером n.

Функція FilePos(f) повертає поточну позицію покажчика, тобто. номер того елемента, на який встановлено покажчик.

Функція FileSize(f) повертає кількість елементів у файлі.

Напишемо програму попереднього розділу з використанням типованих файлів. У цьому прикладі ми створили два файли, кожен елемент яких складався лише з одного поля.

Файл Name

Іванов
Петров
……
Сидорів
Яковлєв

ФайлK o l

2
10
15
0

Як ми зазначали, типизированные файли дозволяють організувати зберігання даних складнішої структури, тобто. складаються з кількох полів, що мають різні типи. Створимо один файл з ім'ям "File_manager.dat", що має таку структуру запису:

name	comp
Іванов	2
Петров	10
……	
Сидорів	15
Яковлєв	0

Для цього спочатку створимо тип даних запис наступної структури:

```
type
    manager = record
```

```
        name: string[18];
        comp: integer;
    end;
```

Програма створення та введення даних у типізований файл буде виглядати так:

```
program create_files;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
type
  manager = record
  name: string
  [18]; comp:
  integer;
end;
var
  company: manager;
  fmanager: File of manager; // Файлова змінна
  k: integer;
  n: integer;
begin
  AssignFile(fmanager, 'File_manager.dat');
  {Перевірка  файлів.  Якщо  файли
  існують,  вони  відкриваються  для
  дозапису. Якщо ні, то створюються
  нові пусті файли}
  if not FileExists('File_manager.dat') then
    Rewrite(fmanager)
```

```
  else
```

```
  Reset(fmanager);
  Seek(fmanager, System.FileSize(fmanager));
  while not Eof(fmanager) do
     Read(fmanager, company);
  writeln(UTF8ToConsole('Введіть кількість менеджерів'));
  readln(n);
  with company do
  begin
    for k:= 1 to n do
    begin
      writeln(UTF8ToConsole('Введіть прізвище'));
      readln(name);
      writeln(UTF8ToConsole('Введіть кількість
      комп'ютерів')); readln(comp);
      Write (fmanager, company); // запис у файл
    end;
  end;
  writeln(UTF8ToConsole('Інформація на диск
  записана')); CloseFile(fmanager);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Зверніть увагу, процедура

```
Seek(fmanager, System.FileSize(fmanager));
```

переводить покажчик на кінець файлу, що дозволяє робити дозапис елементів у кінець файлу. Також зверніть увагу на виклик функції FileSize(). Майте на увазі, що існують дві версії функції

FileSize():

- function FileSize(const Filename: string): int64;
- function FileSize(var f: File): int64;

Першу функцію описано в модулі FileUtil, вона повертає повний обсяг файлу в байтах. А друга описана в модулі System, вона повертає кількість записів у файлі. Нам потрібна саме друга функція, тому перед ім'ям файлу ми вказуємо ім'я модуля System, а потім через точку ім'я функції.

Напишемо програму видачі зведених даних. Оскільки ми збираємося надалі використовувати створений типізований файл менеджерів та програму виведення зведеної відомості на екран, оформимо цю програму у вигляді модуля. Тоді програма будемо виглядати так:

```
unit OutScr;
interface
uses
  CRT, Classes, FileUtil, SysUtils, LCLProc;
procedure output_to_screen(var name_file: string);
implementation
procedure output_to_screen(var name_file: string);
type
    manager = record
    name: string [18];
    comp: integer;
end;
var
  company: manager;
  sum, cost, prem, k: integer;
```

```
  sumc, sumv, sump, sum1: integer;
  fmanager: File of manager;
begin
  Assign(fmanager, name_file);
  Reset(fmanager);
  ClrScr;
  GoToXY(6, 1);
  write(UTF8ToConsole("Відомості про реалізацію
  комп'ютерів")); GoToXY (14, 2);
  write(UTF8ToConsole('за січень 2010
  р.')); GoToXY(1, 3);
  write('------------------------------');
  GoToXY(1, 4);
  write(UTF8ToConsole(' Прізвище Кількість Виручка
  Премія')); GoToXY(1, 5);
  write('------------------------------');
  cost:= 1000; // Вартість комп'ютера
  prem:= 25;   // розмір премії
  sumc:= 0;    // Загальна кількість
               комп'ютерів
  sumv:= 0;    // Загальна сума виторгу
  sump:= 0;    // Загальна сума премії
   k: =        // Відстежує поточний рядок на
    0;         екрані
  with company do
  begin
    while not Eof(fmanager) do
    begin
      Read(fmanager, company);
      sum: = comp * cost; // сума виручки одного менеджера
```

```
      sum1:= comp * prem; // сума преміальних одного
                          менеджера
      sumc:= sumc + comp;
      sumv:= sumv + sum;
      sump:= sump + sum1;
      writeln;
      writeln(name);
      GoToXY(24, k + 6);
      write
      (comp);
      GoToXY(32, k + 6);
      write(sum);
      GoToXY(40, k + 6);
      write(sum1);
      k: = k + 1;
    end;
  end;
  GoToXY(1, k + 6);
  write('------------------------------');
  GoToXY(17, k + 7);
  write(UTF8ToConsole('Разом:'))
  ;GoToXY(24, k + 7);
  write(sumc);GoToXY(32
  , k + 7);
  write (sumv);
  GoToXY(40, k + 7);
  write(sump);GoToX
  Y(1, k + 9);
  CloseFile(fmanager);
end;
end.
```

Скопіюйте отримані після компіляції об'єктні модулі (з розширенням OutScr.o та OutScr.ppu) у папку, де знаходяться ваші модулі, що часто використовуються, наприклад у папку my-units (див. розділ 3.5.1).

Програма, яка використовує цей модуль, буде такою:

```
program manager_computer;
uses
  CRT, SysUtils, FileUtil, OutScr;
var
  name_file: string;
begin
  {За потреби вкажіть повний шлях
 до файлів або скопіюйте ці файли
 в папку з цим проектом або
 збережіть проект у папці, де
 створений проект
 create_files.lpr}
  name_file:='File_manager.dat';
  якщо немає FileExists(name_file)
  then begin
    writeln(UTF8ToConsole('Файли не існують'));
    writeln(UTF8ToConsole('Спочатку створіть їх'));
    writeln(UTF8ToConsole('Натисніть будь-яку
    клавішу')); readkey;
    exit;
  end;
  output_to_screen(name_file);writeln(UTF8ToConso
  le('Натисніть будь-яку клавішу')); readkey;
end.
```

Проаналізувавши нашу програму, ми змушені констатувати, що

під час створення файлу відсутня контроль даних, що вводяться. Виправимо цей недолік. Скористаємося в такий спосіб. Вводитимемо кількість менеджерів і кількість комп'ютерів спочатку в символьні змінні. Потім скористаємося функцією Val() для перетворення рядка на число. Як ви знаєте, ця функція має спеціальний параметр і якщо при перетворенні відбулася помилка, функція передає через цей параметр код помилки. Якщо перетворення пройшло успішно, параметр дорівнює нулю. Контролювати введення будемо в циклі для того, щоб у разі помилки під час введення дати можливість користувачу повторити введення.

```
program create_files;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
type
  manager=record
  name:string[18];c
  omp: integer;
 end;
var
  company:manager;
  fmanager: File of manager; // Файлова змінна
  k: integer;
  n: integer;
  code: integer;
  str_n, str_comp: string;
begin
  AssignFile(fmanager, 'File_manager.dat');
  {Перевірка файлів. Якщо файли
```

існують, вони відкриваються для дозапису.

Якщо ні, то створюються нові пусті файли}

```
if not FileExists('File_manager.dat') then
  Rewrite(fmanager)
else Reset
(fmanager);
Seek(fmanager, System.FileSize(fmanager));
while not Eof(fmanager) do
   Read(fmanager, company);
while true do
begin
  writeln(UTF8ToConsole('Введіть кількість менеджерів'));
  readln(str_n);
  Val(str_n, n, code);
  if code = 0 then
    break
  else
    writeln(UTF8ToConsole('Помилка! Повторіть
введення')); end;
with company do
begin
  for k:= 1 to n do
  begin
    writeln(UTF8ToConsole('Введіть прізвище'));
    readln(name);
    while true do
    begin
    writeln(UTF8ToConsole('Введіть кількість
    комп'ютерів')); readln(str_comp);
```

```
      Val(str_comp, comp, code);
      if code = 0 then
        break
      else
        writeln(UTF8ToConsole('Помилка! Повторіть
      введення')); end;
      Write (fmanager, company); // запис у файл
    end;
  end;
  writeln(UTF8ToConsole('Інформація на диск
  записана')); CloseFile(fmanager);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

3.6.3.4. Процедури роботи з нетипізованими файлами

У файлах будь-якого типу читання та запис даних проводиться порціями певної довжини, які називаються блоками або записами. У текстових та типизованих файлах розмір записів визначається автоматично. За цим слідкує операційна система, точніше, її частина, яка називається файловою системою. Для нетипізованих файлів програміст сам повинен стежити за розміром записів, що вводяться і виводяться. Зверніть увагу, що в даному випадку поняття "запис" та тип даних у Паскалі "запис" - це зовсім різні речі. При операціях із файлами під записом розуміється розмір фізичного блоку даних, які записуються чи зчитуються із зовнішнього пристрою.

Нетипізовані файли також є файлами прямого доступу. До них застосовні всі процедури, які використовуються для типізованих файлів, за винятком процедур введення/виводу.

Введення/виведення у нетипізованих файлах здійснюється за допомогою процедур, що мають наступний формат:

```
BlockRead(f, buf, count [, fact_count]);
BlockWrite(f, buf, count[, fact_count]);
```

Процедура BlockRead здійснює читання даних із дискового пристрою, а процедура BlockWrite запис даних.

Тут f – файлова змінна, buf – змінна, вміст якої записується у файл, або дані з файлу зчитуються в цю змінну, count – змінна цілого типу, містить кількість записів, які необхідно прочитати або записати, fact_count – необов'язковий параметр, змінна цілого типу, у ньому міститься фактична кількість прочитаних.

Створити або відкрити файл можна процедурами Rewrite або Reset, при цьому параметри процедур можна додати необов'язковий параметр - розмір записів, наприклад:

```
Rewrite(f, 256);
Reset(f1, 1);
```

Цими процедурами створюється файл f з довжиною запису 256 байтів і відкривається файл f1, розмір запису якого становить 1 байт.

Якщо другий параметр не вказано, розмір запису за замовчуванням приймається рівним 128 байтам. Для максимальної швидкості обміну даними можна вказувати розмір запису кратній величині кластера дискового накопичувача. Однак на практиці користуються розміром запису всього в один байт, що дозволяє обмінюватись даними блоками будь-якої довжини.

Часто потрібно визначити розмір пам'яті, який займає той чи інший об'єкт. Для цього використовується функція SizeOf(x), яка повертає кількість байт, які займає аргумент x у пам'яті.

При написанні нашої улюбленої програми – рішення системи лінійних

алгебраїчних рівнянь методом Гауса введені коефіцієнти не зберігалися. Перепишемо програму так, щоб введені коефіцієнти розширеної матриці зберігалися на диску. Програма записує у нетипізований файл кількість рівнянь системи та коефіцієнти розширеної матриці системи.

```
program Input_coeff;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  matrix: File;
  temp: real;
  i, j, n, count: integer;

begin
  AssignFile(matrix, 'Coeff.dat');
  {Створюється нетипізований файл. Довжина блоку
  1 байт} Rewrite(matrix, 1);
  writeln(UTF8ToConsole('Введіть кількість невідомих'));
  readln(n);
  {На диск буде записано count блоків завдовжки
  по 1 байту. Оскільки цілий тип займає 4 байти,
  буде записано 4 блоки} count: = SizeOf
  (integer);
  BlockWrite(matrix, n, count);
  {Оскільки дійсний тип real займає 8
  байт, буде записано 8 блоків по 1 байту}
  count: = SizeOf (real);
  {Введення коефіцієнтів розширеної
  матриці} for i:=1 to n do
  begin
    for j:=1 to n do
    begin
      writeln(UTF8ToConsole('Введіть a'), i, j);
      readln(temp);
      BlockWrite(matrix, temp, count);
    end;
    writeln(UTF8ToConsole('Введіть b'), i);
    readln(temp);
    {Можна відразу записати функцію SizeOf як
```

```
      3-го параметра процедури запису/читання} BlockWrite(matrix,
      temp, SizeOf(real));
  end;
  writeln(UTF8ToConsole('Інформація на диск записана'));
  CloseFile(matrix);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Програма вирішення системи лінійних рівнянь алгебри методом Гауса, коефіцієнти розширеної матриці вводяться з нетипізованого файлу:

```
program Gauss_File;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  matrix: File;
  a: array of array of real; {матриця коефіцієнтів системи,
  двовимірний динамічний масив}
  vector: array of real; {перетворений
  одновимірний динамічний масив}
  b: array of real;
  x: array of real;
  temp: real;
  i, j, k, n: integer;
{Процедура залишається без змін}
procedure gauss(var vector: array of real;
                var b: array of real;
                var x: array of real;
                var n: integer);
var
  a: array of array of real; {матриця коефіцієнтів системи,
  двовимірний динамічний масив}
  i, j, k, p, r: integer;
  m, s, t: real;
begin
  SetLength(a, n, n); // Встановлення фактичного розміру
  масиву
```

```
{Перетворення одновимірного масиву на
двовимірний} k:=1;
for i:=0 n-1 do for
  j:=0 n-1 do begin
    a[i,j]:= vector[k];
    k:=k+1;
  end;
for k:=0 n-2 do begin
  for i:=k+1 n-1 do begin
    if (a[k,k]=0) then
    begin
      {перестановка рівнянь}
      p:=k; // В алгоритмі використовується буква l, але
      вона схожа на 1
      // Тому використовуємо ідентифікатор p
      для r:=і до n-1 до
      початку
        if abs(a[r,k]) > abs(a[p,k]) then p:=r;
      end;
      if p<>k then
      begin
        for j:= до n-1 до
        початку
          t: = a [k, j];
          a[k,j]:=a[p,j];
          a[p,j]:=t;
        end; t: = b
        [k];
        b[k]:=b[p];
        b[p]:=t;
      end;
    end; // кінець блоку перестановки рівнянь
    m:=a[i,k]/a[k,k];
    a[i,k]:=0;
    for j:=k+1 n-1 do
    begin
      a[i,j]:=a[i,j]-m*a[k,j];
    end;
    b[i]:= b[i]-m*b[k];
  end;
end;
```

```
  {Перевірка існування
  рішення} if a[n-1,n-1] <>
  0 then begin
    x[n-1]:=b[n-1]/a[n-1,n-
    1];for i:=n-2 downto 0 do
    begin
      s:=0;
      for j:=i+1 n-1 do
      begin
        s:=sa[i,j]*x[j];
      end;
      x[i]:=(b[i] + s)/a[i,i];
    end;
    writeln('');
    writeln(UTF8ToConsole('Рішення:'));
    writeln('');
    for i:=0 to n-1 do
      writeln('x', i+1, '=', x[i]:0:4);
  end
  else
  if b[n-1] = 0 then
    writeln(UTF8ToConsole('Система не має
    рішення.'))
  else
    writeln(UTF8ToConsole('Система
    рівнянь'+ 'має безліч рішень.'));
  writeln('');
  {звільнення пам'яті,
```

розподіленої для динамічного масиву a:=nil;

```
end;
{Початок основної
програми} begin
  AssignFile(matrix, 'Coeff.dat');
  Reset(matrix, 1);
  {Читання кількості рівнянь системи із файлу}
  BlockRead(matrix, n, SizeOf(integer));
  {Установка реальних розмірів динамічних
  масивів} SetLength(a, n, n);
  SetLength(vector, n*n);
  SetLength(b, n);
  SetLength(x, n);
  {Введення коефіцієнтів розширеної
  матриці} for i:=1 to n do
```

```
  begin
    for j:=1 to n do
    begin
      BlockRead(matrix, temp, SizeOf(real));
      a [i-1, j-1]: = temp;
    end;
    BlockRead(matrix, temp, SizeOf(real));
    b [i-1]: = temp;
  end;
  {Перетворення двовимірного масиву в
  одновимірний} k:=1;
  for i:=0 n-1 do for
    j:=0 n-1 do begin
      vector[k]:=a[i,j];
     k:=k+1;
    end;
  CloseFile(matrix);
  {Виклик процедури розв'язання системи
  лінійних рівнянь алгебри методом Гауса}
  gauss(vector, b, x, n);
  {звільнення пам'яті,
  розподіленої динамічних
  масивів} a:=nil;
  vector:=nil;
  x:=nil;
  b:=nil;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

3.6.3.5. Організація контролю введення/виводу під час роботи файлами

При роботі з файлами часто виникають непередбачені ситуації, які призводять до помилок вводу/виводу. Такі ситуації виникають, наприклад, якщо вказаний файл не існує на диску або диск не готовий до роботи або файл з якихось причин був зіпсований і прочитати дані неможливо тощо. У таких випадках, якщо не передбачити відповідних дій, програма аварійно завершується. Паскаль надає можливість ввести до програми контроль операцій введення/виводу. Це дає можливість навіть якщо

не виправити відразу виниклу помилку, хоча б локалізувати її, вивести якесь повідомлення чи попередження і продовжити роботу з програмою.

Для організації контролю за операціями введення/виведення використовується функція IOResult. Перед викликом цієї функції необхідно вимкнути автоматичний контроль операцій введення/виведення директивою компілятора

{$I-}. Цією директивою програма повідомляє компілятору і операційній системі, що контроль наступних операцій введення/виводу вона бере на себе. Тільки після цього функція IOResult стає доступною. Після завершення чергової операції введення/виведення функція повертає 0, якщо операція пройшла успішно. В іншому випадку, функція повертає код помилки. Необхідно аналізувати значення функції IORsult відразу після вводу/виводу. В іншому випадку функція скидає своє значення в 0. Після завершення небезпечної ділянки програми автоматичний контроль вводу/виводу потрібно відновити директивою компілятора {$I+}.

Розглянемо приклад. Нехай текстовий файл Data.txt. У програмі моделюється ситуація, коли користувач записує у файл послідовності цілих чисел у їхньому символьному поданні. Потім програма читає ці числа у змінну цілого типу. Вперше користувач вводить рядок '1234', вдруге користувач випадково ввів замість цифри неприпустимий для числа символ, наприклад, вводить рядок '12#4', тобто. натиснув клавішу з цифрою 3 у верхньому регістрі.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  F: TextFile;
```

```
code: слово;
number:integer;
```

```
  s:string;
begin
  AssignFile(F, 'Data.txt');
  Rewrite(F);
  s:='1234';
  Writeln(F, s);
  s:='12#4';
  Writeln(F, s);
  Reset(F);
  while not Eof(F) do
  begin
    {$I-}
    Readln(F, number);
    {$I+}
    code:= IOResult;
    if code<>0 then
    begin
      writeln(UTF8ToConsole('Помилка перетворення типів'));
      writeln(UTF8ToConsole('код помилки'), code);
      break
    end;
    writeln('number=', number);
  end;
  CloseFile(F);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Як бачимо, перший рядок файлу прочитано правильно. Виводиться значення цілого числа number=1234. Під час читання другого рядка файлу відбувається

помилка перетворення рядка символів у ціле число, про що програма і повідомляє користувача.

Для перевірки існування файлу зручніше користуватися функцією

FileExists(fname), де ім'я файлу fname – рядок символів. У цьому фрагменті програмі перевіряється існування файлу на диску. Якщо файл існує, він відкривається для читання, інакше створюється новий порожній файл з тим самим ім'ям:

```
if not FileExists('My_file.txt') then
    Rewrite(kol)
  else
    Reset(kol);
```

3.6.3.6. Створення простої бази даних із типізованими файлами.

Розглянемо приклад розробки досить великої та складної програми. При її написанні ми використовуватимемо всі отримані нами знання з мови Паскаль, викладені у попередніх розділах. Зокрема, методи роботи з типізованими файлами, в яких використовуються записи складної структури, процедури та функції, контроль операцій введення/виводу. Освоїмо техніку створення та роботи з меню. Практично створюється маленька і простенька база даних. Щоб не захаращувати текст програми численними перевірками в програму введено процедури Reset_file, Read_File і Write_File всередині яких і здійснюється контроль правильності операцій вводу/виводу. Структура запису, яка використовується у програмі має вигляд:

Прізвище	Група	Предмет	Оцінка

```
program Database_Student;
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil, SysUtils, LConvEncoding, LCLType;
type
  student = record{  тип запис}
  fio: string[24];// прізвище:
  string[32]; / / предмет gruppa: string [24]; //
  група ocenka: integer;
end;

  fstud = File of student;

var
  fam:     string[24];     //
  прізвище  sub:  string[32];
  /  /  предмет  gr:  string
  [24]; // група
  answ: TUTF8Char; // Символ прийому відповіді користувача
  f, v: fstud; {f,v – файлові змінні,
  де f – ім'я основного файлу;
  v – ім'я допоміжного файлу}
  new_student: student;
  choice,choose, ocen: integer; {змінні,
  призначені для вибору режиму}
  fname: string; {рядкова змінна,
  ім'я файлу на диску без
  розширення}
  full_fname: string; {рядкова змінна, повне
  ім'я файлу на диску з розширенням}
  code_error: integer; // код помилки
  введення/виводу

// Випереджальне оголошення функцій та
процедур procedure Reset_File (var f: fstud);
forward; procedure Read_File(var f: fstud;
                    var st: student); forward;
procedure Write_File(var f: fstud;
                     var st: student); forward;
procedure check_file; forward;

{ ================ Введення даних ================ }
procedure input_data;
begin
  with new_student do
  begin
```

```
    writeln(UTF8ToConsole('Прізвище  '));
    readln(fio);writeln(UTF8ToConsole('
    Група                              '));
    readln(gruppa);writeln(UTF8ToConsole
    ('Предмет')); readln(predmet);
    writeln(UTF8ToConsole('Оцінка   '));
    repeat
      {$I-}
      readln(ocenka);
      {$I+}
      if (IOResult <> 0) or ((ocenka > 5)
       or (ocenka < 1)) then
      begin
        writeln(UTF8ToConsole('Введіть ціле число від 1 до
      5')); end;
    until (ocenka >= 1) and (ocenka <= 5);
  end;
end;

{ ================ Створення файлу ================ }
procedure create_data;
begin
  check_file;
  Rewrite(f);
  writeln(UTF8ToConsole(' Введіть дані'));
  repeat
  input_data;
  Write(f, new_student);
  writeln(UTF8ToConsole(' Продовжити?, відповідь - д/н
  (y/n)')); readln(answ);
  {$IFDEF WINDOWS}
     answ:= CP866ToUTF8(answ);
  {$ENDIF}
  until (answ='N') or (answ='n') or (answ='н') or (answ= 'Н');
end;

procedure check_file;
var
answ: TUTF8char;
begin
  if FileExists(full_fname) then
  begin
```

```
    Assign(f, full_fname);
    Reset(f);
    writeln(UTF8ToConsole('Поточний файл буде знищено!!'));
    writeln(UTF8ToConsole('Щоб стерти існуючий'));
    writeln(UTF8ToConsole('файл, натисніть клавішу Esc,'));
    writeln(UTF8ToConsole('інакше натисніть будь-яку
    клавішу.')); repeat
      answ: = readkey;
        if answ= #27 then
        begin
          writeln(UTF8ToConsole('Ви впевнені?
          Натисніть ')); writeln(UTF8ToConsole('ще раз
          клавішу Esc')); writeln(UTF8ToConsole('Для
          скасування натисніть '));
          writeln(UTF8ToConsole('будь-яку клавішу.'));
          answ: = readkey;
          if answ = #27 then
            break;
        end;
        writeln(UTF8ToConsole('Введіть інше ім'я файлу'));
        CloseFile(f);
        readln(fname);
        Assign(f, fname + '.dat');
        break;
    until answ = # 27;
  end;
end;

{ ================ Виведення вмісту файлу ================
} procedure out_to_screen;
var j: integer;
begin
  Reset_File(f);
  ClrScr;
  GoToXY(1, 5);
  j: = 0;
  writeln(UTF8ToConsole('*прізвище*група*предмет*оцінка*'
  )); wri-
teln('=============================================');
  while not Eof(f) do
  begin
    read(f,
    new_student);j: = j +
    1;
```

```
    GoToXY(2, 6 + j);
```

```
    writeln(new_student.fio);GoT
    oXY(15, 6 + j);
    writeln(new_student.gruppa);
    GoToXY(28, 6 + j);
    writeln(new_student.predmet);GoToXY(48
    , 6 + j);
    writeln(new_student.ocenka);
  end;
  wri-
teln('=============================================');
  writeln(UTF8ToConsole('Кількість
  студентів='),j:2);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end;

{ ========= Пошук записів по заданим полям ============ }
procedure select_data;
begin
  repeat
  Reset_File(f);
  ClrScr;GoToXY(1
  0, 10);
  write (UTF8ToConsole('Вибір інформації з:'));
  GoToXY (10, 11);
  write (UTF8ToConsole('групі-          1'));
  GoToXY (10, 12);
  write (UTF8ToConsole('предмету-        2'));
  GoToXY (10, 13);
  write (UTF8ToConsole('оцінки-          3'));
  GoToXY (10, 14);
  writeln(UTF8ToConsole('вихід з режиму-      4'));
  readln(choice);
  ClrScr;
  case choice of
    1: begin
         write(UTF8ToConsole(' Група -'));
         readln(gr);
         writeln(UTF8ToConsole('Довідка по групі'),
                 UTF8ToConsole(gr):5);
       end;
    2: begin
         write(UTF8ToConsole('Предмет-'));
         readln(sub);
```

```
        writeln(UTF8ToConsole('Довідка з предмета'),
                UTF8ToConsole(sub):15);
      end;
    3: begin
        write(UTF8ToConsole(' Оцінка ='));
        readln(ocen);
        writeln(UTF8ToConsole('Інформація з оцінки'),
                ocen:1);
      end;
  else
    exit;
  end; { end of case
  } while not eof(f)
  do begin
    Read_File(f,new_student);cas
    e choice of
      1: if new_student.gruppa=gr then
           writeln(new_student.fio:15,
           ' ',new_student.predmet:15,
           ' ',new_student.ocenka:1);
      2: if new_student.predmet=sub
           then
           writeln(new_student.fio:15,
           ' ',new_student.gruppa:15,
           '', new_student.ocenka:1);
      3: if new_student.ocenka=ocen
           then
           writeln(new_student.fio:15,
           ' ',new_student.predmet:15,
           ' ',new_student.gruppa:5);
    end; {end of case}
  end; { end of while }
  GoToXY (5, 24);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  until choice = 4;
end;

{ ======== Відновлення файлу під основне ім'я f ==========
} procedure restorefile;
begin
  CloseFile(f);
  CloseFile(v);
  Erase(f);
```

```
  Rewrite(f);
```

```
  Reset(v);
  while not Eof(v) do
  begin
    Read_File(v, new_student);
    Write_File(f, new_student);
  end;
  CloseFile(f); CloseFile(v); Erase(v);
    {віддалений допоміжний файл v під зовнішнім
ім'ям s.dat} end;

{================ Додавання записів у файл
================} procedure add_data;
begin
  Assign(v, 's.dat');
  Rewrite(v);
  {"s.dat" - ім'я допоміжного файлу}
  Reset_File(f);

  { копіювання вмісту файлу f у файл v }
  while not Eof(f) do
  begin
    Read_File(f, new_student);
    Write_File(v, new_student);
  end;
  writeln(UTF8ToConsole(' Вводьте інформацію'));
  { записи додаються в кінці
  файлу } repeat
    input_data;
    Write_File(v, new_student);
    writeln(UTF8ToConsole(' Продовжити?, відповідь - д/н
    (y/n)')); readln(answ);
    {$IFDEF WINDOWS}
      answ:= CP866ToUTF8(answ);
    {$ENDIF}
  until (answ='N') or (answ='n') or (answ='н') or (answ= 'Н');
  restorefile;
end;

{================ Видалення записів з файлу
================} procedure delete_data;
begin
  Assign(v, 's.dat'); Rewrite(v); Reset(f);
  ClrScr;
```

```
  GoToXY(10, 10);
  writeln(UTF8ToConsole('Видалення інформації
  по:')); GoToXY (10, 11);
  writeln(UTF8ToConsole('групі-         1'));
  GoToXY (10, 12);
  writeln(UTF8ToConsole('прізвища-      2'));
  GoToXY (10, 13);
  writeln(UTF8ToConsole('предмету-      3'));
  GoToXY (10, 14);
  writeln(UTF8ToConsole('оцінки-        4'));
  GoToXY (10, 15);
  writeln(UTF8ToConsole('вихід з режиму-    5'));
  GoToXY (10, 16);
  write(UTF8ToConsole('вибір режиму ='));
  readln(choice);
  case choice of
    1: begin
         write(UTF8ToConsole(' Група - '));
         readln(gr);
       end;
    2: begin
         write(UTF8ToConsole('Прізвище -'));
         readln(fam);
       end;
    3: begin
         write(UTF8ToConsole('Предмет -'));
         readln(sub);
       end;
    4: begin
         write(UTF8ToConsole(' Оцінка - '));
         readln(ocen);
       end;
    5: exit; {вихід в основну програму}
  end; {end of case}

{ ========= пошук запису для видалення
  =========== } while not Eof(f) do
  begin
    Read_File(f, new_student);
    case choice of
      1: if new_student.gruppa<>gr then
           Write_File(v,new_student);
```

```
      2: if new_student.fio<>fam then
           Write_File(v,new_student);
      3: if new_student.predmet<>sub then
           Write_File(v,new_student);
      4: if new_student.ocenka<>ocen then
           Write_File(v,new_student);
         else
         begin
           writeln(UTF8ToConsole('Помилка при введенні'));
           writeln(UTF8ToConsole('Натисніть будь-яку
           клавішу')); readkey;
         end;
    end; {end of
  case} end; { end
  of while }
  restorefile;
end;

{======== процедура відкриття файлу з контролем операції
======== } procedure Reset_File( var f:fstud);
begin
  {$I-}
  Reset(f);
  {$I+}
  code_error:= IOResult;
  if code_error <> 0 then
  begin
    writeln(UTF8ToConsole('Файл не існує, кодпомилки   '),
code_error);
    writeln(UTF8ToConsole('Натисніть будь-яку
    клавішу')); readkey;
    Halt;
  end;
end;

{========== процедура читання з контролем операції
===========} procedure Read_File(var f: fstud; var st:
student); begin
  {$I-}
  Read(f, st);
  {$I+}
  code_error:= IOResult;
  if code_error <> 0 then
  begin
```

```
    writeln(UTF8ToConsole('Помилка читання, код помилки '),
code_error);
    writeln(UTF8ToConsole('Натисніть будь-яку
    клавішу')); readkey;
    Halt;
  end;
end;

{========== процедура запису з контролем операції
============} procedure Write_File(var f: fstud; var
st: student); begin
  {$I-}
  Write(f, st);
  {$I+}
  code_error:= IOResult;
  if code_error <> 0 then
  begin
    writeln(UTF8ToConsole('Помилка запису у файл, код
помилки'), code_error);
    writeln(UTF8ToConsole('Натисніть будь-яку
    клавішу')); readkey;
    Halt;
  end;
end;
{ ============== коригування записів у файлі ==============
} procedure find_data;
var r: student;
begin
  Reset_File(f); Assign(v,'s.dat'); Rewrite(v);
  ClrScr;
  GoToXY (10, 9);
  writeln(UTF8ToConsole('Вкажіть ключ (поле) для пошуку'));
  GoToXY(10, 10);
  writeln(UTF8ToConsole('коригованого запису - по:'));
  GoToXY (10, 11);
  writeln(UTF8ToConsole('групі-        1'));
  GoToXY (10, 12);
  writeln(UTF8ToConsole('прізвища-      2'));
  GoToXY (10, 13);
  writeln(UTF8ToConsole('предмету-      3'));
  GoToXY (10, 14);
  writeln(UTF8ToConsole('оцінки-        4'));
  GoToXY (10, 15);
```

```
writeln(UTF8ToConsole('вихід з режиму-      5'));
GoToXY (10, 16);
write(UTF8ToConsole('вибір режиму ='));
readln(choice); ClrScr;
GoToXY (10, 9);
writeln(UTF8ToConsole('Заміна інформації
                '));case choice of{
                пошук запису }
  1: begin
       GoToXY(10, 10);
       write(UTF8ToConsole('група='));
       readln(gr);
       input_data;
     end;
  2: begin
       GoToXY(10, 10);
       write(UTF8ToConsole('прізвище='));
       readln(fam);
       input_data;
     end;
  3: begin
       GoToXY(10, 10);
       write(UTF8ToConsole('предмет='));
       readln(sub);
       input_data;
     end;
  4: begin
       GoToXY(10, 10);
       write(UTF8ToConsole('оцінка='));
       readln(ocen);
       input_data;
     end;
  5: exit; {вихід в основну програму}
end; {end of case}
while not Eof(f) do
begin
  Read_File(f, r);
  case choice of
    1: begin
         if gr=r.gruppa then
           Write_File(v,new_student)
         else
           Write_File(v,r)
       end;
```

```
      2: begin
           if fam=r.fio then
             Write_File(v,new_student)
           else
             Write_File(v,r)
         end;
      3: begin
           if sub=r.predmet then
             Write_File(v,new_student)
           else
             Write_File(v,r)
         end;
      4: begin
           if ocen=r.ocenka then
             Write_File(v,new_student)
           else
             Write_File(v,r)
         end;
    end; {end of
  case} end; { end
  of while }
  restorefile;
end;
   { ============== основна програма ==================}
begin
  writeln(UTF8ToConsole('Введіть ім'я файлу:'));
  readln(fname);
  {$IFDEF WINDOWS}
    fname:=CP866ToUTF8(fname);
    fname:=UTF8ToAnsi(fname);
  {$ENDIF}
  full_fname:=fname +
  '.dat';
  Assign(f,full_fname);
  repeat
    ClrScr;
{ Формування меню роботи з основним файлом f}
    GoToXY (10, 7);
    writeln(UTF8ToConsole('Виберіть потрібний режим
    роботи:')); GoToXY (10, 8);
    writeln(UTF8ToConsole('Створення файлу          1'));
   GoToXY (10, 9);
    writeln(UTF8ToConsole('Виведення вмісту файлу
                                                    2'));Go
```

```
    ToXY(10, 10);
    writeln(UTF8ToConsole('Пошук по заданим полям  3'));
```

```
    GoToXY (10, 11);
    writeln(UTF8ToConsole('Додавання записів у файл
                                                    4'));Go
    ToXY (10, 12);
    writeln(UTF8ToConsole('Видалення записів із файлу
                                                    5'));Go
    ToXY (10, 13);
    writeln(UTF8ToConsole('Коригування записів у файлі
                                                     6'));Go
    ToXY (10, 14);
    writeln(UTF8ToConsole('Вихід із програми          7'));
    readln(choose);
    case choose of
  {choose – значення для вибору режиму роботи з
      файлом f} 1: create_data;
      2: out_to_screen;
      3: select_data;
      4: add_data;
      5: delete_data;
      6: find_data;
    end; {end of
    case}
  until choose = 7;
end.
```

У цій програмі новим для нас є лише випереджаюче оголошення функцій та процедур. У Паскалі суворо дотримується правило – кожен об'єкт перед використанням має бути описаний. Що стосується функцій та процедур, то тут зроблено деякі "поблажки". Нехай є дві процедури, одна з яких викликає іншу.

```
procedure A (param: integer;)
begin
  . . . . .
  B(i);
   . . . . .
end;
procedure B(param: integer);
begin
```

. . . . .

```
end;
```

При такому розташуванні та описі процедур компілятор видасть помилку "Identifier not found "B"". Можна звичайно переставити місцями ці процедури, але можна зробити так званий випереджальний опис процедури таким чином:

```
procedure B(param: integer); forward;
procedure A (param: integer;)
begin
  . . . . .
  B(i);
   . . . . .
end;
procedure B(param: integer);
begin
  . . . . .
end;
```

Як бачимо, випереджальний опис полягає в тому, що оголошується лише заголовок процедури, а тіло процедури замінюється директивою forward. Тепер у процедурі A можна використовувати звернення до процедури, оскільки вона вже описана, точніше, відомі її формальні параметри, і компілятор може правильним чином організувати її виклик.

Розглянемо іншу ситуацію. Припустимо, що процедури A та B викликають одна одну:

```
procedure A (param: integer;)
begin
  . . . . .
```

```
  B(i);
   . . . . .
end;
procedure B(param: integer);
begin
  . . . . .
  A(i);
   . . . . .
end;
```

Тепер не допоможе і перестановка місцями процедур, оскільки вони "зацик- лені" один на одного. Тільки використання випереджуючого оголошення процедури дозволяє вирішити цю проблему.

```
procedure B(param: integer); forward;
procedure A (param: integer;)
begin
  . . . . .
  B(i);
   . . . . .
end;
procedure B(param: integer);
begin
  . . . . .
  A(i);
   . . . . .
end;
```

Розділ 4 Типові алгоритми обробки інформації

До типових алгоритмів я відношу алгоритми сортування, пошуку та алгоритми роботи з динамічними структурами. Можна, звичайно, розглянути і безліч інших алгоритмів, віднісши їх до типових. Але ми цьому розділі розглянемо саме ці алгоритми.

4.1. Алгоритми сортування

Сортування - процес упорядкування елементів будь-якого об'єкта за зростанням або зменшенням значень обраної ознаки. Після закінчення сортування елементи розташовуються або за зростанням або за спаданням. Якщо серед елементів є однакові, то кажуть сортування за незменшенням або невростанням. На сортування інформації витрачається 25-50% машинного часу під час роботи зі складними програмними комплексами (різні інформаційно-пошукові системи на базі СУБД, компілятори та інші програмні засоби).

Сортувати можна дані, котрим визначено операції порівняння. Найчастіше сортуванню підлягають числа. Однак сортувати можна не лише числа як такі. Ми з вами вже знаємо, що символи відображаються в пам'яті у вигляді деяких кодів. Таким чином, цілком можна сортувати символи, наприклад, прізвища можна відсортувати за абеткою. Сортувати можна як об'єкти, що містять одиночні елементи, а й об'єкти досить складної структури, наприклад записи, містять кілька елементів. Файли зазвичай містять записи складної структури. Таким

В основному, сортуванню підлягають файли. У програмі попереднього розділу структура запису файлу складалася з кількох полів, таких як "Прізвище", "Група", "Предмет", "Оцінка". Цілком можна здійснити сортування цього файлу, наприклад за групою, а всередині групи відсортувати прізвища студентів за алфавітом. Поле, яким здійснюється сортування називається ключем. У нашому випадку ключ це спочатку поле "Група", потім поле "Прізвище" всередині відсортованої частини файлу, що відповідає тій чи іншій групі.

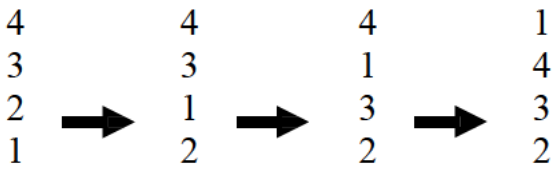
Ефективність методів сортування оцінюється за кількістю порівнянь елементів між собою та кількістю перестановок елементів для впорядкування масиву або файлу. Існує безліч алгоритмів сортування. Якщо файл, що сортується, повністю поміщається в оперативній пам'яті, то сортування називається внутрішнім. Сортування файлів на диску називається зовнішнім сортуванням. Крім того, всі алгоритми сортування можна умовно розділити на "прості", але "повільні" та "складні", але "швидкі". Взяття в лапки тут невипадково. У деяких випадках "складні", але "швидкі" алгоритми виявляються повільнішими за "прості". Найчастіше це відбувається для невеликих файлів. Для переважної кількості випадків " прості " алгоритми цілком підходять - за швидкістю роботи немає сенсу " перекручуватися " , намагаючись реалізувати складні і хитромудрі алгоритми. Необхідно пам'ятати, що чим простіше алгоритм (і це стосується не лише алгоритмів сортування), тим надійніше працюватиме ваша програма. І тільки для великих файлів, коли саме на сортуванні ваша програма "застряє", варто пошукати більш складні і швидкі алгоритми.

Розглянемо кілька алгоритмів сортування. У прикладах ми розглядатимемо сортування масивів цілих чисел.

4.1.1 Обмінне сортування (метод "бульбашка")

Свою назву алгоритм отримав завдяки тому, що в процесі сортування менші (великі) числа спливають вгору як бульбашки повітря у воді. Якщо йде сортування за зростанням, "спливають" менші числа, якщо за спаданням, то більші. Розглянемо приклад.

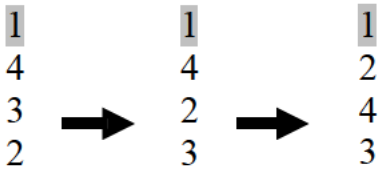
Нехай є масив, що складається із чотирьох чисел: 4, 3, 2, 1. Необхідно розмістити елементи масиву за зростанням. Порівнюються, починаючи з кінця масиву кількох сусідніх чисел. Якщо вони розташовані в неправильному порядку – міняємо їх місцями. Через війну першого проходу найменший елемент (число 1) виявляється " зверху " , дома нульового елемента, тобто. "легкий" елемент "сплив на поверхню", рис. 4.1.



Мал. 4.1 Стан масиву під час першого проходу.

Як видно з малюнка, на першому кроці поточного проходу змінюються місцями числа "1" та "2". На другому кроці змінюються "1" та "3" і на четвертому кроці змінюються місцями "1" та "4".

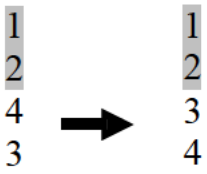
Далі здійснюється другий прохід, у якому проглядаються елементи масиву, крім першого, що вже "встав" на своє місце. Стан масиву під час другого проходу, рис. 4.2:



Мал. 4.2 Стан масиву під час другого проходу.

На малюнку елемент, який не бере участі у перегляді, виділено.

На наступному (останньому для даного масиву) проході поміняються місцями числа "4" та "3". У результаті отримуємо відсортований масив:

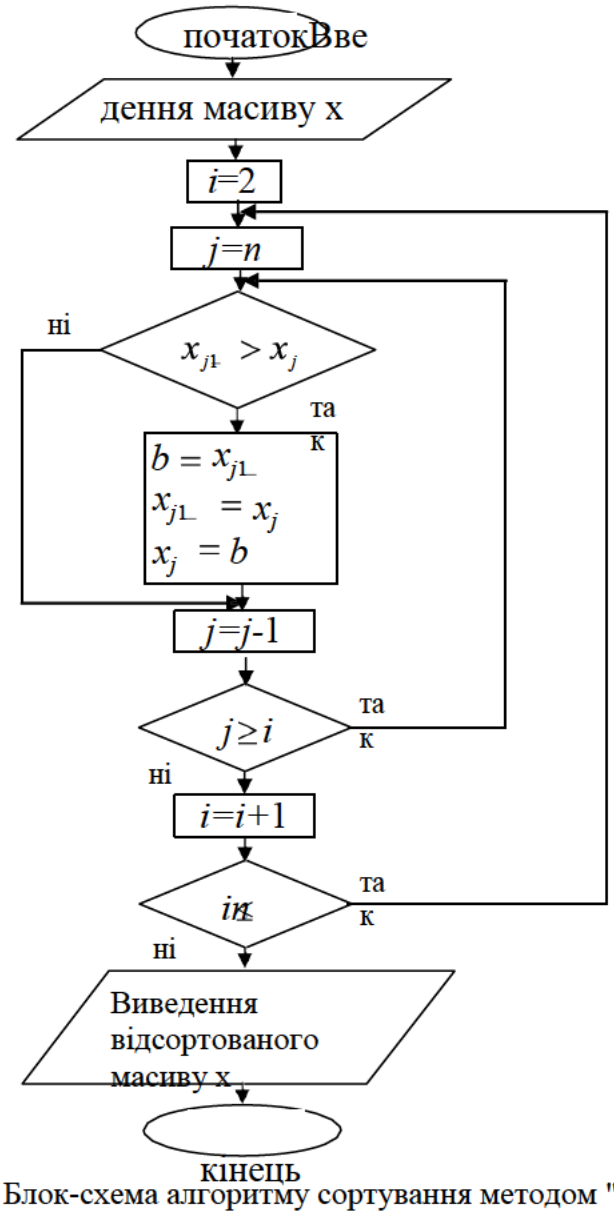


Мал. 4.3 Після третього проходу одержуємо відсортований масив.

Таким чином, проходи робляться по нижній частині масиву, що зменшується, до тих пір, поки в ній не залишиться тільки один елемент. На цьому сортування закінчується, оскільки послідовність упорядкована за зростанням. Блок-схему алгоритму сортування методом "бульбашка" наведено на малюнку 4.4.

Складемо програму з використанням динамічних масивів. Під час аналізу програми не забудьте, що індексація елементів у таких масивах починається з 0!

```
program bubble_sort;
uses
  CRT, FileUtil;
var
  i, n: integer;
     vector: array of integer;
{ ============ Сортування методом "бульбашка"
======== } procedure bubble(var vector: array of
integer); var
  temp: integer;
  i, j, count: integer;
begin
```



Мал. 4.4 Блок-схема алгоритму сортування методом "бульбашка".

```
count: = high (vector);
for i:= 1 to count   do
for j:= count downto i do
if vector[j - 1] > vector[j] then
begin
  temp: = vector [j - 1];
  vector[j - 1]: = vector[j];
  vector[j]:= temp;
end
```

```
end;
begin
  writeln(UTF8ToConsole('Введіть кількість елементів
  масиву')); readln(n);
  SetLength(vector, n);
  writeln(UTF8ToConsole('Введіть'), n);
  writeln(UTF8ToConsole('значень елементів
  масиву')); for i: = 0 to n - 1 do read (vector
  [i]); bubble(vector);
  writeln;
  writeln(UTF8ToConsole('Відсортований
  масив')); for i:= 0 to n - 1 do
  write(vector[i], ''); writeln;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Алгоритм бульбашкового сортування є одним із найповільніших. У реалізації алгоритму є два цикли. При першому виконанні внутрішнього циклу буде виконано n-1 порівнянь, при другому n-2 порівнянь тощо. Усього буде виконано n-1 таких циклів. Таким чином, все буде виконано

$$(n-1) + (n-2) + \ldots + 1$$

порівнянь. Вираз можна спростити:

$n(n-1)/2$ або $(n2- n)/2$

Таким чином, бульбашкова сортування вимагає O(n2) порівнянь. Кількість перестановок для найгіршої нагоди, коли елементи масиву відсортовані у зворотному порядку дорівнює кількості порівнянь, тобто. O(n2).

Є можливість невеликого покращення алгоритму. Якщо у внутрішньому циклі немає жодної перестановки, це означає, що масив вже відсортований і подальше виконання алгоритму можна припинити.

У кращому випадку, коли вхідний масив вже відсортований, алгоритму потрібно всього (n-1) порівнянь і жодної перестановки. На жаль, у найгіршому випадку, коли всі елементи масиву розташовані у зворотному порядку, виграшу у часі виконання алгоритму не відбувається. Наведемо таки реалізацію цієї модифікації методу:

```
program modify_bubble_sort;
uses
  CRT, FileUtil;
var
  i, n: integer;
  vector: array of integer;
{ ============ Сортування методом "бульбашка"
======== } procedure bubble(var vector: array of
integer); var
  temp: integer;
  i, j, count: integer;
  perestanovka: boolean = false;
begin
  count: = high (vector);
  for i:= 1 to count   do
  begin
    for j:= count downto i do
    if vector[j - 1] > vector[j] then
    begin
      perestanovka:= true;
      temp: = vector [j - 1];
      vector[j - 1]: =
      vector[j]; vector[j]:=
      temp;
```

```
    end;
    if not perestanovka then break;
  end;
end;
begin
  writeln(UTF8ToConsole('Введіть кількість елементів
  масиву')); readln(n);
  SetLength(vector, n);
  writeln(UTF8ToConsole('Введіть'), n);
  writeln(UTF8ToConsole('значень елементів
  масиву')); for i: = 0 to n - 1 do read (vector
  [i]); bubble(vector);
  writeln;
  writeln(UTF8ToConsole('Відсортований
  масив')); for i:= 0 to n - 1 do
  write(vector[i], ''); writeln;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Існують ще кілька модифікацій та покращень цього алгоритму. За бажанням ви можете ознайомитися з ними у спеціальній літературі.

4.1.2 Сортування вибором

Алгоритм сортування вибором працює наступним чином: знаходимо найменший елемент у масиві та обмінюємо його з елементом, що знаходиться на першому місці. Потім шукаємо мінімальний елемент без урахування першого елемента та знайдений мінімальний елемент обмінюємо з другим елементом тощо. На і-му кроці вибираємо найменший елемент a[i], ..., a[n] і змінюємо його

місцями з a[i]. Після кроку i, послідовність a[0],..., a[i] буде вже упорядкованою. Тепер шукаємо мінімальний елемент серед a[i+1], ..., a[n] та змінюємо його з a[i+1]. Таким чином, на (n-1)-му кроці вся послідовність, крім a[n] виявляється відсортованою, а a[n] виявляється саме там, де він і повинен стояти, тому що всі менші елементи вже "пішли" вліво. Цей метод називається сортуванням вибором, оскільки він на кожному наступному кроці алгоритму знаходить найменший з елементів масиву, що залишилися, і переставляє його відразу в потрібне місце в масиві.

Кількість порівнянь для першого проходу дорівнює n, другого n-1 і т.д. Загальна кількість порівнянь дорівнює n(n+1)/2 – 1, тобто. Цей алгоритм вимагає O(n2) порівнянь. Кількість перестановок у цьому алгоритмі менша, тому що в кожному проході він переставляє елементи лише один раз і число перестановок становить O(n). Таким чином, алгоритм вибору дещо ефективніший за бульбашковий метод. До того ж, алгоритм вибору є стійким. Що це означає? Якщо серед елементів сортованого об'єкта є однакові, алгоритм не порушує їх взаємного розташування у вихідному об'єкті. Нехай є два записи з однаковими ключами

1 A

2 B

1 C

Мал. 4.5. Записи з однаковими ключами

Стійкий алгоритм відсортує записи у такому вигляді:

1 A

1 З

2 B

Мал. 4.6. Результат сортування стійким алгоритмом

а нестійкий може впорядкувати записи у такому вигляді:

1 З

1 А

2 В

Мал. 4.7. Результат сортування нестійким алгоритмом

тобто. порядок запису з однаковими ключами в порівнянні з вихідним файлом може порушитися.

Блок-схему алгоритму скласти нескладно. Пропоную це зробити вам самим.

```
program select_sort;
uses
  CRT, FileUtil;
var
  vector: array of integer;
  i, n: integer;
  { ============ Сортування вибором ======== }
procedure select (var vector: array of integer);
var
  i, j, count, min, t: integer;
begin
  count: = high (vector);
  for i:= 0 to count - 1 do
  begin
   min: = i;
   for j: = i + 1 to count do
   if vector[j] < vector[min] then min:=j;
   t: = vector [min];
```

```
   vector [min]: = vector
   [i]; vector[i]:= t;
  end;
end;
{ ============================================ }

begin
writeln(UTF8ToConsole('Введіть кількість елементів
  масиву')); readln(n);
  SetLength(vector, n);
  writeln(UTF8ToConsole('Введіть'), n);
  writeln(UTF8ToConsole('значень
  масиву')); for i: = 0 to n - 1 do read
  (vector [i]); select(vector);
  writeln;
  writeln(UTF8ToConsole('Відсортований
  масив')); for i:= 0 to n - 1 do
  write(vector[i], ''); writeln;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Розберемо нагоду сортування файлів, використовуючи алгоритм вибору. Для цього скористаємося файлом менеджерів, створений у програмі розділу 3.6.3.3. Для того, щоб вам простіше було порівнювати результати, програма створює новий відсортований файл, а старий не відсортований залишає без змін, хоча в реальних програмах, як правило, відсортований файл заміщає вихідний. Для невеликих файлів ефективнішим є внутрішнє сортування, тобто. весь файл спочатку зчитується в деякий мас-

Сив сортується і потім записується на диск на місце вихідного не відсортованого файлу. Лістинг програми сортування типізованого файлу виглядає так:

```
program select_sort_file;
uses
  CRT, FileUtil, SysUtils, OutScr;
type
  manager = record
  name: string
  [18]; comp:
  integer;
 end;
var
  company: manager;
    {Файлові змінні}
  f_not_sorted; f_sorted: File of manager;
  vector: array of manager;
  i, n: integer;
  name_file: string;

{ Сортування вибором, файлу на прізвища менеджерів
} procedure select (var vector: array of manager);
var
  i, j, count, min: integer;
  t: manager;
begin
  count: = high (vector);
  for i:= 0 to count - 1 do
  begin
```

```
   min: = i;
   for j: = i + 1 to count do
   if vector[j].name < vector[min].name then min:= j;
   t: = vector [min];
   vector [min]: = vector
   [i]; vector[i]:= t;
  end;
end;
{ ================================================ }

begin
  {За потреби вкажіть повний шлях до
   файлу або скопіюйте файл у папку з цим
   проектом}
  if not FileExists('File_not_sorted.dat') then
  begin
    writeln(UTF8ToConsole('Файли не існують'));
    writeln(UTF8ToConsole('Спочатку створіть їх'));
    writeln(UTF8ToConsole('Натисніть будь-яку
    клавішу')); readkey;
    exit;
  end;
  AssignFile(f_not_sorted, 'File_not_sorted.dat');
  Reset(f_not_sorted);

  // Визначення кількості записів у
  файлі n: = System.FileSize
  (f_not_sorted); SetLength(vector,
  n);
  {Підготовка до внутрішнього
```

```
  сортування, зчитування записів
  файлу до масиву,
```

у процедуру передається масив, а не файл. i:=
0;

```
while not Eof(f_not_sorted) do
begin
   Read(f_not_sorted, company);

   // масив, який сортуватиметься
   vector[i]: = company;
   i:= i + 1;
end;
select(vector); // Виклик процедури сортування методом
вибору
CloseFile(f_not_sorted);AssignFile(f_sor
ted, 'File_sorted.dat');
Rewrite(f_sorted);
for i:= 0 to n - 1 do
Write(f_sorted, vector[i]);
CloseFile(f_sorted);
name_file:= 'File_sorted.dat';
{Виклик процедури для виведення на екран
зведеної відомості. Процедура знаходиться у
модулі OutScr} output_to_screen(name_file);
write(UTF8ToConsole('Натисніть будь-яку
клавішу')); readkey;
end.
```

Відсортуємо файл менеджерів за кількістю проданих комп'ютерів. Для цього достатньо у процедурі сортування select змінити оператор

```
if vector[j].name[1] < vector[min].name[1] then min:= j;
```

на

```
if vector[j].comp < vector[min].comp then min:= j;
```

4.1.3 Сортування вставками

Алгоритм цього методу наступний: беремо перші два елементи і розташовуємо їх у правильному порядку. Потім беремо наступний елемент і вставляємо його у потрібне місце серед тих, що ми вже обробили. Елемент, що розглядається, вставляється в позицію за допомогою переміщення більшого елемента на одну позицію вправо і потім розміщенням меншого елемента в позицію, що звільнилася. На кожному кроці і елемент a[i] поміщається у відповідну позицію серед елементів a[1], ..., a[i-1]. Тепер елементи a[1], ..., a[i] є впорядкованими, але не "зовсім", оскільки серед елементів, що залишилися, a[i+1], ..., a[n] на якомусь k-му кроці може знайтися елемент менший, ніж a[1], ..., a[i], a[i+1], …я[c-1] і він буде вставлений в. Процес завершиться, коли останній n-й елемент буде поміщений у відповідне для нього місце. Проілюструємо сказане:

4 3 2 1

Мал. 4.8 Вихідний масив

3 4 2 1

Мал. 4.9 Крок перший, змінюються перші два елементи

3 2 4 1

2 3 4 1

Мал. 4.10 Крок другий, елемент "2" зайняв своє місце, при цьому "3" та "4" зрушили праворуч.

2 3 1 4

2 1 3 4

1 2 3 4

Мал. 4.11 Крок третій, елемент "1" зайняв своє місце, при цьому "2", "3" та "4" зрушили праворуч.

Алгоритм міститиме два цикли. У зовнішньому циклі покажчик i рухатиметься праворуч (збільшуватиметься) починаючи з 2 до n, де n кількість елементів. У внутрішньому циклі покажчик j зрушується вліво, починаючи з i-1. На кожному кроці j-му кроці відбувається зсув елементів праворуч доти, доки a[i] не виявиться меншим за a[j]. Тоді a[i] залишиться на цьому місці і внутрішній цикл завершиться. Внутрішній цикл також може завершитися, коли буде досягнуто початку списку. Розглянемо програму сортування рядка символів за алфавітом методом вставок:

```
program insertion_sort_string;
uses
  CRT, FileUtil;
var
  str: string;

{Процедура сортування методом
вставок} procedure insert(var str:
string); var
  i, j: integer;
  v: char;
begin
  for i:= 2 to Length(str) do
```

```
  begin
    v:= str[i]; j:= i - 1;
    {Умова виходу з циклу: знайдено
    відповідне місце для поточного елемента
    або досягнуто початку рядка} while
    (str[j] > v) and (j > 0) do
    begin
      str[j+1]:= str[j];
      dec(j);
    end;str[j+1]:
    = v;
  end;
end;

begin
  str:= 'hgfedcba';
  writeln(UTF8ToConsole('Початковий
  рядок:'),
          UTF8ToConsole(str));
  insert(str);
  writeln(UTF8ToConsole('Відсортований рядок: '),
          UTF8ToConsole(str));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

У програмі ми розглядали рядок як масив символів. Для рядка, що складається із символів кирилиці, алгоритм необхідно дещо змінити. Зовнішній цикл буде починатися з 3, так як у кодуванні UTF-8 перший байт другого символу рядка матиме номер 3.

```
program insertion_sort_string;
uses
  CRT, FileUtil;
var
  str: string;

{Процедура сортування методом вставок кирилиці}
procedure insert_kyr(var str: string); var
  i, j: integer;
  v, v1: string[2];
begin
  for i:= 3 to Length(str) do
  begin
    v: = Copy (str, i, 2);
    j:= i - 2;
  {Умова виходу з циклу: знайдено відповідне місце для
  поточного елемента або досягнуто початку рядка}
    while (Copy(str, j, 2) > v) and (j > 0) do
    begin
      v1: = Copy (str, j, 2);
      str[j+2]:= v1[1];
      str[j+3]:= v1[2];
      dec(j, 2);
    end;
    str[j+2]:= v[1];
    str[j+3]:= v[2];
  end;
end;
```

```
begin
  str:= 'зжедгвба';
  writeln(UTF8ToConsole('Початковий
  рядок:'),
          UTF8ToConsole(str));
  insert_kyr(str);writeln(UTF8ToConsole('Відсо
  ртований рядок: '),
          UTF8ToConsole(str));writeln(UTF8ToConso
  le('Натисніть будь-яку клавішу')); readkey;
end.
```

Що буде, якщо рядок "змішаний"? Тобто. містить і символи кирилиці та символи латиниці. Найпростіше вчинити так. Продивитися весь рядок і сформувати два нові рядки. Одна міститиме лише кирилицю, а інша тільки латиницю. Потім "годувати" процедуру insert_kyr рядок з кирилицею, процедурі insert рядок з латиницею. І на завершення об'єднати відсортовані рядки функцією Concat.

```
program insertion_sort_string;
uses
  CRT, FileUtil;
var
  str, str_lat, str_kyr: string;
  i: integer;

{Процедура сортування методом вставок латиниці}
procedure insert(var str: string); var
  i, j: integer;
```

```
  v: char;
begin
  for i:= 2 to Length(str) do
  begin
    v:= str[i]; j:= i - 1;
  {Умова виходу з циклу: знайдено відповідне місце для
  поточного елемента або досягнуто початку рядка}
    while (str[j] > v) and (j > 0) do
    begin
      str[j+1]:= str[j];
      dec(j);
    end;str[j+1]:
    = v;
  end;
end;
```

{Процедура сортування методом вставок кирилиці}

```
procedure insert_kyr(var str: string); var
  i, j: integer;
  v, v1: string[2];
begin
  for i:= 3 to Length(str) do
  begin
    v: = Copy (str, i, 2);
    j:= i - 2;
  {Умова виходу з циклу: знайдено відповідне місце
  для поточного елемента або досягнуто початку рядка}
    while (Copy(str, j, 2) > v) and (j > 0) do
```

```
    begin
      v1: = Copy (str, j, 2);
      str[j+2]:= v1[1];
      str[j+3]:= v1[2];
      dec(j, 2);
    end;
    str[j+2]:= v[1];
    str[j+3]:= v[2];
  end;
end;

begin
  str:=
  'зжедгвбahgfedcbaiклмнijklmn';
  i:= 1;
  str_lat:='';
  str_kyr:='';
  while i <= Length(str) do
  begin
    if ord(str[i]) < 128
    then
    begin
      str_lat:= str_lat + str[i];
      inc(i);
    end
    else
    begin
      str_kyr: = str_kyr + Copy (str, i,
      2); inc(i, 2);
    end;
```

```
  end;
  writeln(UTF8ToConsole('Початковий рядок: '),
          UTF8ToConsole(str));
  insert(str_lat); // Сортування латиниці
  insert_kyr(str_kyr); // сортування кирилиці
  str: = Concat (str_lat, str_kyr);
  writeln(UTF8ToConsole('Відсортований рядок:
  '),
          UTF8ToConsole(str));writeln(UTF8ToConso
  le('Натисніть будь-яку клавішу')); readkey;
end.
```

Напишемо програму сортування масиву рядків методом вставок. У програмі рядки сортуються за першим символом рядка. Наприклад, якщо заданий масив із 4 рядків:

Яковлєв
Петров
Сидоров
Алексєєв

то програма видасть масив у такому вигляді:

Алексєєв
Петров
Сидоров
Яковлєв

```
program insertion_sort_array;
uses
  CRT, FileUtil;
var
  str: array of string;
  i, n: integer;

{Процедура сортування методом вставок}
procedure insert(var str: array of string);
var
  i, j: integer;
  stroka: string;
begin
  for i:= 1 to High(str) do
  begin
    stroka: = str [i]; j:= i - 1;
    while (str[j][1] > stroka[1]) and (j >= 0) do
    begin
      str[j+1]:= str[j];
      dec(j);
    end;
    str[j+1]:= stroka;
  end;
end;

begin
  writeln(UTF8ToConsole('Введіть кількість рядків у
  масиві')); readln(n);
  SetLength(str, n);
```

```
  writeln(UTF8ToConsole('Введіть'), n);
  writeln(UTF8ToConsole('рядок(і)
  символів')); for i:= 0 to n - 1 do
  readln(str[i]);writeln(UTF8ToConsole('Початко
  вий масив рядків')); for i:= 0 to n - 1 do
  writeln(str[i]);
  insert(str);
  writeln(UTF8ToConsole('Відсортований масив рядків:
  ')); for i:= 0 to n - 1 do
  writeln(str[i]);writeln(UTF8ToConsole('Натисн
  іть будь-яку клавішу')); readkey;
end.
```

Відсортуємо масив менеджерів (розділ 3.6.3.3. та 4.1.2) методом вставок:

```
program insertion_sort_file;
uses
  CRT, SysUtils, OutScr, FileUtil;
type
  manager = record
  name: string
  [18]; comp:
  integer;
 end;
var
  company: manager;
  f_not_sorted; f_sorted: File of manager;
  vector: array of manager;
```

```
  i, n: integer;
  name_file: string;

{Процедура сортування методом вставок файлу на прізвища
менеджерів} procedure insert(var vector: array of manager);
var
  i, j, count: integer;
  t: manager;
begin
  count:     =     high
  (vector);  for  i:=  0
  to count do begin
    t: = vector [i];
    j:= i - 1;
    while (vector[j].name > t.name) and (j >= 0) do
    begin
      vector[j + 1]: = vector[j];
      dec(j);
    end;
    vector [j + 1]: =
  t; end;
end;
{ ================================================ }
begin
  {За потреби вкажіть повний шлях до
   файлу або скопіюйте файл у папку з цим
   проектом}
  if not FileExists('File_not_sorted.dat') then
  begin
    writeln(UTF8ToConsole('Файли не існують'));
```

```
  writeln(UTF8ToConsole('Спочатку створіть їх'));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  exit;
end;
AssignFile(f_not_sorted, 'File_not_sorted.dat');
Reset(f_not_sorted);
// Визначення кількості записів у
файлі n: = System.FileSize
(f_not_sorted); SetLength(vector,
n);
{Підготовка до внутрішнього
 сортування, зчитування записів
 файлу до масиву,
```

у процедуру передається масив, а не файл. i:=

```
0;
while not Eof(f_not_sorted) do
begin
   Read(f_not_sorted, company); vector[i]:
   = company; // Сортований масив i:= i +
   1;
end;
insert(vector); // Виклик процедури сортування методом
вставок
CloseFile(f_not_sorted);AssignFile(f_sor
ted, 'File_sorted.dat');
Rewrite(f_sorted);
for i:= 0 to n - 1 do
Write(f_sorted, vector[i]);
CloseFile(f_sorted);
```

```
name_file:= 'File_sorted.dat';
{Виклик процедури для виведення на екран зведеної
відомості.
```

```
    Процедура знаходиться у модулі OutScr}
  output_to_screen(name_file); write(UTF8ToConsole('Натисніть
  будь-яку клавішу')); readkey;
end.
```

У внутрішньому циклі способу вставок здійснюється дві перевірки. Одна для знаходження відповідного місця поточному елементу, а інша (j>0, якщо нумерація індексів починається з 1 і j>=0, якщо нумерація індексів починається з 0) для запобігання виходу за межі лівого краю масиву (рядок символів можна розглядати як масив, елементами якого є окремі символи).

Проте ця додаткова перевірка, як свідчать численні експерименти, уповільнюють роботу алгоритму майже 7%. Виходом із цієї ситуації є поміщення у перший елемент масиву так званого "сторожового" елемента, який був би мінімальним елементом масиву. Але заздалегідь (до початку роботи алгоритму) значення мінімального елемента, як правило, невідоме. Отже, необхідно попередньо переглянути весь масив, знайти мінімальний елемент і перемістити його на початок масиву (фактично це виконання першого циклу сортування методом вибору). Тепер, коли перший елемент вже знаходиться у потрібній позиції, можна запустити алгоритм сортування методом вставок без перевірки на вихід за межі початку масиву. Програма для покращеного методу вставок виглядає так:

```
program modify_insertion_sort_string;
uses
  CRT, FileUtil;
var
```

```
  str: string;

{Процедура, що реалізує покращений
 алгоритм сортування методом вставок}
procedure insert(var str:string);
var
  i, j: integer;
  v: char;
  min: integer;
begin
  {Пошук мінімального елемента у
  рядку} min:=1;
  for i:= 2 to length(str) do
  if str[i] < str[min] then min:=i;
  {Змінюємо місцями знайдений символ із
   першим символом у рядку}
  v:= str[1];
  str[1]:= str[min];
  str[min]: = v;
  {Реалізація власне методу
  вставок} for i:= 2 to
  length(str) do begin
    v:= str[i]; j:= i - 1;
    // Тепер у циклі перевіряється лише одна умова
    while str[j] > v do
    begin
      str[j+1]:= str[j];
      dec(j);
    end;
```

```
    str[j+1]:= v;
  end;
end;
begin
  str:= 'dkfhytcba';
  writeln(UTF8ToConsole('Початковий рядок:'),
          UTF8ToConsole(str));
  insert(str);
  writeln(UTF8ToConsole('Відсортований рядок: '),
          UTF8ToConsole(str));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Як вправу напишіть програми сортування рядка з кирилицею, масиву рядків та файлу менеджерів покращеним методом вставок.

Як і попередні алгоритми, сортування методом вставок належить класу O(n2). Якщо список частково відсортований, алгоритм методу вставок працює дуже швидко і має порядок O(n). Важливим є те, що алгоритм є стійким.

4.1.4 Метод швидкого сортування

Алгоритм швидкого сортування було розроблено К. Хоаром у 1960 році. Швидке сортування називають ще сортуванням з поділом або просто алгоритмом сортування Хоара. Суть методу полягає в наступному: вибирається якийсь елемент списку, який називають базовим або опорним. Після цього сортований список ділиться на дві частини: у ліву поміщаються всі елементи менші за базовий елемент, у праву елементи, більші за базовий елемент.

мента, а сам базовий елемент при цьому опиниться на потрібному місці у списку. Далі отримані два списки сортуються так само, тобто. знову вибираються базові елементи (вже всередині підсписків), підсписки діляться на два, у ліву поміщаються елементи менші за базовий елемент, у праву елементи, більші за базовий елемент тощо. При реалізації алгоритму процедура, що виконує основні дії з сортування, викликає саму себе. Такий виклик функції чи процедури себе називається рекурсією. Розглянемо алгоритм докладніше.

Спочатку поговоримо про вибір базового елемента. Ідеальним випадком було б вибір базового елемента середнього за значенням елемента списку. Але для цього потрібно попередньо переглянути весь список, обчислити середнє значення, потім знову переглянути весь список – чи є середнє значення серед елементів списку і все це рекурсивно! За кількістю операцій це рівносильно самому сортуванню.

Якщо в якості базового вибирати мінімальний або максимальний елемент списку, то, по-перше, для цього все одно потрібен попередній перегляд всього списку (підписків надалі), по-друге, що ще гірше, при розділенні списку на два, один з підписків виявиться порожнім, оскільки всі елементи списку будуть по одному. При числі елементів списку n будуть виконані n рекурсивних дзвінків. При великому числі n може вистачити стекової пам'яті, причому алгоритм може легко зациклитися.

Не вдаючись у подальші тонкощі, скажімо, що найчастіше як базовий елемент беруть елемент середній за місцем знаходження елемента в списку. Отже, алгоритм:

Заводяться два покажчики (індекси) i та j. На $i = 1$ і $j = n$де n - початку число записів, що сортуються.

На кожному кроці виконується розбиття записів на дві підгрупи так, щоб

ліва підгрупа	правапідгрупа
$R_i < R_{base}$	$R_j > R_{base}$

R_i, R_j - поточні записи (елементи); R_{base} - базовий елемент розглянь-групи з n записів (елементів). Є два внутрішні цикли. Перший цикл робить перегляд елементів правої підгрупи праворуч. Порівнюють-ся R_{base} зR_jj $= n, n-1, \square,$ доти, доки не зустрінеться$R_j < R_{base}$. Потім другий внутрішній цикл виконує перегляд елементів лівої підгрупи сліва. Порівнюються R_{base} зR_i, i $= 1,2, \square$доти, доки не зустрінеться$R_i > R_{base}$. Тепер можливі два випадки:

1. $i < j$Це означає, що два елементи, на які вказують індекси розташовані в неправильному порядку. Дійсно, елемент, який знаходиться у лівому підсписку більше, ніж базовий елемент, а елемент, який знаходиться у правому підсписку, менший за базовий елемент.

Змінюємо місцями елементи та продовжуємо виконання внутрішніх циклів.

2. $i >= j$Це означає, що поточна підгрупа успішно розділена. Виконання внутрішніх циклів завершується.

Зовнішній цикл починає свою роботу знову з визначення базового елементу, тепер уже для поточної підгрупи.

Теоретичні розрахунки ефективності методу показують у середньому $n\log2$ (n)операцій порівнянь.

Напишемо програму швидкого сортування для масиву з довільною кількістю елементів. Як елементи візьмемо цілі числа.

```
program quick_sort;
uses
  CRT, FileUtil;
```

```
type
  vector = array of integer;
var
  i, n: integer;
  sorted_array: vector; // сортований масив

  { ============ Метод швидкого сортування

======== } procedure QuickSort (var sorted_array:

vector);
  {rec_sort - рекурсивна процедура}
procedure rec_sort(first, last: integer;
                   var sorted_array: vector);
 { first, last - перший та останній елементи поточної
групи } var
  i: integer; // покажчик для лівого
  списку j: integer; // покажчик
  правого списку middle: integer; //
  Середній елемент списку temp:
  integer;
begin
  while (first < last) do
  begin
    middle: = sorted_array [(first + last) div
    2]; i:= pred(first);
    j: = succ
    (last); while
    true do begin
      repeat dec(j);
```

```
    until sorted_array[j] <= middle;
    repeat inc(i);
```

```
      until sorted_array[i] >= middle;
      if i >= j then break;
      temp: = sorted_array [i];
      sorted_array[i]: = sorted_array[j];
      sorted_array[j]:= temp;
      end;
  {рекурсивний виклик процедури}
  if first < j then rec_sort(first, j, sorted_array);
  first:= succ(j);
  end;
end;
{ ======= кінець процедури rec_sort
======== } begin
  { Перший виклик рекурсивної процедури}
  rec_sort(0, n - 1, sorted_array);
end;

{ ======= кінець процедури QuickSort ======== }

begin
  writeln(UTF8ToConsole('Введіть кількість елементів
  масиву')); readln(n);
  SetLength(sorted_array, n);
  writeln(UTF8ToConsole('Введіть елементи
  масиву')); for i:= 0 to n - 1 do
    read(sorted_array[i]);
  QuickSort(sorted_array);writeln(UTF8ToConsole('
  Відсортований масив:')); writeln;
```

```
  for i:= 0 to n - 1 do
    write(sorted_array[i], '');
  writeln;
  writeln;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

У програмі використані функції pred() та succ().

Функція succ збільшує (інкрементує) значення порядкової змінної. Можна інкрементувати:

- Символи;
- Нематеріальні числові типи;• Тип перерахування;
- Покажчики.

Функція pred зменшує значення порядкової змінної.

Вище ми неодноразово зазначали, що той самий алгоритм можна реалізовувати по-різному. Ось приклад іншої реалізації того самого алгоритму.

```
program quick_sort;
uses
  CRT, FileUtil;
type
  vector = array of integer;
var
  i, n: integer;
  sorted_array: vector; // сортований масив
```

```
  { ============ Метод швидкого сортування

======= } procedure QuickSort (var sorted_array:

vector);
  {rec_sort - рекурсивна процедура}
procedure rec_sort(first, last: integer;
                   var sorted_array: vector);
 { first, last - перший та останній елементи поточної
групи } var
  i: integer; // покажчик для лівого
  списку j: integer; // покажчик
  правого списку middle: integer; //
  Середній елемент списку temp:
  integer;
begin
  i: = first; // Перший елемент поточного списку
  j: = last; // останній елемент поточного списку
  middle: = sorted_array [(first + last) div 2];
  repeat
    while sorted_array[i] < middle do
      i:= i + 1;
    while middle < sorted_array [j] do
      j: = j - 1;
    if i <= j then
    begin
      temp: = sorted_array [i];
      sorted_array[i]: = sorted_array[j];
      sorted_array[j]:= temp;
      i:= i + 1;
```

```
      j: = j -
      1;
    end;
```

```
  until i > j;
  {рекурсивні виклики процедури для лівих та правих
  підписів} if first <j then rec_sort(first, j,
  sorted_array); if i < Last then rec_sort(i, Last,
  sorted_array);
end;
 { ================================================= }
begin
  {перший виклик рекурсивної процедури}
  rec_sort(0, n - 1, sorted_array);
end;
      {=======================================}
begin
  writeln(UTF8ToConsole('Введіть кількість елементів
  масиву')); readln(n);
  SetLength(sorted_array, n);
  writeln(UTF8ToConsole('Введіть елементи
  масиву')); for i:= 0 to n - 1 do
    read(sorted_array[i]);
  QuickSort(sorted_array);writeln(UTF8ToConsole('
  Відсортований масив:')); writeln;
  for i:= 0 to n - 1 do
    write(sorted_array[i], '');
  writeln;
  writeln;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Напишемо програму сортування файлу алгоритмом Хоара.

```
program quick_sort_file;
uses
  CRT, FileUtil, SysUtils, OutScr;
type
  manager = record
  name: string
  [18]; comp:
  integer;
 end;

var
  company: manager;
  f_not_sorted; f_sorted: File of manager;
  vector: array of manager;
  i, n: integer;
  name_file: string;

  { ========= Метод швидкого сортування ======== }

procedure QuickSort(var sorted_array: array of manager);

  {rec_sort - рекурсивна процедура}
procedure rec_sort(first, last: integer;
                   var sorted_array: array of manager);
 { first, last - перший та останній елементи поточної
групи } var
  i: integer; // покажчик для лівого списку
  j: integer; // покажчик для правого списку
```

```
  middle: manager; // Середній елемент списку
  temp: manager;
begin
  while (first < last) do
  begin
    middle: = sorted_array [(first + last) div
    2]; i:= Pred(first);
    j: = Succ
    (last); while
    true do begin
      repeat dec(j);
      until sorted_array[j].name <= middle.name;
      repeat inc(i);
      until sorted_array[i].name >= middle.name;
      if i >= j then break;
      temp: = sorted_array [i];
      sorted_array[i]: = sorted_array[j];
      sorted_array[j]:= temp;
      end;
 {рекурсивний виклик процедури}
if first < j then rec_sort(first, j, sorted_array);
  first:= Succ(j);
  end;
end;
 { ======================================= }
begin
  {перший виклик рекурсивної процедури}
  rec_sort(0, n - 1, sorted_array);
end;
```

```
begin
  {За потреби вкажіть повний шлях до
   файлу або скопіюйте файл у папку з цим
   проектом}
  if not FileExists('File_not_sorted.dat') then
  begin
    writeln(UTF8ToConsole('Файли не існують'));
    writeln(UTF8ToConsole('Спочатку створіть їх'));
    writeln(UTF8ToConsole('Натисніть будь-яку
    клавішу')); readkey;
    exit;
  end;
  AssignFile(f_not_sorted, 'File_not_sorted.dat');
  Reset(f_not_sorted);
  // Визначення кількості записів у
  файлі n: = System.FileSize
  (f_not_sorted); SetLength(vector,
  n);
  {Підготовка до внутрішнього
   сортування, зчитування
   записів файлу до масиву,
   у процедуру передається масив, а не файл. i:=
  0;
  while not Eof(f_not_sorted) do
  begin
     Read(f_not_sorted, company); vector[i]:
     = company; // Сортований масив i:= i +
     1;
  end;
  QuickSort(vector); // виклик процедури швидкого
```

```
сортування
CloseFile(f_not_sorted);AssignFile(f_sor
ted, 'File_sorted.dat');
```

```
  Rewrite(f_sorted);for
  i:= 0 to n - 1 do
  Write(f_sorted, vector[i]);
  CloseFile(f_sorted);
  name_file:= 'File_sorted.dat';
  {Виклик процедури для виведення на екран
  зведеної відомості. Процедура знаходиться у
  модулі OutScr} output_to_screen(name_file);
  write(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Резюмуючи все вищесказане щодо методів сортування, можна сказати, що найчастіше використовуються метод вставок і алгоритм швидкого сортування Хоара. Щодо інших алгоритмів сортування, то ви можете їх знайти у спеціальній літературі [9, 10, 12].

4.2. Алгоритми пошуку

Пошук яких-небудь даних, поряд з сортуванням, є одним з найбільш поширених завдань при розробці складних програмних проектів, зокрема при розробці інформаційно-пошукових та інформаційно-довідкових систем. У будь-якій базі даних присутні завдання сортування та пошуку. Алгоритми пошуку є не менш складними та цікавими, ніж алгоритми сортування.

Розглянемо деякі алгоритми пошуку у масивах.

4.2.1 Пошук у масивах

Нехай нам дано якийсь масив. Потрібно знайти елемент масиву за заданим ключем. Перше, що спадає на думку, щоб вирішити це завдання, це послідовно порівнювати елементи масиву із заданим ключем, доки не буде знайдено шуканий елемент. Якщо елемент знайдено, алгоритм має видати номер індексу цього елемента у масиві. Якщо ж такого елемента в масиві немає, то алгоритм має нам якимось чином повідомити про це. Найчастіше у разі алгоритм повертає значення, якого явно немає у масиві, а точніше такого індексу. Такий алгоритм називається алгоритмом лінійного пошуку. Напишемо відповідну функцію у припущенні, що пошук ведеться у масиві цілих чисел.

```
функція LinearSearch(var a: array of integer;
                     key: integer): integer;
var
  i: integer;
begin
```

```
  for i:= 0 to High(a) do
  begin
    if key = a[i] then
    begin
      LinearSearch:= i;
      exit;
    end;
  end;
LinearSearch:= -1;
end;
```

Отже, ця функція повертає нам номер індексу в масиві, де знаходиться елемент, що шукається. Повертати саме значення немає сенсу, ми його знаємо (значення key). У разі відсутності елемента, що шукається, функція повертає -1. Зверніть увагу, що в самому масиві може бути і є елемент зі значенням -1, але функція повертає індекс елемента в масиві. Як ми знаємо, індекс не може бути негативним, повернення -1 і говорить нам про те, що елемента, що шукається в масиві, немає.

Програма лінійного пошуку для цілісного масиву:

```
program search_in_array;
uses
  CRT, FileUtil;
const SizeOfFile =
9; var
  a:array[0..SizeOfFile] of integer =
  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  n: integer;
  key: integer;
```

```
  { ============ Лінійний пошук ======== }
function LinearSearch(var a:array of integer;
                      key: integer): integer;
var
  i: integer;
begin
  for i:=0 to SizeOfFile do
  begin
    if key = a[i] then
    begin
      LinearSearch := i;
      exit;
    end;
  end;
LinearSearch:= -1;
end;
{ ======================================== }

begin
  writeln(UTF8ToConsole('Введіть ключ для
  пошуку')); readln(key);
  {виклик функції пошуку}
  n:= LinearSearch(a, key);
  if n= -1 then
     writeln(UTF8ToConsole('Такого елемента в масиві
  немає')) else
     writeln(UTF8ToConsole('Елемент знайдено, його номер у
     масиві'),
n);
  write(UTF8ToConsole('Натисніть будь-яку клавішу'));
```

```
  readkey;
end.
```

Напишемо програму пошуку менеджера на прізвище. Для зручності перевірки результатів роботи програми одночасно виводиться весь файл менеджерів та вказується номер шуканого запису.

```
program search_in_file;
uses
  CRT, FileUtil, SysUtils, OutScr;
type
    manager = record
    name: string [18];
    comp: integer;
 end;
var
  company: manager;
  f_not_sorted: File of manager; // Файлова змінна
  vector: array of manager;
  i, n: integer;
  name_file: string;
  name_manager: string;

  { ========= Лінійний пошук ======== }
function LinearSearch(var not_sorted_array:
                      array of manager; name_manager:
                      string): integer;
var
  i: integer;
```

```
begin
  for i:= 0 до High(not_sorted_array) do
  begin
    if name_manager = not_sorted_array[i].name then
    begin
      LinearSearch := i;
      exit;
    end;
  end;
LinearSearch:= -1;
end;
{ ================================== }

begin
  {За потреби вкажіть повний шлях до
   файлу або скопіюйте файл у папку з цим
   проектом}
  if not FileExists('File_not_sorted.dat') then
  begin
    writeln(UTF8ToConsole('Файл не існує'));
    writeln(UTF8ToConsole('Спочатку створіть
    його')); writeln(UTF8ToConsole('Натисніть будь-
    яку клавішу')); readkey;
    exit;
  end;
  AssignFile(f_not_sorted, 'File_not_sorted.dat');
  Reset(f_not_sorted);
  // Визначення кількості записів у
  файлі n: = System.FileSize
  (f_not_sorted); SetLength(vector,
```

```
  n);
```

```
  {Підготовка пошуку,
   зчитування записів файлу в масив,
   у функцію передається масив, а не файл. i:=
  0;
  while not Eof(f_not_sorted) do
  begin
     Read(f_not_sorted, company);
vector[i]: = company; // масив, у якому вестиметься пошук
     i:= i + 1;
  end;
  CloseFile(f_not_sorted);
  writeln(UTF8ToConsole('Введіть прізвище менеджера'));
  readln(name_manager);
  {виклик функції пошуку}
  n:= LinearSearch(vector, name_manager);
  name_file:= 'File_not_sorted.dat';
  {Виклик процедури для виведення на екран списку
  менеджерів. Процедура знаходиться у модулі
  OutScr} output_to_screen(name_file);
  if n= -1 then
     writeln(UTF8ToConsole('Такого менеджера
  немає')) else
     writeln(UTF8ToConsole('Менеджер знайдений,  його
                                  номер  '), n+1);
  write(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Очевидно, що час пошуку за цим алгоритмом залежить від розміру маси-

ва – кількості його елементів n. Тобто. цей алгоритм належить класу O(n).

Якщо масив невпорядкований, то єдиний спосіб знайти елемент, це лінійний пошук. Якщо ж масив упорядкований, то існують більш швидкі та ефективні алгоритми. Найчастіше використовується алгоритм двоичного (бінарного) пошуку. Його також часто називають метод поділу навпіл.

Суть методу в тому, що, користуючись тим, що масив відсортований, замість перегляду всіх елементів масиву підряд, знаходимо елемент, що знаходиться посередині масиву. Назвемо його середнім елементом (не за значенням, а за місцем у масиві). Потім порівнюємо шуканий елемент із цим середнім елементом. Якщо шуканий елемент менший за середній, то шукати тепер його слід лише серед тих, які менші за середній. Таким чином, діапазон пошуку зменшується рівно наполовину. Беремо ту половину масиву, де слід шукати наш елемент і знову знаходимо середній елемент, знову порівнюємо його з шуканим елементом і знову звужуємо зону пошуку наполовину і т.д. продовжуємо процес до тих пір, поки в аналізованих підмасивах не залишиться один елемент. Якщо він дорівнює шуканому, значить елемент знайдено, якщо не дорівнює, значить такого елемента в масиві немає.

За виконання алгоритму кожному наступному циклі розмір масиву зменшується вдвічі, звідси оцінка швидкості роботи алгоритму O(log2n).

Напишемо програму бінарного пошуку в масиві цілих чисел:

```
program search_in_array;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
const SizeOfFile =
9; var
  a: array[0..SizeOfFile] of integer =
```

```
  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  n: integer;
  key: integer;

  { ======== Бінарний пошук ======== }
функція BinarySearch(var a: array of integer;
                     key: integer): integer;
var
  left, right, middle: integer;
begin
  left:= Low(a);
  right:= High(a);
  repeat
  middle: = (left + right) div
  2; if key < a[middle] then
    right:= middle - 1
  else
    if key > a[middle] then
      left:= middle + 1
    else
    begin
      BinarySearch:= middle;
      exit;
    end;
  until left > right;
  BinarySearch:= -1;
end;
{ ======================================== }
```

```
begin
  writeln(UTF8ToConsole('Введіть ключ для
  пошуку')); readln(key);
  {виклик функції пошуку}
  n:= BinarySearch(a, key);
  if n= -1 then
     writeln(UTF8ToConsole('Такого елемента в масиві
  немає')) else
     writeln(UTF8ToConsole('Елемент знайдений, його
  номер'), n); writeln(UTF8ToConsole('Натисніть будь-
  яку клавішу')); readkey;
end.
```

Напишемо програму пошуку у файлі менеджерів на прізвище бінарним методом:

```
program search_in_file;
uses
  CRT, FileUtil, SysUtils, OutScr;
type
    manager = record
    name: string [18];
    comp: integer;
 end;
var
  company: manager;
  f_not_sorted; f_sorted: File of manager;
  vector: array of manager;
  i, n: integer;
```

```
  name_file: string;
  name_manager: string;
  { ======= Бінарний пошук ======== }
функція BinarySearch(var a: array of manager;
                     name_manager: string): integer;
var
  left, right, middle: integer;
begin
  left:= Low(a);
  right:= High(a);
  repeat
  middle: = (left + right) div 2;
  if name_manager < a[middle].name then
    right:= middle - 1
  else
    if name_manager > a[middle].name then
      left:= middle + 1
    else
    begin
      BinarySearch:= middle;
      exit;
    end;
  until left > right;
  BinarySearch:= -1;
end;
{ ======================================== }
begin
  {За потреби вкажіть повний шлях до
   файлу або скопіюйте файл у папку з цим
   проектом}
```

```
if not FileExists('File_not_sorted.dat') then
begin
  writeln(UTF8ToConsole('Файл не існує'));
  writeln(UTF8ToConsole('Спочатку створіть
  його')); writeln(UTF8ToConsole('Натисніть будь-
  яку клавішу')); readkey;
  exit;
end;
AssignFile(f_sorted, 'File_sorted.dat');
Reset(f_sorted);
n:= System.FileSize(f_sorted);
SetLength(vector, n);
{Підготовка до пошуку,
 зчитування записів файлу в масив,
 у функцію передається масив, а не файл. i:=
0;
while not Eof(f_sorted) do
begin
   Read(f_sorted, company);
   vector[i]: = company; // масив, який сортуватиметься
   i:= i + 1;
end;
CloseFile(f_sorted);
writeln(UTF8ToConsole('Введіть прізвище менеджера'));
readln(name_manager);
{виклик функції пошуку}
n:= BinarySearch(vector, name_manager);
name_file:= 'File_sorted.dat';
{Виклик процедури для виведення на екран зведеної
відомості.
```

```
  Процедура знаходиться у модулі OutScr}
  output_to_screen(name_file);
   if n= -1 then
     writeln(UTF8ToConsole('Такого менеджера
  немає')) else
     writeln(UTF8ToConsole('Менеджер знайдений,  його
                                  номер  '), n+1);
  write(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

4.2.2 Вставка та видалення елементів у впорядкованому масиві

Часто буває необхідно вставляти нові елементи в упорядкований масив, а також видаляти елементи з масиву. Такі завдання часто виникають у системах управління базами даних, де користувачі вводять велику кількість даних. "Пристойна" система має давати користувачам можливість коригування введених даних, тобто. вставки нових даних, зміни вже введених даних та видалення даних. Завдання це не таке тривіальне, як може здатися на перший погляд. Нехай необхідно вставити новий елемент x у впорядкований за зростанням масив a[1..n] у місце з номером k, де k<n.

Якщо просто привласнити

a[k]: = x;

то елемент з індексом k, який був там раніше, буде втрачено. Отже, попередньо необхідно зрушити всі елементи a[k], a[k+1], …, a[n] однією позицію вправо. Це можна зробити за допомогою циклу:

```
for i:= n downto k
    a[i+1]:= a[i];
```

```
        inc(n);
```

Те саме при видаленні елемента з масиву. Якщо видаляється елемент a[k], то всі елементи, що залишилися a[k+1], …, a[n] треба зрушити на одну позицію вліво, щоб закрити виниклу "дірку" в k-й позиції масиву.

Код для зсуву елементів масиву при видаленні елемента a[k] буде таким:

```
    for i: = k + 1 to n
        do a [i-1]: = a
        [i]; dec(n);
```

Напишемо програму вставки нового елемента в масив. Припустимо, що масив складається з цілих чисел 1, 2, 3, 4. Моделюємо ситуацію, коли користувач при введенні масиву припустився помилки, пропустив число 3. Оскільки оператор readln "не реагує" на введення порожнього рядка, була введена наступна послідовність чисел: 1 , 2 . скоригувати дані, введені їм у масив.

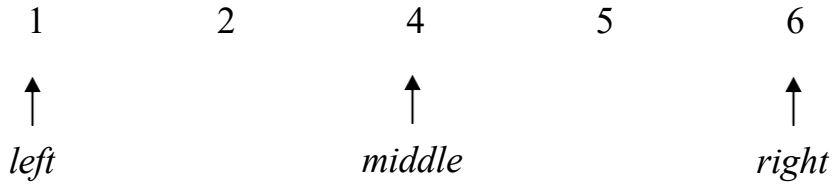
```
program project1;
uses
  CRT, FileUtil;
var
  i, n: integer;
  a: array of integer;
  NewElement: integer;
  IndexOfNewElement: integer;
  SizeOfArray: integer;
  answ: char;
```

```
begin
  writeln(UTF8ToConsole('Введіть розмір масиву'));
  readln(SizeOfArray);
  SetLength(a, SizeOfArray);
  writeln(UTF8ToConsole('Введіть елементи
  масиву')); for i:= 0 to SizeOfArray - 1 do
  read(a[i]);writeln(UTF8ToConsole('Введ
  ений масив:')); for i:= 0 to
  SizeOfArray - 1 do write(a[i], '');
  writeln;
  writeln(UTF8ToConsole('Відкоригуйте дані'));
  writeln(UTF8ToConsole('Якщо все правильно, натисніть
  Enter')); writeln(UTF8ToConsole('інакше - будь-яку
  клавішу'));
  answ: = readkey;
  if answ = #13 then exit;
  writeln(UTF8ToConsole('Введіть новий
  елемент')); readln(NewElement);
  writeln(UTF8ToConsole('Введіть індекс у
  масиві,'));
  writeln(UTF8ToConsole('куди потрібно вставити новий
  елемент')); readln(IndexOfNewElement);
  for i:= SizeOfArray - 1 downto IndexOfNewElement do
    a[i+1]:= a[i];
  a[IndexOfNewElement]:= NewElement;
  if IndexOfNewElement= SizeOfArray then
  inc(SizeOfArray);
  writeln(UTF8ToConsole('Масив після вставки нового
  елемента')); for i:= 0 to SizeOfArray - 1 do
  write(a[i], ''); writeln;
```

```
writeln(UTF8ToConsole('Натисніть будь-яку
клавішу'));
```

```
  readkey;
end.
```

Як завжди, шукаємо можливість покращення програми. У попередній програмі користувач сам вводив індекс у масиві, куди потрібно вставити новий елемент. Виявляється, алгоритм бінарного пошуку дозволяє автоматично вставляти новий елемент у потрібне місце. Яким чином? Розглянемо на конкретному прикладі. Нехай є масив із 5 елементів. Необхідно вставити число 3. Згадаймо алгоритм бінарного пошуку. Початкове розташування покажчиків буде, оскільки показано малюнку 4.12.



Мал. 4.12. Початкове розташування вказівників

При цьому значення змінних дорівнюватимуть:

key= 3, left = 0 (не забувайте, нумерація в динамічних масивах з нуля!)

right= 4, middle = (left+right) div 2 = (0 + 4) div 2 = 2

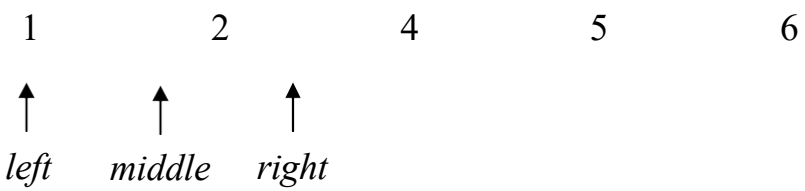
a[middle] = a[2] = 4

key= 3 < a[2] = 4, тобто. значення ключа менше значення середнього елемента. Відповідно до алгоритму, змінній right буде надано значення:

right= middle - 1 = 2 - 1 = 1

middle= (Left + Right) div 2 = (0 + 1) div 2 = 1

Нове розташування покажчиків показано малюнку 4.13.



Мал. 4.13. Нове розташування вказівників

a[middle] = a[1] = 2

key= 3> a [1] = 2, тобто. значення ключа більше значення середнього елемента, отже, змінної left буде присвоєно значення

left= middle + 1 = 1 + 1 = 2

Значення left виявляється більшим за right:

left= 2 > right = 1

Алгоритм пошуку закінчує свою роботу, шуканий елемент не знайдено. Але, подивіться чому рівне значення left. Воно дорівнює 2, тобто. саме тому індексу в масиві, куди має бути поміщений новий елемент, тобто. key рівний 3!

У алгоритмі достатньо змінити один оператор, там, де функція BinarySearch повертає -1 (елемент не знайдено!), тобто. замість

```
BinarySearch:= -1;
```

необхідно записати

```
BinarySearch:= left;
```

Отже, покращена програма попереднього прикладу. У цій програмі вставка нового елемента оформлена у вигляді процедури. Крім того, введено перевірки на правильність введення розміру масиву

```
program modify_insert;
uses
  CRT, FileUtil;
var
  i: integer;
  a: array of integer;
  NewElement: integer;
```

```
  IndexOfNewElement: integer;
  SizeOfArray: integer;
  answ: char;
функція BinarySearch(var a: array of integer;
                     key: integer): integer;
var
  left, right, middle: integer;
begin
  left:= Low(a);
  right:= High(a);
  repeat
  middle: = (left + right) div
  2; if key < a[middle] then
    right:= middle - 1
  else
    if key > a[middle] then
      left:= middle + 1
    else
    begin
      BinarySearch:= middle;
      exit;
    end;
  until left > right;
  BinarySearch:= left;
end;
procedure insert_new_element(var a: array of integer;
var NewElement: integer; var IndexOfNewElement: integer);
begin
  for i:= SizeOfArray - 1 downto IndexOfNewElement do
```

```
  a[i + 1]: = a[i];
  a[IndexOfNewElement]:= NewElement;
  if IndexOfNewElement= SizeOfArray then
  inc(SizeOfArray);
end;
begin
  writeln(UTF8ToConsole('Введіть розмір
  масиву')); while true do
  begin
    readln(SizeOfArray);
    if SizeOfArray = 0 then
    begin
      writeln(UTF8ToConsole('Розмір масиву не може бути = 0'));
      writeln(UTF8ToConsole('Введіть розмір масиву'));
    end
    else
    if SizeOfArray < 0 then
    begin
      writeln(UTF8ToConsole('Розмір масиву не може бути < 0'));
      writeln(UTF8ToConsole('Введіть розмір масиву'));
    end
    else break;
  end;
  SetLength(a, SizeOfArray);
  writeln(UTF8ToConsole('Введіть елементи
  масиву')); for i:= 0 to SizeOfArray - 1 do
  read(a[i]);writeln(UTF8ToConsole('Введ
  ений масив:')); for i:= 0 to
  SizeOfArray - 1 do
```

```
  write(a[i], ''); writeln;
  writeln(UTF8ToConsole('Відкоригуйте дані'));
  writeln(UTF8ToConsole('Якщо все правильно, натисніть
  Enter')); writeln(UTF8ToConsole('інакше - будь-яку
  клавішу'));
  answ: = readkey;
  if answ = #13 then exit;
  writeln(UTF8ToConsole('Введіть новий
  елемент')); readln(NewElement);
  IndexOfNewElement:= BinarySearch(a, NewElement);
  insert_new_element(a, NewElement, IndexOfNewElement);
  writeln(UTF8ToConsole('Масив після вставки нового
  елемента')); for i:= 0 to SizeOfArray - 1 do
  write(a[i], ''); writeln;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

У програмі передбачається, що початковий розмір масиву не повинен змінюватись. Якщо ви пам'ятаєте, у прикладі ми моделювали ситуацію, коли користувачеві необхідно було відкоригувати помилково введені дані. Якщо ж вам потрібно, щоб елемент, що вставляється, розширював вихідний масив, вам потрібно в процедурі insert_new_element замінити умовний оператор

```
if IndexOfNewElement = SizeOfArray then
  inc(SizeOfArray);
```

на просто оператор інкременту

```
inc(SizeOfArray);
```

Пропоную вам самостійно написати процедуру вставки в масив унікального елемента, тобто. якщо у вихідному масиві є такий елемент, то вставка не проводиться.

4.3. Динамічні структури даних

При описі деяких завдань застосовуються абстрактні структури даних, зокрема графи.

Орієнтований граф - це система з двох множин $G = (X, U)$, де X - безліч елементів, званих вершинами, а U - певне на множині X відношення (тобто $U \supseteq X \times X$), елементи якого називаються дугами чи ребрами. Якщо a і b – вершини, то (a, b) – ребро. Говорять, що ребро спрямоване від a до b .

Неорієнтований граф - це такий граф, в якому відсутня орієнтація. тація ребер, тобто. (a, b) (b, a).

Якщо кожному ребру поставити у відповідність деяке число, це зважений граф.

Вершини графа або його ребра (або ті та інші) можуть бути позначені. Як позначки можуть використовуватися символи або числа.

Мал. 4.14. Приклади графів

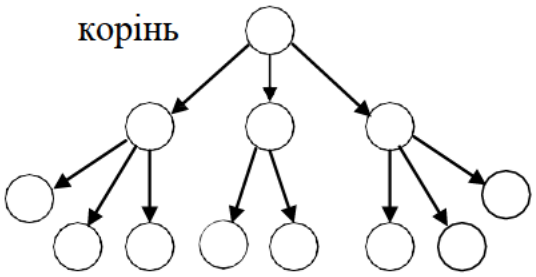
Шляхом у графі називається послідовність вершин, пов'язаних між

собою ребрами. Дві вершини пов'язані між собою, якщо є шлях від однієї вершини до іншої.

Граф називається зв'язковим, якщо всі пари вершин пов'язані. Будемо говорити, що граф порожній, якщо в ньому немає вершин (а отже, і ребер).

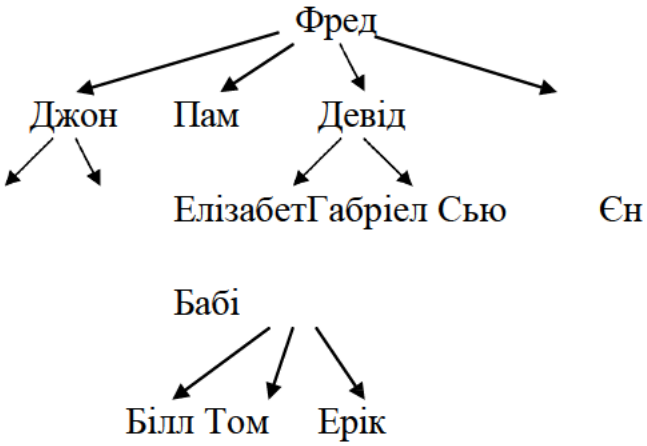
Розглянемо тепер графи спеціального вигляду, які називають деревами. Деревом називається зв'язковий граф, у якому:

1) є єдина особлива вершина, називається коренем, у якому не входить жодне ребро;
2) у всі інші вершини (інакше званих листям, а також вузлами) заходить тільки одне ребро, а виходить скільки завгодно ребер;
3) немає циклів (тобто замкнутих петель)



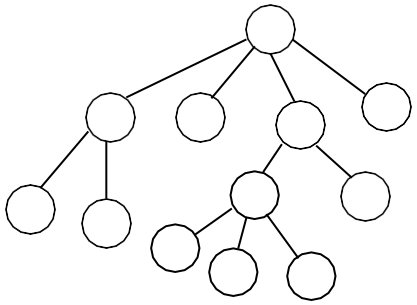
Мал. 4.15. Приклад дерева

На малюнках корінь вказують за допомогою наочного розташування, коли корінь зображується найвищою вершиною. За допомогою дерева ми можемо уявити родовід деякої людини.



Мал. 4.16. Родовід людини на ім'я Фред

Цьому родоводу відповідає дерево виду:



Мал. 4.17. Дерево, що відповідає родоводу Фреда

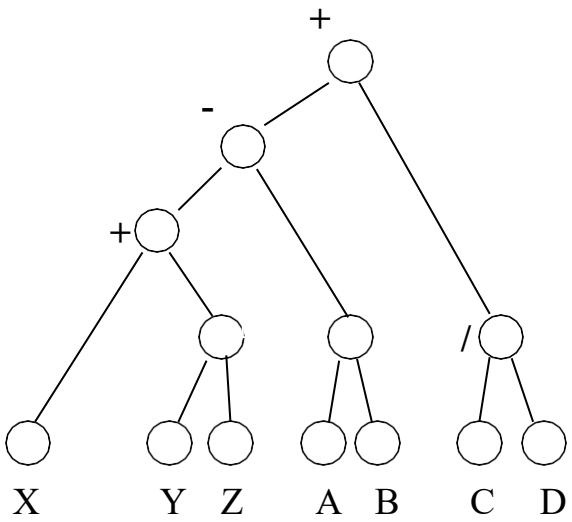
Тому вершини нижніх рівнів називають ще нащадками або синами.

ями.

Одне з перших застосувань дерев у програмуванні – це трансляція.

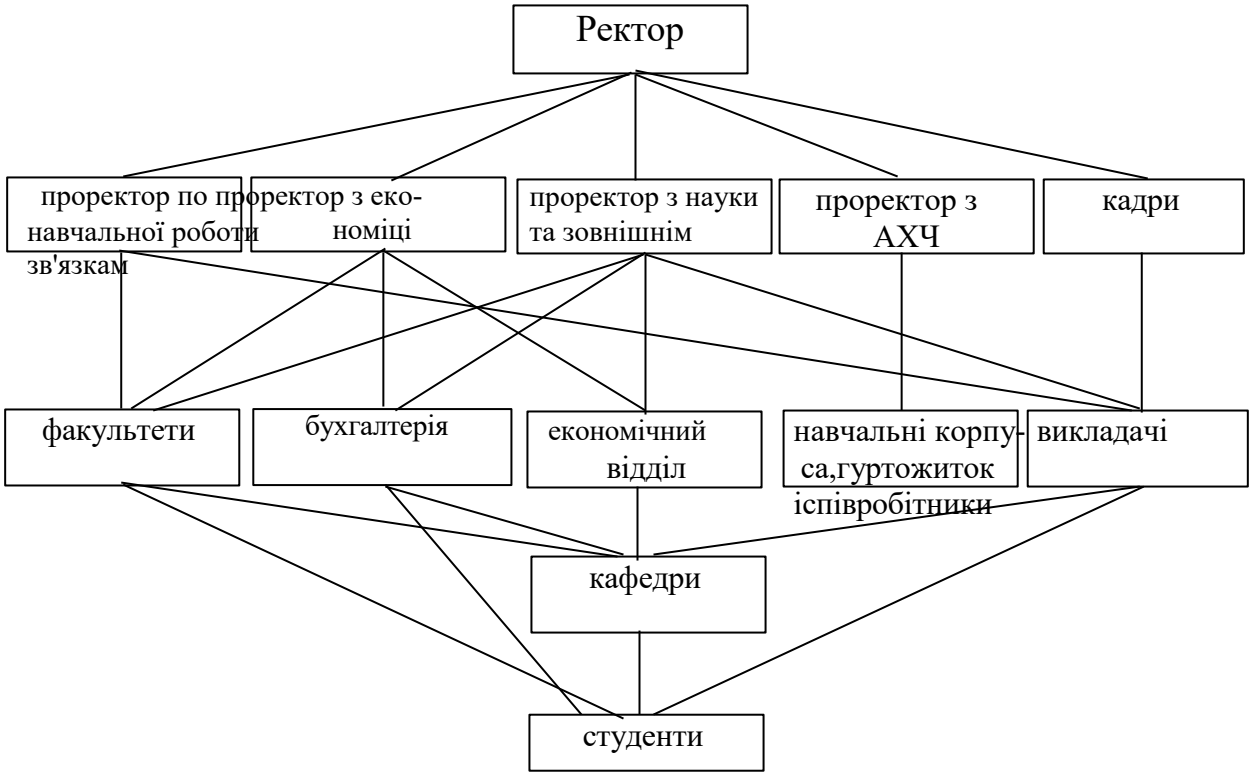
арифметичних виразів.Нехай є деяке арифметичне вираз: X+Y*ZA*B+C/D

При перекладі цього виразу на внутрішню мову компілятор будує дерево виду:



Мал. 4.18. Дерево, побудоване компілятором при розборі виразу

Будь-яка ієрархічна структура може бути представлена у вигляді дерева. Розглянемо спрощену структуру типового університету.



Мал. 4.19. Структура типового університету

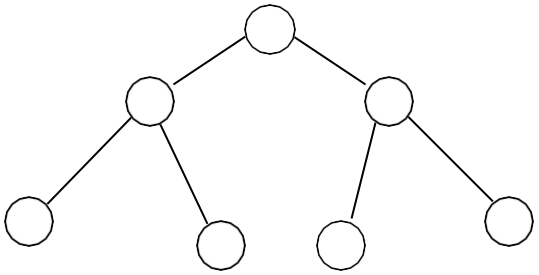
Намалювати відповідне дерево надається самому читачеві.

Кількість ребер, що виходять з будь-якої вершини, називається ступенем вузла або ступенем вершини. Якщо з вершини не виходить жодного ребра, ступінь такої вершини дорівнює нулю.

Найважливішим класом дерев, які найчастіше використовуються у програмуванні, є так звані двійкові або бінарні дерева. Двійковим деревом називається таке дерево, з кожної вершини якого виходить не більше двох ребер, тобто. у ступінь двійкового дерева не перевищує двох. Суворе бінарне дерево складається тільки з вузлів, що мають ступінь два або нуль. Нестрогое бінарне дерево містить вузли зі рівнем 0, 1 або 2.

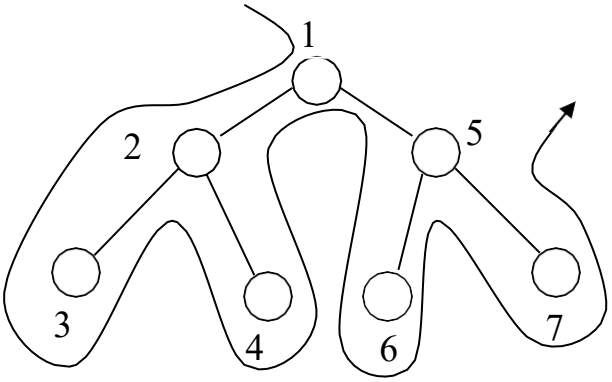
У бінарному дереві кожному рівні n може бути трохи
більше 2n-1 вершин.

Бінарне дерево називається повним, якщо у строгому бінарному дереві на кожному n-му рівні містяться всі 2n-1 вершин, рис. 4.20.



Мал. 4.20. Бінарне (двійкове) дерево

Найчастіше використовувані дії над деревами – це обхід дерева. Обходячи дерево, ми можемо щось робити з вершиною, яку обходимо, наприклад, друкувати номер вершини. Розрізняють 3 способи обходу дерева:



Мал. 4.21. Обхід дерева

1) обхід зверху, при цьому спочатку обробляють корінь, потім ліве піддерево, а потім праве піддерево. Тоді будуть надруковані 1, 2, 3, 4, 5, 6, 7;
2) обхід ліворуч. Тут спочатку обробляють ліве піддерево, потім корінь, потім праве піддерево. Будуть надруковані 3, 2, 4, 1, 6, 5, 7;
3) обхід знизу. Тут обробляються спочатку ліве піддерево, потім праве піддерево, потім корінь. Будуть надруковані 3, 4, 2, 6, 7, 5, 1.

Обхід ліворуч часто використовується в алгоритмах сортування при роботі з таблицями.

Розглянемо ще одну інформаційну структуру – стек.

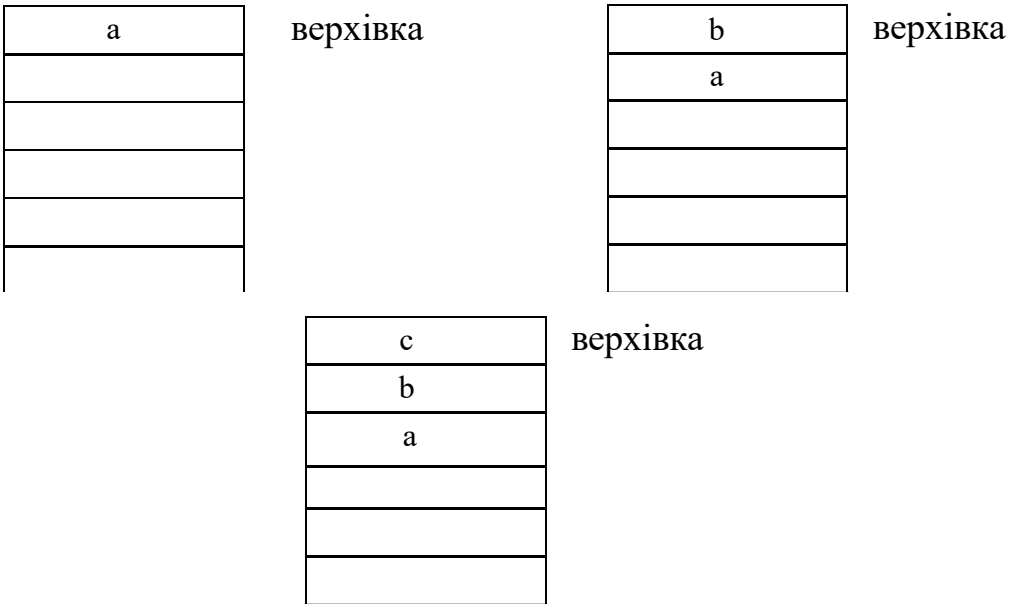
Стек це динамічна структура, пристосована для того, щоб додавати елементи до стек та витягувати їх звідти за певними правилами – черговий елемент у стек можна помістити або взяти лише через спеціальне місце, яке

називається верхівкою стека. У верхівці стека знаходиться завжди еле-

мент, вміщений до нього останнім. Стек працює за принципом "останнім прийшов, першим пішов" ("Last In - First Out", LIFO).

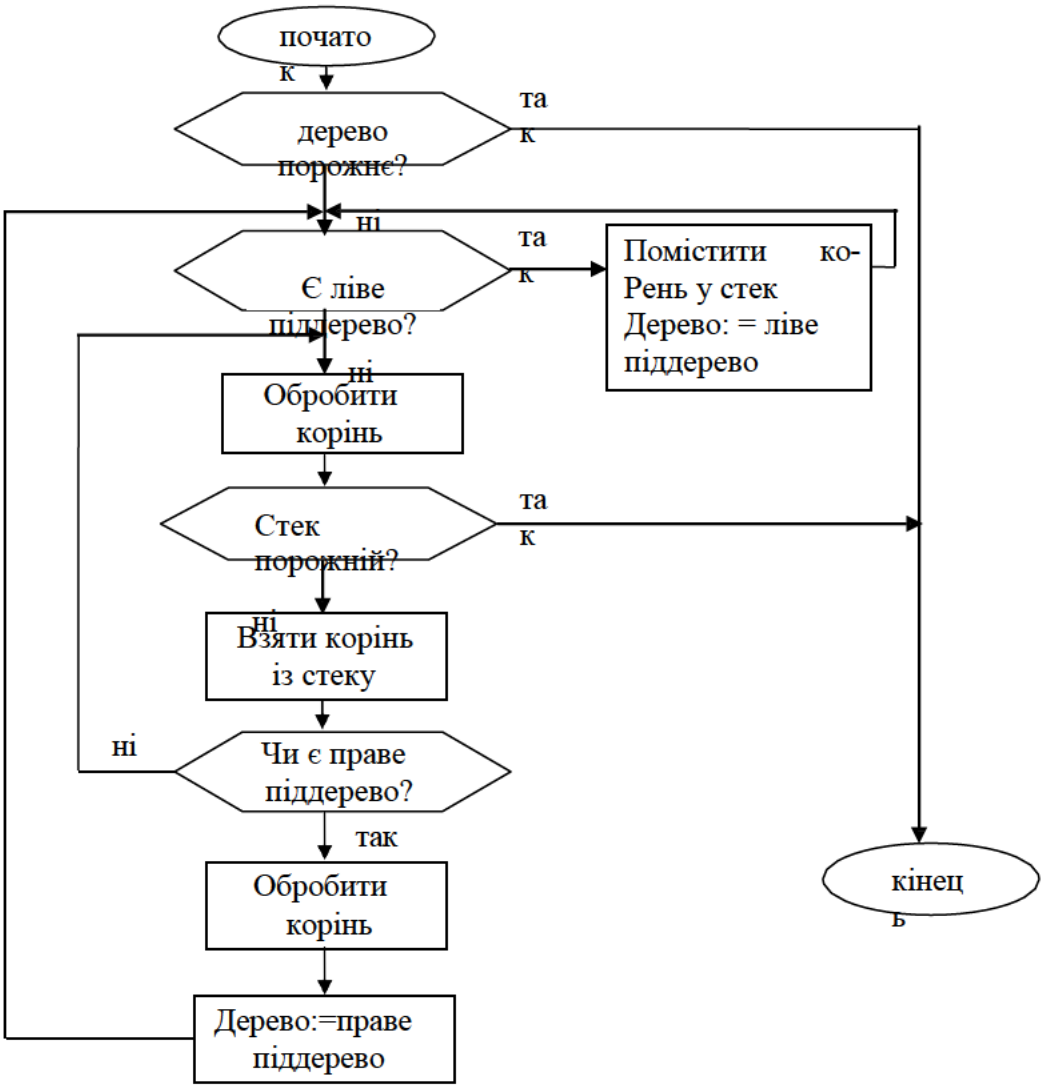
Особам чоловічої статі, можливо, буде легше зрозуміти принцип роботи стека, якщо вони уявлять собі обойму пістолета. Патрон, поміщений в обойму останнім, першим піде в ціль. Особам жіночої статі приємніше буде асоціювати стек зі чаркою тарілок. Тарілка, поміщена в стопку останньої, першою піде у "справу".

Розглянемо процес приміщення у стек трьох елементів a, b, c. Спочатку стек нехай буде порожнім. На малюнку 4.22 показано послідовні стани стека при додаванні елементів a, b, c.



Мал. 4.22. Послідовні стани стека при додаванні елементів

Таким чином, останній доданий у стек елемент з виявляється у верхівці стека. Застосування стека розглянемо з прикладу складання алгоритму обходу двійкового дерева зліва.



Мал. 4.23 Алгоритм обходу двійкового дерева зліва

4.3.1 Подання в пам'яті комп'ютера динамічних структур.

Пам'ять комп'ютера складається з осередків, які називаються байтами. Кожен байт має свій номер чи інакше адресу. Нумерація йде від 0 до N-1. Де N обсяг оперативної пам'яті. Якщо ми хочемо записати в пам'ять комп'ютера значення певної змінної, ми не вказуємо конкретну адресу, куди має бути записано значення змінної. Ми просто даємо цій змінній ім'я, тобто. позначаємо її, наприклад А. Компілятор відведе цій змінній потрібну кількість байтів. Кожна змінна в пам'яті займає залежно від її типу певну кількість байтів, що розташовані поспіль. При цьому, адре-

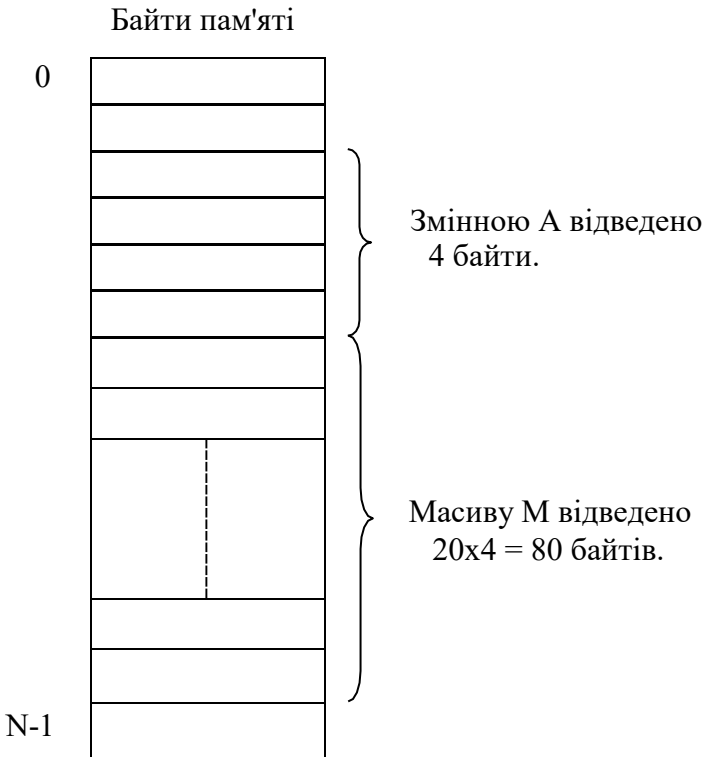
Сом змінною вважається адреса її першого байта. Таким чином, під А ми маємо на увазі не саме значення змінної, а її адресу (адреса першого байта). Хоча у програмі ми пишемо, наприклад А: = 28; насправді ми говоримо комп'ютеру: запиши в байти пам'яті починаючи з номера А число 28. Змінною А компілятор відведе чотири розміщені байти поспіль, оскільки число 28 є цілим.

Якщо ми хочемо уявити у пам'яті певний масив, під цей масив компілятор також відведе групу підряд розташованих байтів. Нехай цьому масиву надано ім'я М, кількість елементів масиву 20, тип масиву – цілий, тобто. кожен його елемент це цілі числа.

Тоді компілятор відведе під цей масив 20 * 4 = 80 байтів. Причому звернення до окремих елементів масиву використовується індекс M[1], M[I],B[I+K], тобто. у цьому випадку адреса складається з двох частин: одна частина вказує на початок усієї сукупності, що відводиться для масиву, а інша – адреса байта щодо початку сукупності. Зазвичай адресу початку сукупності називають базовою адресою, а адресу байта щодо початку сукупності – усуненням.

Таким чином:

```
Повна адреса = базова адреса + усунення
```



Мал. 4.24. Схема розподілу пам'яті змінної А та масиву М

Така сукупність називається вектор пам'яті. У мові Pascal масиви (як одновимірні, і багатовимірні) представляються векторами.

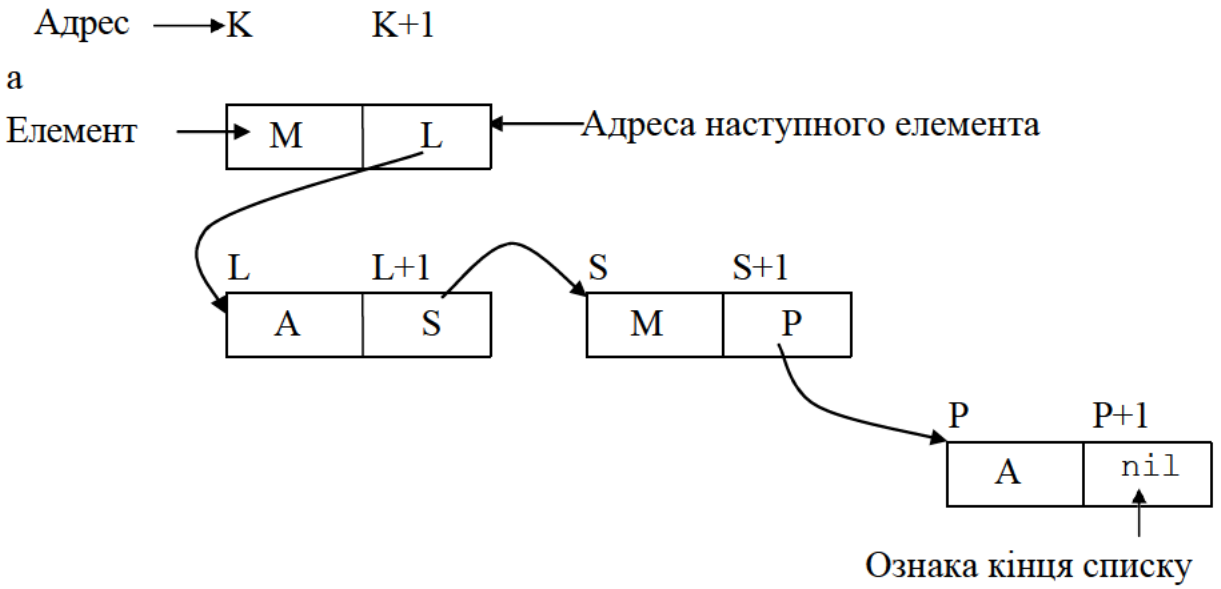
Але можна використовувати інший спосіб подання інформації.

Як відомо, послідовності байтів пам'яті ми можемо інтерпретувати як завгодно, зокрема їх можна інтерпретувати як адресу іншої іншої сукупності байтів. Наприклад, значення змінної А:

<A> = 2000← можна вважати адресою

Змінна, значенням якої є деяка адреса пам'яті, називається покажчиком, а адреса, на яку вказує покажчик, називається ланкою. Якщо є деяка послідовність покажчиків та ланок, то її називають ланцюжком або зчепленням ланок.

Розглянемо найпростішу структуру зчеплення – пов'язаний перелік. Подаємо, наприклад, за допомогою зв'язаного списку рядок символів "МАМА".



Мал. 4.25. Пов'язаний список

Як видно з малюнка, пов'язаний список складається з двох полів. Перше поле містить безпосередньо сам елемент списку. Друге поле – вказівник на наступний елемент списку. Причому поле покажчика останнього елемента списку містить спеціальну ознаку кінця списку – nil.

Тип nil означає " порожній " адресу, тобто. ця адреса не може бути адресою жодної змінної, вона є "нічийною", фіктивною адресою. У списках nil означає кінець списку.

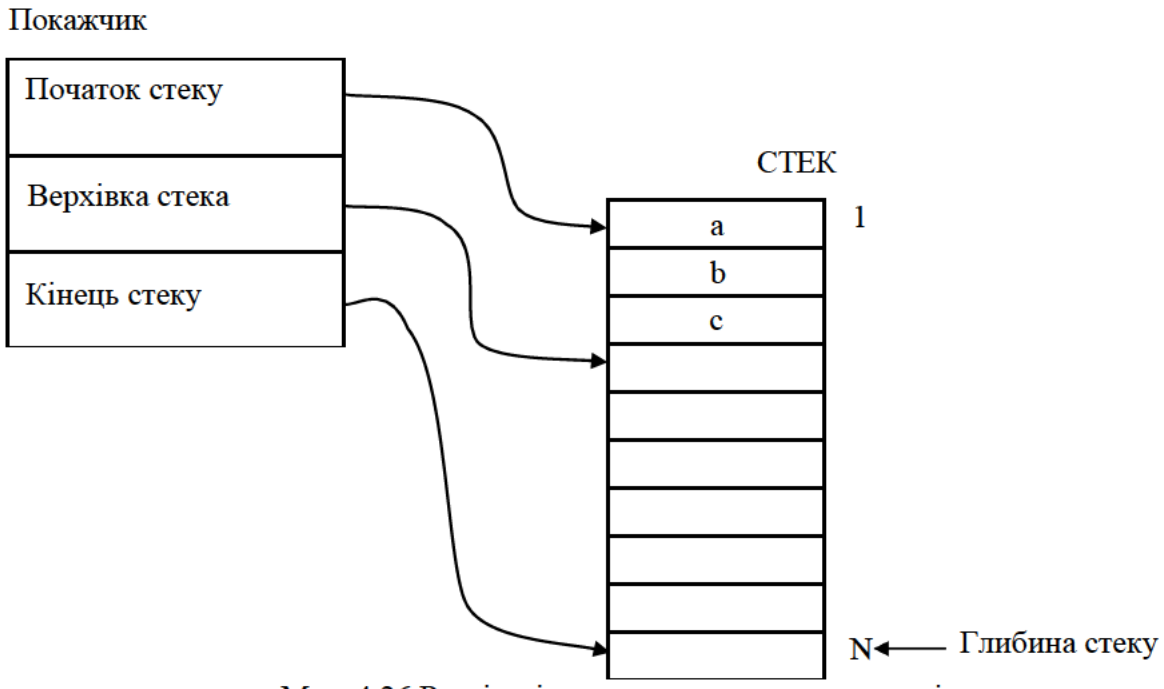
Пов'язані списки називають лінійними списками.

Стек та дерева можна реалізувати як за допомогою векторів пам'яті (масивів), так і за допомогою лінійних списків, використовуючи покажчики.

4.3.2 Реалізація стеку за допомогою масивів

Напишемо програму роботи зі стеком. Існує лише дві операції зі стеком – помістити елемент у стек та витягти елемент зі стека. Використовуються також терміни вштовхнути елемент у стек і виштовхнути елемент зі стека. Процедуру, що реалізує операцію приміщення елемента в стек, назвемо Put_Stack, а процедуру, що реалізує процес вилучення елемента зі стека - Take_Stack. Цілком можна було б назвати ці процедури і Push_Stack,

Pop_Stack. Спочатку реалізуємо стек за допомогою масиву, рисунок 4.26.



Мал. 4.26 Реалізація стека з урахуванням масивів.

З малюнка видно, що для реалізації стека нам потрібні два масиви, масив-покажчик, назвемо його ukaz[1..3] та масив-тіло стека, назвемо його СТЕК. Для певності нехай він складається із 100 елементів, тобто. глибина стеку дорівнює 100 елементам. Нехай також для визначеності елементами стека є просто цілі числа. На практиці елементами стека можуть бути і складніші структури, наприклад, записи. У масиві ukaz перший елемент, тобто. ukaz[1] містить індекс масиву СТЕК, що дорівнює початку стеку, зазвичай це

1. Елемент ukaz[3] містить індекс масиву СТЕК, рівний кінці стека, тобто. задану глибину стека N = 100 і зазвичай збігається з максимальним розміром масиву СТЕК. Елемент масиву ukaz[2] – це індекс у масиві СТЕК, куди буде вміщено новий елемент стека, а ukaz[2]-1 це індекс, яким зі СТЕК можна взяти останній вміщений у нього елемент.

Програма виводитиме на екран меню, за допомогою якого можна поміщати новий елемент у стек, брати елемент із верхівки стека та просматри-

вати весь масив стека.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
const
  SizeOfArray = 100;
type
  c= array[1..SizeOfArray] of integer;
  u = array [1..3] of integer;
var
  choose: integer;
  EMPTY: boolean;
  ERROR: boolean;
  CTEK: c;
  ukaz: u;
  ELEM: integer;
{ ================================================ }
procedure Put_Stack(var ELEM: integer; var CTEK: c;
                    var ukaz: u; var ERROR: Boolean);
{ ================================================ }

var k: integer;
begin
  if ukaz[2] > ukaz[3] then
      ERROR:= true
  else
  begin
```

```
    k:= ukaz[2];
    CTEK[k]:= ELEM;
    ukaz[2]:= k + 1;
  end;
end;

{ ================================================ }
procedure Take_Stack(var ELEM: integer; var CTEK: c;
                     var ukaz: u; var EMPTY: Boolean);
{ ================================================ }
var k: integer;
begin
  if ukaz[2] <= ukaz[1] then
    EMPTY:= true
  else
  begin
    k:= ukaz[2] - 1;
    ELEM: = CTEK [k];
    ukaz[2]:= k;
  end;
end;
procedure View_Stack(var CTEK: c; var ukaz: u);
var
  i: integer;
begin
  writeln(UTF8ToConsole('Масив
  стека')); for i:= 1 to ukaz[2] - 1 do
    write(CTEK[i], '');
  writeln;
```

```
end;

begin
  {Ініціалізація
  стека} ukaz[1]:=
  1;
  ukaz[2]:= 1;
  ukaz[3]:= SizeOfArray;
  repeat
    EMPTY:       =
    false;  ERROR:
    = false;
    writeln(UTF8ToConsole('Виберіть потрібний режим
    роботи:')); writeln(UTF8ToConsole('Помістити елемент у
    стек
                                                    1'));wr
    iteln(UTF8ToConsole('Взяти елемент із стеку
                                                    2'));wr
    iteln(UTF8ToConsole('Переглянути вміст стека 3'));
    writeln(UTF8ToConsole('Вихід із програми 4'));
    readln(choose);
    case choose of
    1: begin
         writeln(UTF8ToConsole('Введіть новий елемент
         стека')); readln(ELEM);
         Put_Stack(ELEM, CTEK, ukaz, ERROR);
         if ERROR then
         begin
           writeln(UTF8ToConsole('Переповнення стека.'));
           writeln(UTF8ToConsole('Збільшіть розмір
```

```
        масиву')); writeln(UTF8ToConsole('Натисніть
        будь-яку клавішу')); readkey;
      end;
    end;
```

```
    2: begin
         Take_Stack(ELEM, CTEK, ukaz, EMPTY);
         if EMPTY then
           writeln(UTF8ToConsole('Стек
         порожній')) else
           writeln(UTF8ToConsole('З стека взятий
                   елемент'), ELEM);
       end;
    3: View_Stack(CTEK, ukaz);
    end; {end of case}
  until choose = 4;
end.
```

Процедура Put_Stack просто поміщає новий елемент масив СТЕК по індексу ukaz[2] і потім збільшує значення цього індексу на одиницю. Перед цим процедура перевіряє стек переповнення, тобто. чи не перейшли ми межі масиву СТЕК.

Процедура Take_Stack повертає елемент масиву СТЕК з індексом ukaz[2]-1 та зменшує значення індексу на одиницю. Перед цим вона перевіряє чи не порожній уже стек.

Якщо уважно придивитися до процедур Put_Stack і Take_Stack, то виникає цілком резонне питання, а навіщо, власне кажучи, тут потрібен масив-покажчик ukaz? І ми будемо абсолютно праві. При реалізації стека можна обходитися одним масивом, тільки для розміщення елементів стека. А як покажчик використовувати просту ціло- чисельну змінну, яку назвемо також ukaz. Ось лістинг іншої, покращеної реалізації стека:

```
program modify_stack;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
const
  SizeOfArray = 100;
type
  c= array[1..SizeOfArray] of integer;
var
  choose: integer;
  EMPTY: boolean;
  ERROR: boolean;
  CTEK: c;
  ukaz: integer;
  ELEM: integer;
{ ================================================= }
procedure Put_Stack(var ELEM: integer; var CTEK: c;
                var ukaz: integer; var ERROR: Boolean);
{ ================================================= }

begin
  if ukaz > SizeOfArray then
      ERROR:= true
  else
  begin
    CTEK[ukaz]:= ELEM;
    ukaz:= ukaz + 1;
  end;
end;
```

```
{ ================================================ }
procedure Take_Stack(var ELEM: integer; var CTEK: c;
               var ukaz: integer; var EMPTY: Boolean);
{ ================================================ }

begin
  if ukaz = 1 then
    EMPTY:= true
  else
  begin
    ukaz:= ukaz - 1;
    ELEM:= CTEK[ukaz];
  end;
end;
procedure View_Stack(var CTEK: c; var ukaz: integer);
var
  i: integer;
begin
  writeln(UTF8ToConsole('Масив
  стека')); for i:= 1 to ukaz - 1 do
    write(CTEK[i], '');
  writeln;
end;

begin
  {Ініціалізація покажчика стека (початкового
  індексу в масиві)} ukaz: = 1;
  repeat
    EMPTY: = false;
```

```
ERROR: = false;
writeln(UTF8ToConsole('Виберіть потрібний режим роботи:')); writeln(UTF8ToConsole('Помістити елемент у стек
                                                          1'));wr
iteln(UTF8ToConsole('Взяти елемент із стеку
                                                          2'));wr
iteln(UTF8ToConsole('Переглянути вміст стека 3'));
writeln(UTF8ToConsole('Вихід із програми 4'));
readln(choose);
case choose of
1: begin
     writeln(UTF8ToConsole('Введіть новий елемент
     стека')); readln(ELEM);
     Put_Stack(ELEM, CTEK, ukaz, ERROR);
     if ERROR then
     begin
       writeln(UTF8ToConsole('Переповнення стека.'));
       writeln(UTF8ToConsole('Збільшіть розмір
       масиву')); writeln(UTF8ToConsole('Натисніть
       будь-яку клавішу')); readkey;
     end;
   end;
2: begin
     Take_Stack(ELEM, CTEK, ukaz, EMPTY);
     if EMPTY then
       writeln(UTF8ToConsole('Стек
     порожній')) else
       writeln(UTF8ToConsole('З стека взятий
                                 елемент '),
```

```
ELEM);
        end;
    3: View_Stack(CTEK, ukaz);
```

```
    end; {end of case}
  until choose = 4;
end.
```

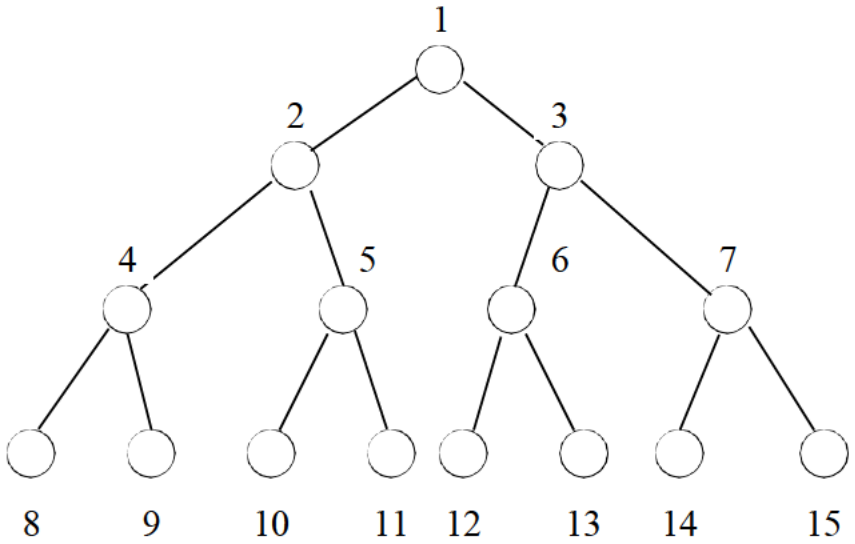
У цьому реалізації змінна ukaz є індексом (покажчиком) верхівки стека і водночас показує кількість елементів у стеку, тобто. визначає фактичний розмір масиву. Для того, щоб помістити елемент у стек, ми записуємо його в масив СТЕК за індексом ukaz та збільшуємо ukaz на одиницю. Для того, щоб взяти елемент зі стеку, процедура Take_Stack повинна зменшити значення ukaz на одиницю і повернути елемент з індексом ukaz. Таким чином, покажчик (індекс у масиві) розташований так, як показано на малюнку 4.26.

Суттєвим недоліком уявлення стека векторним способом є необхідність завдання максимальної глибини стека, оскільки не завжди відома точна глибина стека. Звідси необхідність завдання глибини стека із " запасом " , тобто. пам'ять використовується нераціонально. Гідність – відносна простота роботи зі стеком.

4.3.3 Подання двійкового дерева у вигляді масиву та реалізація алгоритму обходу двійкового дерева зліва.

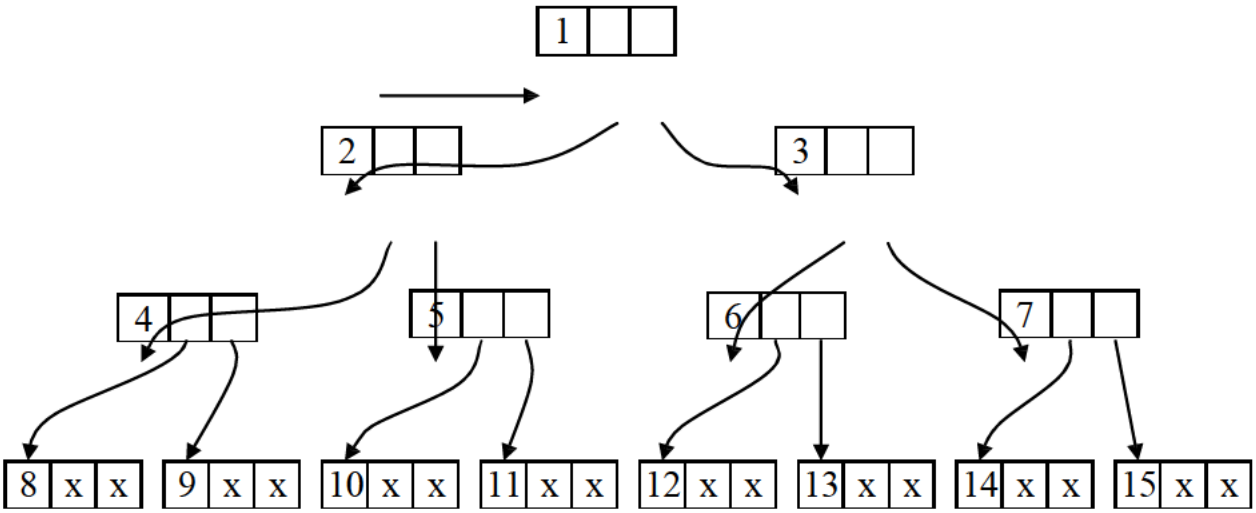
Дерева є найбільш важливими структурами, що використовуються в програмуванні. Тому необхідно освоїти техніку роботи з ними. Тут ми розглянемо подання дерева за допомогою масиву.

Нехай є дерево, рис. 4.27.



Мал. 4.27. Двійковий дерево. Як вершин виступають цілі числа

У цьому дереві вершинами є просто цілі числа, хоча як вершини можуть виступати будь-які об'єкти. Уявимо наше дерево у вигляді, рис. 4.28. Звідси виникає ідея зберігати у масиві як саму вершину, а й індекси лівого і правого поддерева. Таким чином, кожен елемент масиву є трійкою чисел – корінь та індекси в масиві лівого та правого піддерева. Якщо ліве чи праве піддерева порожні, то індекс дорівнює -1 (аналог nil).



Мал. 4.28. Схема представлення двійкового дерева за допомогою масиву

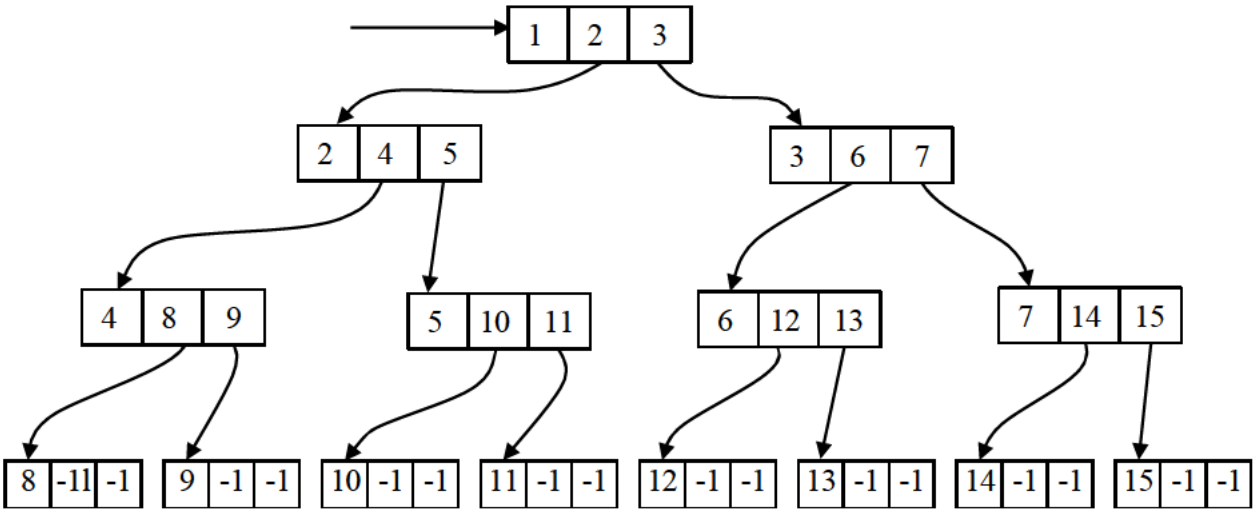
Уважно придивіться малюнку. Фактично це схоже на пов'язаний список, лише замість покажчиків ми використовуємо індекси масиву. Подання цього ж дерева за допомогою "справжнього" пов'язаного списку та указу-

телів ми розглянемо в наступному розділі. Для того щоб записати індекси на ліві та праві піддерева в масиві надійдемо таким чином: якщо вершина – корінь має номер індексу n, то ліве та праве піддерево – відповідно 2*n та 2*n+1. Нумерація починається з n=1. Осередок зі значенням -1 означає відсутність вершини. Спочатку зробимо вручну розподіл індексів у масиві для нашого дерева, табл. 4.1.

Таблиця 4.1

індекс	корінь	ліве піддерево (індекс у масиві)	праве піддерево (індекс у масиві)
1	1	2	3
2	2	4	5
3	3	6	7
4	4	8	9
5	5	10	11
6	6	12	13
7	7	14	15
8	8	-1	-1
9	9	-1	-1
10	10	-1	-1
11	11	-1	-1
12	12	-1	-1
13	13	-1	-1
14	14	-1	-1
15	15	-1	-1
16	-1	-1	-1

Візуально масив, що реалізує наше вихідне двійкове дерево, можна представити таким чином, рис. 4.29.



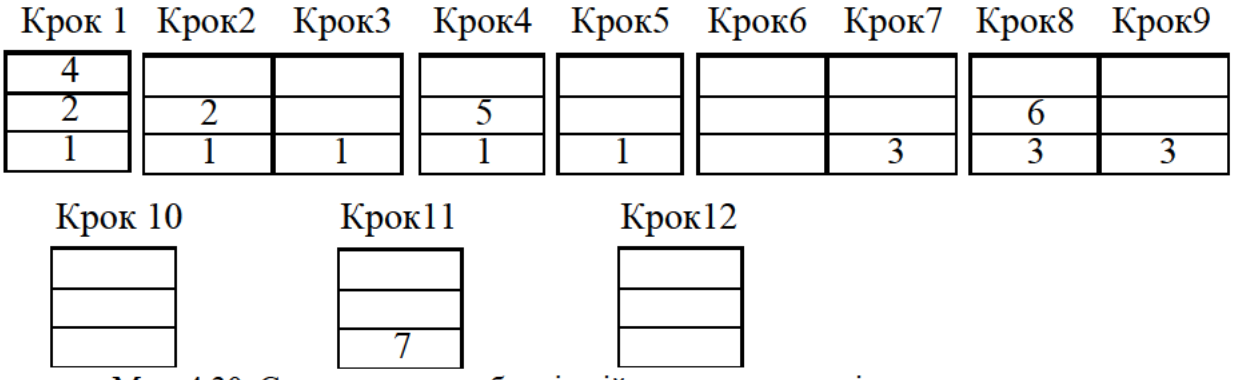
Мал. 4.29. Остаточна схема представлення двійкового дерева за допомогою масиву

При обході цього двійкового дерева ліворуч послідовність обробки вершин буде такою:

8, 4, 9, 2, 10, 5, 11, 1, 12, 6, 13, 3, 14, 7, 15

Блок схему алгоритму обходу ми вже розглядали, див. рис. 4.23.

Послідовність станів стека для даного алгоритму буде слідме, рис. 4.30:



Мал. 4.30. Стан стека при обході двійкового дерева зліва

Напишемо програму обходу двійкового дерева зліва. Спочатку налагодимо програму для вже готового дерева. Далі ми навчимося організовувати введення дерева у діалоговому режимі з клавіатури. Для цього підготуйте у текстовому

редакторі "Блокнот" (NotePad) файл подання двійкового дерева у вигляді масиву наведеного в таблиці 4.1 з ім'ям "derevo.dat" і розмістіть його в папці з проектом. Можете взяти готовий файл з DVD диска, що додається до книги. Відповідно до структури масиву використовуватимемо запис виду:

```
type
  T = record
  TElem: integer;
  TLeft: integer;
  TRight: integer;
  end;
```

Де T – елемент масиву типу запис, TElem вершина дерева, TLeft індекс у масиві лівого піддерева, TRight індекс у масиві правого піддерева.

Щоб не захаращувати текст основний програми, процедури Put_Stack та Take_Stack оформимо у вигляді окремого модуля STACK. Текст модуля:

```
unit STACK;
interface
const
  SizeOfArrayCTEK = 100;
  SizeOfArrayDER = 100;
type
  T = record
  TElem: integer;
  TLeft: integer;
  TRight: integer;
end;
```

```
procedure Put_Stack(var ELEM: integer;
                    var CTEK: array of T;
                    var ukaz: integer;
                    var ERROR: Boolean);

procedure Take_Stack(var ELEM: integer;
                     var CTEK: array of T;
                     var ukaz: integer;
                     var EMPTY: Boolean);
implementation
{ ======================================== }
procedure Put_Stack(var ELEM: integer;
                    var CTEK: array of T;
                    var ukaz: integer;
                    var ERROR: Boolean);
{ ======================================== }
begin
  if ukaz > SizeOfArrayCTEK then
      ERROR:= true
  else
  begin
    CTEK[ukaz].TElem:= ELEM;
    ukaz:= ukaz + 1;
  end;
end;
{ ======================================== }
procedure Take_Stack(var ELEM: integer;
                     var CTEK: array of T;
                     var ukaz: integer;
```

```
                      var EMPTY: Boolean);
{ ===================================== }
begin
  if ukaz = 1 then
    EMPTY:= true
  else
  begin
    ukaz:= ukaz - 1;
    ELEM:= CTEK[ukaz].TElem;
  end;
end;
end.
```

Лістинг програми обходу двійкового дерева зліва:

```
program Derevo_Left;
{$mode objfpc}{$H+}
{Процедури роботи зі стеком ми оформили у
вигляді модуля!} uses
  CRT, FileUtil, STACK;
var
  CTEK:array [1..SizeOfArrayCTEK] of T;
  DER:array[1..SizeOfArrayDER] of T;
  ukaz: integer;
  i ,ELEM: integer;
  ERROR, EMPTY: Boolean;
  fder: TextFile;
begin
 {Ініціалізація покажчика стека (початкового
  індексу в масиві)} ukaz: = 1;
```

```
{Читання з файлу дерева}
Assign(fder, 'derevo.dat');
Reset(fder);
i:= 1;
while not Eof(fder) do
begin
  Read (fder, DER [i]. TElem);
  Read(fder, DER[i].TLeft);
  Read (fder, DER [i].
  TRight); inc(i);
end;
Close(fder);ER
ROR: = false;
EMPTY: =
false; i:= 1;
writeln(UTF8ToConsole('Обхід двійкового дерева
зліва')); while DER[i].TElem <> -1 do
begin
  if DER[i].TLeft <> -1 then
  begin
    ELEM: = i;
    Put_Stack(ELEM, CTEK, ukaz, ERROR);
    if ERROR = true then
    begin
      writeln(UTF8ToConsole('Помилка! Переповнення
      стека.')); writeln(UTF8ToConsole('Збільшіть розмір
      масиву')); writeln(UTF8ToConsole('Натисніть будь-
      яку клавішу')); readkey;
      exit;
```

```
      end;
      i:= DER[i].TLeft;
    end
    else
    begin
      repeat
        writeln(DER[i].TElem);
        Take_Stack(ELEM,CTEK,ukaz,EMPTY);i
        f EMPTY = true then break;
        i:= ELEM;
      until DER [i]. TRight <> -
      1; if EMPTY = true then
      break;
      writeln(DER[i].TElem);
      i:= DER[i].TRight;
    end;
  end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Тепер напишемо програму, де введення дерева виконується з клавіатури.

```
program Derevo_Left;
{$mode objfpc}{$H+}
{Процедури роботи зі стеком ми оформили у
вигляді модуля!} uses
  CRT, FileUtil, STACK;
var
  CTEK: array [1..SizeOfArrayCTEK] of T;
  DER: array[1..SizeOfArrayDER] of T;
```

```
  ukaz: integer;
  i,ELEM:integer;
  ERROR, EMPTY: Boolean;
  k: integer;
  answ: char;
begin
  {Введення дерева}
  writeln(UTF8ToConsole('Вводите дерево строго'));
  writeln(UTF8ToConsole('зверху вниз і зліва
  направо')); writeln(UTF8ToConsole('Щоб закінчити
  введення введіть -1')); i:= 1;
  while true do
  begin
    writeln(UTF8ToConsole('Введіть корінь.'));
    {$i-} // відключення стандартного режиму контролю
    введення
    repeat
      ERROR: = false;
      readln(k);
      ERROR: = (IoResult <> 0);
      if ERROR then {якщо ERROR=true, значить сталася
      помилка при
введення}
      writeln(UTF8ToConsole('Помилка! Введіть номер
      вершини'));
    until not ERROR;
    {$+} // Відновлення стандартного режиму контролю
    введення/виводу
    DER [i]. TElem: =
    k; if k = -1 then
    begin
```

```
DER[i].TLeft:= -1;
DER [i]. TRight: = -1;
```

```
      break;
    end;
    writeln(UTF8ToConsole('Поточна вершина має піддерев'я?'));
    writeln(UTF8ToConsole('Відповідь (y/n)'));
    repeat
      readln(answ);
      if (answ = 'y') then
      begin
        DER [i]. TLeft: = 2 * i; DER [i].
        TRight: = 2 * i + 1;
      end
      else
      begin
        DER[i].TLeft:= -1;
        DER [i]. TRight: = -1;
      end;
    until (answ = 'n') або (answ = 'y');
    inc(i);
 end;
{Обхід заданого двійкового дерева зліва}
{Ініціалізація покажчика стека (початкового
 індексу в масиві)} ukaz: = 1;
 ERROR:      =
 false;
 EMPTY:=false;
 i:= 1;
 writeln(UTF8ToConsole('Обхід двійкового дерева
 зліва')); while DER[i].TElem <> -1 do
 begin
   if DER[i].TLeft <> -1 then
```

```
    begin
      ELEM: = i;
      Put_Stack(ELEM,CTEK,ukaz,ERROR);i
      f ERROR = true then
      begin
        writeln(UTF8ToConsole('Помилка! Переповнення
        стека.')); writeln(UTF8ToConsole('Збільшіть розмір
        масиву')); writeln(UTF8ToConsole('Натисніть будь-
        яку клавішу')); readkey; exit;
      end;
      i:= DER[i].TLeft;
    end
    else
    begin
      repeat
        writeln(DER[i].TElem);Take_Stack(ELE
        M, CTEK, ukaz, EMPTY); if EMPTY =
        true then break;
        i:= ELEM;
      until DER [i]. TRight <> -
      1; if EMPTY=true then
      break;
      writeln(DER[i].TElem);
      i:= DER[i].TRight;
    end;
  end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Намучились із введенням дерева? Справді, нудотне це заняття. На щастя

ністю, в реальних завданнях дерева ніхто і ніколи не вводить. Вони формуються програмно залежно від завдання. Приклад автоматичного створення дерева ми розглянемо у розділі сортування та пошук за допомогою двійкового дерева.

Тепер спробуйте самостійно реалізувати інші засоби обходу двійкового дерева, тобто. обхід зверху та знизу.

Переваги уявлення дерев як масивів – відносна простота реалізації. Для невеликих дерев, особливо якщо кількість вершин заздалегідь відома, використання масивів може виявитися більш ефективним рішенням. Але звідси витікає і недолік, якщо кількість вершин дерева заздалегідь невідома, то, як і у випадку зі стеком, доводиться резервувати пам'ять максимально. А який цей максимум? Певних критеріїв немає. Пам'ять витрачається непродуктивно. Ще один істотний недолік - порядок розташування піддерев'я в масиві не піддається формалізації.

Паскаль надає значно зручніший механізм для представлення списків і, отже, для реалізації стека та дерев. Розглянемо цей механізм.

4.3.4 Вказівники

Існує ряд завдань, де статичні структури даних є неефективними для реалізації алгоритму, тому в Паскалі передбачено можливість роботи з динамічними типами структур. З деякими з них, зокрема стеками та пов'язаними списками, ми познайомилися у попередніх розділах. Також ми побачили, що стек можна продати, використовуючи масиви.

Ефективним засобом побудови пов'язаних списків є покажчики.

Про покажчиків ми вже розмовляли розділ 3.2.1.6, де йшлося про тип даних – покажчик.

Нагадаємо синтаксис оголошення типізованих покажчиків:

```
type
   Pint = ^integer;
var
  p: ^integer; {покажчик на змінну цілого типу}
  p1: ^ string; {покажчик на рядок символів}
  p2: Pint;
```

Нетипізований покажчик описується так:

```
var
  ptr: pointer; {нетипізований покажчик}
```

Опис покажчиків p і p2 еквівалентні, тобто. можна застосовувати той та інший спосіб.

Вказівник фактично є адресою пам'яті, за якою здійснюється доступ до значень динамічної змінної.

Над покажчиками допустимі операції перевірки на рівність (=) та нерівність (<>), а також можливе виконання оператора присвоєння := . Якщо список порожній, то значення змінної-покажчика дорівнює nil.

Пам'ять для динамічних структур, що реалізуються за допомогою покажчиків, розподіляється в спеціальній області оперативної пам'яті комп'ютера і називається "купою" (від англ. heap – купа) або пам'яттю, що динамічно розподіляється (ДРП).

Для роботи з динамічними змінними у Паскалі передбачені процедури:

`New(p) – виділити в пам'яті (ДРП), що динамічно розподіляється, необхідну пам'ять для змінної заданого типу з типізованим покажчиком p;`

Dispose(p) – звільнити ділянку ДРП, зайняту змінною з типізованим покажчиком нар.

Getmem(p, size) – другий спосіб виділити пам'ять змінної із вказівником p розміром size байтів. Цією процедурою можна назвати пам'ять як для типизованих, так нетипізованих покажчиків.

Freemem(p, size)) – звільнити пам'ять розподіленої змінної із покажчиком p розміром size байтів.

Приклад:

```
New (p); {Виділяється пам'ять для змінної
цілого типу} New (p1); {Виділяється пам'ять
для рядка символів}
Getmem(p2, 1000); {Виділяється пам'ять розміром 1000
байт для перемінної з вказівником p2}
```

Як ви бачите, динамічні змінні не мають власного імені та доступ до значення такої змінної здійснюється через операцію розіменування покажчика. Для цього використовується той самий символ "^", що використовувався під час опису покажчиків.

Приклад:

```
p^:= 25;
p1^:=      'Це      рядок
символів'; p2^:= 44;
```

При виділенні пам'яті процедурою Getmem для типізованих покажчиків і визначенні розміру пам'яті, що виділяється, найкраще використовувати процедуру SizeOf так як, розміри пам'яті, що відводиться під той чи інший тип, можуть відрізнятися в різних версіях компіляторів, наприклад:

```
Getmem(p2, SizeOf(integer));
```

Обов'язково звільняйте пам'ять після використання! Майте на увазі

тільки, що після звільнення пам'яті значення покажчиків не зміняться, що може призвести до помилок. Необхідно надавати значенням покажчиків nil після звільнення пам'яті. Крім того, не можна змішувати методи виділення та звільнення пам'яті. Так, якщо пам'ять було виділено процедурою New(), то не можна звільняти пам'ять процедурою Freemem() і, навпаки, якщо пам'ять було виділено процедурою Getmem(), то не можна звільняти процедурою Dispose().

Тепер давайте збудуємо пов'язаний список, використовуючи покажчики. Для цього достатньо визначити тип даних – запис та покажчик на нього. Структура запису буде наступною:

```
type
  PMyList = ^TMyList;
  TMyList = record
  data: integer;
  next: PMyList;
  end;
var
  mylist: PMyList;
```

У полі data знаходиться змістовна частина елемента списку, причому data може бути, своєю чергою, записом досить складної структури. Поле next – покажчик. У ньому міститься посилання на наступний елемент списку. Завдяки цьому покажчику можна пересуватися до наступного елемента списку. Як ознака закінчення списку використовують порожній покажчик nil. Якщо необхідно рухатися за списком у прямому та зворотному напрямку, то до структури запису додають ще одне поле – покажчик, у якому вказують адресу попереднього елемента списку. Такі списки з двома покажчиками називаються двозв'язковими або двонаправленими.

Якщо кінці списку замість порожнього покажчика nil поставити посилання початку списку, тобто. на перший елемент списку, отримаємо так званий кільцевий список.

Для всіх видів списків можна визначити стандартні операції, які виконуються над списками і які має вміти реалізовувати програміст. До таких операцій належать додавання нового елемента до списку, видалення елемента зі списку, пошук потрібного елемента у списку. Розглянемо їх.

4.3.5 Стандартні операції з лінійними списками

Запишемо стандартні операції з лінійними списками у вигляді готових процедур та функцій.

Процедура додавання (вставки) нового елемента до списку:

```
procedure Insert_MyList(Elem: integer;
                        var pHead, pCurrent: PMyList);
{pHead – покажчик на перший елемент списку ("голову"
списку)
 pCurrent – покажчик на поточний елемент
списку} var
  p: PMyList; // Робочий покажчик
begin
  new(p); // Заводимо новий елемент
  p^.data:= Elem; // записуємо до нього дані
  {Якщо список був порожнім, то головою
  списку стає p} if pHead = nil then
  begin
    p^.next: = nil;
    pHead:= p;
  end
```

```
  else
  begin
    p^.next:= pCurrent^.next; { у новому елементі
    робимо посилання наступний елемент, який був до
    вставки} pCurrent^.next:= p; {у поточному елементі
    робимо
    посилання на новий}
  end;
  pCurrent: = p; // поточним стає новий елемент
end;
```

Функція пошуку елемента у списку просто проходить усі елементи списку за посиланнями від початку, доки не буде знайдений шуканий елемент.

```
function Search_MyList(Elem: integer;
                       var pHead: PMyList): boolean;
var
  p: PMyList;
begin
  if pHead <> nil then
    p:= pHead
  else
  begin
    writeln(UTF8ToConsole('Список
    порожній')); exit;
  end;
  Search_MyList:= false;
  while true do
  begin
    if p^.data = Elem then
```

```
    begin
      Search_MyList:=true;
      break;
    end
    else
    if p^.next = nil then break;
    p:= p^.next;
  end;
end;
```

Функція повертає true, якщо потрібний елемент знайдено.

Процедура видалення елемента зі списку:

```
procedure Delete_MyList(Elem: integer;
                        var pHead, pCurrent: PMyList);
var
  p: PMyList;
  pPrev: PMyList; // Попередній елемент списку
  find: boolean;
begin
   if pHead <> nil then
    p:= pHead
  else
  begin
    writeln(UTF8ToConsole('Список
    порожній')); exit;
  end;
  find:       =
  false;  while
  true do begin
```

```
  if p^.data = Elem then
  begin
    find: = true;
    break;
  end
  else
  if p^.next = nil then break;
  pPrev:= p;
  p:= p^.next;
end;
if find then
begin
  if p = pHead then // якщо видаляється перший елемент
  begin
    p:= pHead^.next; // запам'ятовуємо посилання
    наступний елемент
    Dispose(pHead); // видаляємо перший елемент списку
    pHead:= p;//       головою списку стає p end
  else
  begin // якщо видаляється не перший елемент
    pPrev^.next:= p^.next; { в попередньому елементі
    замінюємо посилання на наступний елемент після
    видалення }
    Dispose(p);
    pCurrent: = pPrev; // поточним робимо попередній
    елемент
  end;
  writeln(UTF8ToConsole('Елемент успішно
видалено')); end
else writeln(UTF8ToConsole('Елемент не
```

```
знайдено')); end;
```

Процедура, перш ніж видалити елемент, має знайти. Ця частина збігається з функцією пошуку. Оскільки код функції пошуку невеликий, ми вставили в процедуру сам код. У принципі, у процедурі можна викликати функцію пошуку. Для цього необхідно видозмінити функцію таким чином, щоб вона повертала посилання на знайдений елемент. Якщо елемента немає, вона має повертати nil. Надаємо змінити функцію пошуку самому читачеві.

Напишемо головну програму роботи зі списком (не забудьте включити до програми процедури та функції, які ми щойно розібрали):

```
program operation_list;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  PMyList = ^TMyList;
  TMyList = record
  data: integer;
  next: PMyList;
  end;
var
  pHead, pCurrent: PMyList;
  Elem, choose: integer;
{Сюди додайте процедури та функції, що
реалізують стандартні операції з
лінійними списками} begin
  pHead:= nil;
  pCurrent: = nil;
  repeat
```

```
writeln(UTF8ToConsole('Виберіть потрібну
дію:')); writeln(UTF8ToConsole('1-введення
елемента списку')); writeln(UTF8ToConsole('2-
пошук елемента списку'));
writeln(UTF8ToConsole('3-видалення елемента
списку')); writeln(UTF8ToConsole('4-перегляд
всього списку')); writeln(UTF8ToConsole('5-
вихід із програми')); readln(choose);
case choose of
1: begin {введення елемента списку}
     writeln(UTF8ToConsole('Введіть елемент
     списку')); readln(Elem);
     Insert_MyList(Elem, pHead, pCurrent);
   end;
2: begin {пошук елемента списку}
     writeln(UTF8ToConsole('введіть елемент, що
     шукається')); readln(Elem);
     if Search_MyList(Elem, pHead)then
       writeln(UTF8ToConsole('Елемент знайдено'))
     else
       writeln(UTF8ToConsole('Елемент не
   знайдено')); end;
3: begin {видалення елемента списку}
     writeln(UTF8ToConsole('введіть елемент'));
     readln(Elem);
     Delete_MyList(Elem, pHead, pCurrent);
   end;
4: begin {Виведення списку на екран}
     writeln(UTF8ToConsole('Елементи списку:'));
```

```
          writeln;
          view_MyList (pHead);
          writeln;
        end;
      end; {end of case}
  until choose = 5;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Крім того, часто зустрічаються особливі списки, де додавання або видалення елементів здійснюється тільки на початку або наприкінці списку. Які це списки:

- лінійний список, у якому додавання чи видалення елементів проводиться лише одному кінці. Читач, напевно, вже здогадався, що це не що інше, як стек!
- лінійний список, у якому додавання елементів здійснюється на одному кінці, а видалення на протилежному кінці списку. Якщо уважно подумати, то робота такого списку нагадуватиме живу чергу людей у магазині. Тому такий список так і називають черга. Часто чергу називають структурою FIFO (First In - First Out, тобто "перший прийшов, перший пішов").
- лінійний список, в якому всі додавання та видалення виробляються з обох кінців списку. Такий список зветься дек.

Реалізуємо розглянуті динамічні структури з допомогою лінійних списків.

4.3.6 Реалізація динамічних структур лінійними списками

4.3.6.1. Реалізація стеку

Для того, щоб помістити елемент у стек, достатньо написати наступний код:

```
New (p); // Резервуємо пам'ять нового елемента
p ^.data: = ELEM; //Записуємо дані
p^.next:= СТЕК;// встановлюємо посилання на поточну
вершину стека
СТЕК: = p; // Тепер новий елемент став
верхівкою стека Для того, щоб взяти елемент з
верхівки стека, необхідний код:

if СТЕК <> nil then // Якщо стек не порожній
  begin
    ELEM:= СТЕК^.data; // Взяти елемент
    p:= СТЕК^.next; // Покажчик на наступний елемент
    Dispose (СТЕК); // Видалити поточний елемент з
    вершини стека СТЕК: = p; //Попередній елемент
    зробити вершиною стека
```

Наведемо повний лістинг програми роботи зі стеком:

```
program stack;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  c = ^ MyStack; // покажчик
  MyStack = record // структура запису
```

```
  data: integer; // змістовна частина стеку
```

```
  next: c; // Посилання на наступний елемент
  end;
var
  choose: integer;
  EMPTY: boolean;
  CTEK: c; // покажчик, вказує на вершину стека
  ukaz: c; // Робочий покажчик
  ELEM: integer; // змістовний елемент стека
{ ================================================ }
procedure Put_Stack(var ELEM: integer; var CTEK: c);
{ ================================================ }

begin
  New(ukaz); // створюємо новий елемент стека
  ukaz^.data:= ELEM; // У полі data запису записуємо
  значення
  ukaz^.next:= CTEK; // Посилання на наступний елемент
  CTEK: = ukaz; // вершиною стека стає новий елемент
end;
{ ================================================ }
procedure Take_Stack(var ELEM: integer; var CTEK: c;
                     var EMPTY:Boolean);
{ ================================================ }
var ukaz: c;
begin
  if CTEK <> nil then
  begin
    ELEM:= CTEK^.data; // Беремо елемент із вершини
    стека ukaz:= CTEK^.next; // Покажчик на
    наступний елемент Dispose (CTEK); // Знищення
```

```
    поточної вершини
```

```
    CTEK: = ukaz; // вершиною стека стає наступний елемент
  end
  else EMPTY: = true;
end;
procedure View_Stack(var CTEK: c);
var ukaz: c;
begin
  writeln(UTF8ToConsole('Вміст стека'));
  ukaz:= CTEK; // Беремо вершину стека
  {У циклі проходимо всі елементи стека та
  друкуємо їх} while ukaz <> nil do
  begin
    write(ukaz^.data, '');
    ukaz:= ukaz^.next;
  end;
  writeln;
end;

begin
  {Ініціалізація стека}
  CTEK: = nil; repeat
    EMPTY: = false;
    writeln(UTF8ToConsole('Виберіть потрібний режим
    роботи:')); writeln(UTF8ToConsole('Помістити елемент у
    стек
                                                    1'));writel
    n(UTF8ToConsole('Взяти елемент із стеку
                                                    2'));writel
    n(UTF8ToConsole('Переглянути вміст стека 3'));
    writeln(UTF8ToConsole('Вихід із програми          4'));
```

```
    readln(choose);
```

```
    case choose of
    1: begin
         writeln(UTF8ToConsole('Введіть новий елемент
         стека')); readln(ELEM);
         Put_Stack(ELEM, CTEK);
       end;
    2: begin
         Take_Stack(ELEM, CTEK, EMPTY);
         if EMPTY then
         begin
           writeln(UTF8ToConsole('Стек
         порожній')); end
         else
         begin
           writeln(UTF8ToConsole('З стека взятий
                   елемент'), ELEM);
         end;
       end;
    3: begin
        View_Stack (CTEK);
      end;
    end; {end of case}
  until choose = 4;
end.
```

4.3.6.2. Реалізація черги за допомогою лінійного списку

Найпростіше для цього завести два вказівники. Вказівник на перший елемент черги first і вказівник на останній елемент черги last. Лістинг програми:

```
program queue; // queue означає англійською чергою
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  PQueue = ^Element; // покажчик
  Element = record // структура запису
  data: integer; // змістовна частина черги
  next: PQueue; // Посилання на наступний
  елемент end;
  TQueue = record // структура для реалізації черги
  first: PQueue; // покажчик, вказує перший елемент черги
  last: PQueue; // покажчик, що вказує на останній
  елемент черги end;

{ ============================================= }
procedure Put_Que(ELEM: integer; var Que: TQueue);
{ ============================================= }
var ukaz: PQueue; // тимчасовий покажчик
begin
  if (Que.last = nil) then
  {Якщо черга порожня, то
  покажчики на перший та останній
  елемент співпадатимуть} begin
    {Виділяємо пам'ять нового елемента}
    Que.last:= New(PQueue);
    Que.first:= Que.last; // Прирівнюємо покажчики
    Que.last^.data:= ELEM; // записуємо дані
```

```
    Que.last^.next:= nil; // Наступний елемент поки що
    порожній
  end
  else
  begin
    ukaz: = New (PQueue);
    ukaz^.data:= ELEM;
    ukaz^.next:= nil;
    Que.last^.next:= ukaz; // Посилання на наступний
    елемент
    Que.last:= ukaz; // Зрушуємо покажчик на останній
    елемент
  end;
end;

{ ================================================= }
procedure Take_Que(var ELEM: integer; var Que: TQueue;
                   var EMPTY:Boolean);
{ ================================================= }

var ukaz: PQueue;
begin
  if (Que.first = nil) then
  begin
    EMPTY: = true;
  end
  else
  begin
    ELEM:= Que.first^.data; {Перший елемент
    черги} ukaz: = Que.first; // Запам'ятаємо
    посилання
```

```
    Que.first:=   Que.first^.next;   {перехід на наступний
```
елемент}

```
    Dispose(ukaz); {знищення поточного
  елемента} end;
end;

{ ================================ }
procedure View_Que(var Que: TQueue);
{ ================================ }

var
  ukaz: PQueue;
begin
  writeln(UTF8ToConsole('Вміст черги')); ukaz:=
  Que.first; // Беремо перший елемент черги
  {У циклі проходимо всі елементи черги та
  друкуємо їх} while ukaz <> nil do
  begin
    write(ukaz^.data, '');
    ukaz:= ukaz^.next;
  end;
  writeln;
end;

var
  choose: integer;
  EMPTY: boolean;
  Que: TQueue;
  ELEM: integer; // змістовний елемент черги
begin
  {Ініціалізація черги}
```

```
Que.last:= nil;
repeat
  EMPTY: = false;
  writeln(UTF8ToConsole('Виберіть потрібний режим
  роботи:')); writeln(UTF8ToConsole('Помістити елемент у
  чергу
                                                    1'));writel
  n(UTF8ToConsole('Взяти елемент із черги
                                                    2'));writel
  n(UTF8ToConsole('Переглянути вміст черги 3'));
  writeln(UTF8ToConsole('Вихід із програми          4'));
  readln(choose);
  case choose of
  1: begin
       writeln(UTF8ToConsole('Введіть новий елемент'));
       readln(ELEM);
       Put_Que(ELEM, Que);
     end;
  2: begin
       Take_Que(ELEM, Que, EMPTY);
       if EMPTY then
       begin
         writeln(UTF8ToConsole('Черга
       порожня')); end
       else
       begin
         writeln(UTF8ToConsole('З черги взятий елемент'),
                 ELEM);
       end;
     end;
```

```
3: begin
     View_Que(Que);
```

```
      end;
    end; {end of case} until
  choose = 4;
end.
```

Реалізація дека тепер для вас не є таким складним завданням. Спробуйте продати його самостійно.

4.3.6.3. Реалізація двійкового дерева за допомогою лінійного списку

Двійкове дерево можна уявити за допомогою двозв'язкового списку таким чином:

```
type
  PTree = ^ Tree; // Покажчик на дерево
  Tree = record / /   Саме дерево має тип - запис
  node: string; // Значення вершини (вузла) дерева left:
  PTree; // Посилання на ліве піддерево right: PTree; //
  Посилання на праве поддерево end;
```

Тут для розмаїття значень вершин ми взяли рядки символів. Напишемо рекурсивну процедуру пошуку вершини за його значенням:

```
procedure search_node(Elem: string;
                      ptr_tree: PTree;
                      var current_tree: PTree);
var
  p: ^Tree; // Робочий покажчик
begin
```

```
  p: = ptr_tree;
  if not (p^.node = Elem) then
    begin
      if p^.left <> nil then
        search_node (Elem, p^.left, current_tree);
      if p^.right <> nil then
        search_node (Elem, p^.right, current_tree);
    end
  else current_tree: = p;
end;
```

У процедуру передаються значення шуканого вузла Elem, саме дерево ptr_tree. Процедура повертає посилання знайдений вузол current_tree.

Рекурсивна процедура видалення поточного піддерева:

```
procedure dispose_tree (ptr_tree: PTree);
var
  p: ^Tree;
begin
  if ptr_tree <> nil then
  begin
    p: = ptr_tree;
    if p^.left <> nil then
    begin
      dispose_tree(p^.left);
    end;
    if p^.right <> nil then
    begin
      dispose_tree(p^.right);
```

```
    end;
    dispose(p);
  end
end;
```

Рекурсивна процедура обходу двійкового дерева зліва запишеться на диво просто:

```
procedure obhod(p: PTree);
begin
   if p<>nil then
   begin
      obhod(p^.left);write(p
      ^.node, '');
      obhod(p^.right);
   end;
end;
```

Реалізуйте самостійно обхід двійкового дерева зліва за нерекурсивним алгоритмом 4.30!

Напишемо програму введення двійкового дерева з клавіатури та його обходу зліва:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  PTree = ^ Tree; // Покажчик на дерево
```

```
  Tree= record    // Само дерево, має тип –
                  запис
  node: string;   // значення вершини
                  (вузла) дерева
  left: PTree;    // Посилання на ліве
                  піддерево
  right: PTree; // Посилання на праве піддерево
  end;
var
  ptr_tree, current_tree, root: ^ Tree;
  p, current: ^Tree;
  s: string;
  choose: integer;
{Процедура пошуку вузла}
procedure search_node(Elem: string;
                      ptr_tree:PTree;
                      var current_tree:PTree);
var
  p: ^Tree;
begin
  p: = ptr_tree;
  writeln(p^.node);
  if not (p^.node = Elem) then
    begin
      if p^.left <> nil then
        search_node (Elem, p^.left, current_tree);
      if p^.right <> nil then
        search_node (Elem, p^.right, current_tree);
    end
  else current_tree: = p;
end;
```

```
{Процедура виведення дерева на екран}
```

```
procedure view_tree (ptr_tree: PTree);
var
  p: ^Tree;
begin
  p: = ptr_tree;
  writeln(p^.node);
  if p^.left <> nil then
    view_tree(p^.left);
  if p^.right <> nil then
    view_tree(p^.right);
end;

{Процедура видалення поточного
поддерева} procedure dispose_tree
(ptr_tree:PTree); var
  p: ^Tree;
begin
  if ptr_tree <> nil then
  begin
    p: = ptr_tree;
    writeln(p^.node);
    if p^.left <> nil then
    begin
      dispose_tree(p^.left);
    end;
    if p^.right <> nil then
    begin
      dispose_tree(p^.right);
    end;
```

```
    dispose(p);
  end
end;
procedure obhod(p: PTree);
begin
   if p <> nil then
   begin
      obhod(p^.left);write(p
      ^.node, '');
      obhod(p^.right);
   end;
end;
begin
  writeln(UTF8ToConsole('введіть номер або ім'я вершини'));
  readln(s);
  new
  (current);root: =
  current;
  current^.node:= s;
  current^.left:= nil;
  current^.right:= nil;
  repeat
    writeln;
    writeln(UTF8ToConsole('Корінь поточного
            дерева: '), current^.node);
    writeln(UTF8ToConsole('Виберіть потрібну дію:'));
    writeln(UTF8ToConsole('1-введення лівого
    піддерева')); writeln(UTF8ToConsole('2-введення
    правого піддерева')); writeln(UTF8ToConsole('3-
    зробити корінь піддерева поточним'));
```

```
    writeln(UTF8ToConsole('4-переглянути дерево'));
```

```
writeln(UTF8ToConsole('5-видалити поточне
піддерево')); writeln(UTF8ToConsole('6-обхід
дерева зліва')); writeln(UTF8ToConsole('7-вихід
із програми')); readln(choose);
case choose of
1: begin {Створення лівого
     піддерева} if
     current^.left= nil then
       new(p)
     else
       p:= current^.left;
     writeln(UTF8ToConsole('введіть номер чи ім'я вершини'));
     readln(s);
     p^.node: = s;
     p^.left:= nil;
     p^.right:= nil;
     current^.left:= p;
   end;
2: begin {Створення правого
     поддерева} if
     current^.right= nil then
       new(p)
     else
       p:= current^.right;
     writeln(UTF8ToConsole('введіть номер чи ім'я вершини'));
     readln(s);
     p^.node: = s;
     p^.left:= nil;
     p^.right:= nil;
     current^.right:= p;
```

```
        end;
```

```
3: begin {Пошук потрібної вершини}
     writeln(UTF8ToConsole(') введіть номер чи ім'я вершини'));
     readln(s);
     current_tree:= nil;
     ptr_tree: = root;
     search_node (s, ptr_tree, current_tree);
     if current_tree <> nil then
       current: = current_tree;
   end;
4: begin {Виведення введеного дерева
     на екран} ptr_tree:= root;
     view_tree (ptr_tree);
   end;
5: begin {Видалення піддерева}
     writeln(UTF8ToConsole('введіть букву L для'));
     writeln(UTF8ToConsole('видалення лівого
     піддерева')); writeln(UTF8ToConsole('або будь-
     який символ для'));
     writeln(UTF8ToConsole('видалення правого
     піддерева')); readln(s);
     if (s= 'l') or (s= 'L') then
     begin {Видалення лівого
     піддерева}
       ptr_tree:= current^.left;
       current^.left:= nil;
       dispose_tree(ptr_tree);
     end
     else
     begin {Видалення правого
       піддерева} ptr_tree:=
```

```
current^.right;
current^.right:= nil;
```

```
            dispose_tree(ptr_tree);
          end;
        end;
     6: begin
          writeln(UTF8ToConsole('Обхід двійкового дерева зліва:'));
          writeln;
          obhod(root);
          writeln;
        end;
      end; { end of case
  } until choose = 7
end.
```

4.3.7 Сортування та пошук за допомогою двійкового дерева

Двійкові дерева найчастіше застосовуються для сортування та пошуку. Нехай для певності сортується масив, що складається з цілих чисел. Для цього спочатку будується двійкове дерево за таким алгоритмом: як корінь двійкового дерева береться перший елемент масиву. Наступні елементи масиву стають або лівими або правими нащадками залежно від значення елемента. Якщо наступний елемент менше кореня, то він вставляється в ліве піддерево, якщо більше кореня, то праве піддерево, причому вставляється в потрібне місце в залежності від значення поточного елемента. Після побудови двійкового дерева здійснюється його обхід ліворуч. Легко бачити, що якщо двійкове дерево побудовано так, як описано вище, то при його обході зліва ми отримаємо впорядкований за зростанням масив.

Такі двійкові дерева називаються двійковими деревами пошуку або дерева бінарного пошуку (binary search tree), оскільки під час пошуку використовується впорядкованість двійкового дерева. Пошук відбувається наступним

чином. Як поточний вузл приймаємо корінь. Потім порівнюємо значення шуканого елемента зі значенням поточного вузла. Якщо вони рівні, то необхідний елемент знайдено та алгоритм закінчує свою роботу. В іншому випадку, якщо елемент, що шукається, менший від значення поточного вузла, то поточним робимо ліве піддерево, якщо більше, то поточним стає праве піддерево. Знову порівнюємо наш елемент з поточним вузлом і так далі доти, доки шуканий елемент не буде знайдений, або не буде досягнутий порожній вузол, що означатиме, що елемента в масиві немає.

Вище ми вже зазначали, що дерева зазвичай не запроваджуються, а формуються програмно. Напишемо процедуру формування двійкового дерева за заданим масивом цілих чисел:

```
procedure insert_node(Elem: integer; var root: PTree);
var
  p: ^Tree; // поточний вузол
  newTree: ^Tree; // новий вузол
begin
  p:= root; // Починаємо з кореня і проходимо до потрібного
  вузла
  while (Elem > p^.node) and (p^.right <> nil) or
        (Elem < p^.node) and (p^.left <> nil) do
  if Elem < p^.node then
    p:= p^.left
  else
    p:= p^.right;
  New (newTree); {Створення нового
  сайту} newTree^.left:= nil;
  newTree^.right:= nil;
  newTree^.node:= Elem;
  {Залежно від значення Elem новий
```

вузол додається або праворуч, або зліва} if
Elem > p^.node then
p^.right:= newTree
else
p^.left:= newTree;
end;

Процедуру пошуку ми вже розробляли в останньому прикладі попереднього розділу. На цей раз оформимо пошук у вигляді функції:

```
function Search_Elem(Elem: integer;
                     var root: PTree): boolean;
var
  p: PTree;
begin
  Search_Elem:= false;
  if root = nil then exit;
  p:= root;
  if p^.node = Elem then
    Search_Elem:= true // елемент знайдений
  else
    if Elem < p^.node then
       Search_Elem:= Search_Elem(Elem, p^.left)
    else
       Search_Elem:= Search_Elem(Elem, p^.right);
end;
```

Тепер давайте напишемо програму сортування масиву цілих чисел за зростанням та пошуком у дереві бінарного пошуку потрібного елемента. Все потрібне

ні процедури у нас вже є.

```
program in_order;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  vector = array of integer; PTree
  = ^ Tree; // Покажчик на дерево
  Tree = record / /  Саме дерево має тип - запис
  node: integer; // Значення вершини (вузла) дерева left:
  PTree; // Посилання на ліве піддерево
  right: PTree; // Посилання на праве піддерево
  end;
var
  i, n, choose: integer;
  Elem: integer;
  sorted_array: vector; // сортований масив
  root: PTree;
{Процедура обходу двійкового
дерева зліва} procedure obhod(p:
PTree);
begin
   if p <> nil then
   begin
      obhod(p^.left);write(p
      ^.node, '');
      obhod(p^.right);
   end;
end;
```

```
{Процедура пошуку вузла для вставлення нового вузла}
procedure insert_node(Elem: integer; var root: PTree);
var
  p: ^Tree; // поточний вузол
  newTree: ^Tree; // новий вузол
begin
  p:= root; // Починаємо з кореня і проходимо до потрібного
  вузла
  while (Elem > p^.node) and (p^.right <> nil) or
        (Elem < p^.node) and (p^.left <> nil) do
  if Elem < p^.node then
    p:= p^.left
  else
    p:= p^.right;
  {Створення нового вузла}
  New (newTree);
  newTree^.left:= nil;
  newTree^.right:= nil;
  newTree^.node:= Elem;
  {Залежно від значення Elem
 новий вузол додається або
 праворуч, або ліворуч}
  if Elem > p^.node then
    p^.right:= newTree
  else
    p^.left:= newTree;
end;
  { ========= Сортування двійковим деревом пошуку
======== } procedure Tree_Sort(var sorted_array: vector);
var
```

```
  Elem: integer;
```

```
begin
  Elem: = sorted_array [0];
  New (root);
  root^.left:= nil;
  root^.right:= nil;
  root^.node:= Elem;
  for i:= 1 до High(sorted_array) do
  begin
    Elem: = sorted_array
    [i]; insert_node(Elem,
    root);
  end;
  obhod(root); // Обхід отриманого дерева зліва
end;

{Функція пошуку у бінарному дереві}
function Search_Elem(Elem: integer;
                     var root: PTree): boolean;
var
  p: PTree;
begin
  Search_Elem:= false;
  if root = nil then exit;
  p:= root;
  if p^.node = Elem then
    Search_Elem:= true // елемент знайдений
  else
    if Elem < p^.node then
       Search_Elem:= Search_Elem(Elem, p^.left)
    else
```

```
        Search_Elem:= Search_Elem(Elem, p^.right);
end;
begin
  writeln(UTF8ToConsole('Введіть кількість елементів
  масиву')); readln(n);
  SetLength(sorted_array, n);
  writeln(UTF8ToConsole('Введіть елементи
  масиву')); for i:= 0 to n - 1 do
    read(sorted_array[i]);
  repeat
    writeln(UTF8ToConsole('Виберіть потрібну
    дію:')); writeln(UTF8ToConsole('1-сортування
    масиву')); writeln(UTF8ToConsole('2-пошук
    елемента масиву')); writeln(UTF8ToConsole('3-
    вихід із програми')); readln(choose);
    case choose of
    1: begin {Сортування}
     {Виклик процедури сортування масиву бінарним деревом
         пошуку} Tree_Sort(sorted_array); writeln;
       end;
    2: begin {пошук}
         writeln(UTF8ToConsole('введіть потрібний
         елемент')); readln(Elem);
         if Search_Elem(Elem, root) then
           writeln(UTF8ToConsole('Елемент знайдено'))
         else
           writeln(UTF8ToConsole('Елемент не
       знайдено')); end;
```

```
    end; {end of case}
  until choose = 3;
end.
```

Під час роботи з програмою зверніть увагу, що, перш ніж викликати функцію пошуку, необхідно відсортувати масив, тобто. побудувати відповідне двійкове дерево.

Глава 5 Основи об'єктно-орієнтованого програмування

5.1. Три джерела та три складові ООП.

Абревіатура ООП розшифровується як об'єктно-орієнтоване програмування. Розглянемо три джерела і три складові марксизму-ленінізму, ой вибачте (куди це мене занесло?!), звичайно ж, ОВП!

Три джерела – це об'єкти, абстракція та класифікація.

Основна ідея ООП полягає в об'єднанні даних, з якими працює програма та процедур, які ці дані обробляють в єдине ціле – об'єкт. Така організація програми дозволила максимально наблизити до природного сприйняття людиною навколишніх предметів, сутностей і понять. Адже людина сприймає навколишній світ, предмети та явища в сукупності властивостей, складових їх елементів та їх поведінки.

Програміст при вирішенні задачі з будь-якої предметної області виділяє окремі об'єкти, виходячи з особливостей задачі. Цей процес називається об'єктною декомпозицією [5]. Об'єкти складаються з даних, що описують властивості цих об'єктів та процедур, що опрацьовують ці дані. Наприклад, під час створення, скажімо, бази даних студентів деякого університету, можна виділити об'єкт "Студент". p align="justify"> Даними (властивостями) для цього об'єкта можуть виступати прізвище та ім'я студента, курс, група, його оцінки і т.д. При цьому можна визначити деякі процедури для обробки цих даних, наприклад, процедуру обчислення середньої оцінки за семестр (цю процедуру можна надалі використовувати для встановлення розміру стипендії), процедуру переведення з курсу на курс, процедуру відрахування (на жаль) з університету тощо.

При розгляді об'єктів дуже важливим є рівень абстракції (або рівень деталізації) з якою ми розглядаємо об'єкт. У нашому випадку нас зовсім не цікавлять такі характеристики об'єкта як зростання, колір очей, розмір взуття тощо. Ми абстрагуємося від цих властивостей і виділяємо ті властивості, які дозволяють нам вирішувати поставлене завдання.

У той же час важливо розуміти, що конкретний "примірник" студента, наприклад, на прізвище Іванов є представником цілого класу студентів. Цю класифікацію можна продовжувати і розвивати як кажуть "в різні боки". Взагалі студент, студент будь-якої групи, студент якогось ВНЗ. Отже, ми можемо ввести до розгляду об'єкт "Група", об'єкт "ВНЗ". Нарешті, студент без жодного сумніву є людиною!! Звідси ми можемо говорити про такий об'єкт, як "Людина".

Три складові – це інкапсуляція, успадкування і поліморфізм.

Об'єднання даних і їх функцій і процедур у вигляді окремих об'єктів називається інкапсуляцією. Основна складність ООП полягає в мистецтві виділити об'єкти, використовуючи абстрагування та класифікацію, які якомога точніше описували вирішуване завдання і, крім того, дозволяли б їх повторне використання. Внутрішній "пристрій" об'єкта може бути досить складним, але він "прихований від чужих очей". Для зв'язку із "зовнішнім світом" використовуються лише невеликі обсяги даних, причому кількість і тип цих даних суворо під контролем. Це суттєво підвищує надійність програми.

Спадкування одна з найважливіших властивостей ООП. Створення нових об'єктів шляхом використання вже існуючих дає програмісту низку переваг:• Не потрібно повторно розробляти код. Весь код для наявних об'єктів автоматично може бути використаний для нових об'єктів;

• Імовірність помилок різко знижується. Якщо код для вже існуючих об'єктів був уже налагоджений і перевірений, то будь-які помилки слід шукати в кодах, які були додані для нових об'єктів і, навпаки, якщо

знайдено та виправлено помилки в кодах для вихідних об'єктів, то вони будуть автоматично виправлені та в кодах для нових об'єктів, створених шляхом наслідування;

- Код програми стає зрозумілішим. Для того, щоб зрозуміти, як працюють об'єкти, створені шляхом успадкування, достатньо зрозуміти, як працюють додані дані та функції.

Під поліморфізмом розуміється можливість одних і тих же функцій або процедур по-різному обробляти дані, що належать різним об'єктам. Наприклад, нехай у вас є об'єкт "Геометрична фігура" і нехай у неї є функція обчислення площі. Якщо необхідно обчислити площу прямокутника, то функція обчислюватиме її за одним алгоритмом, а якщо це трикутник, то її площу функція обчислюватиме за зовсім іншим алгоритмом.

5.2. Класи та об'єкти.

Основним поняттям ОВП є клас. Клас являє собою структуру, що складається з опису полів даних, функцій та процедур, що обробляють поля даних та властивостей. Функції та процедури класу прийнято називати методами. Поля, методи та властивості називаються елементами чи членами класу.

Поле - це звичайна змінна мови Паскаль. Дозволяються будь-які типи змінних у мові, у тому числі поле може мати тип класу. З допомогою полів описуються стан об'єкта. За допомогою методів – поведінка об'єкта. Властивість це специфічний спосіб організації зв'язку з даними класу.

Таким чином, клас описує стан та поведінку об'єкта. Під об'єктом розуміється конкретний екземпляр (сутність) класу. У програмі об'єк-

ект описується як змінна типу клас. У мові Free Pascal як і більшості сучасних об'єктно-орієнтованих мов програмування змінна типу клас містить не самі дані, а посилання ними, тобто. є вказівником, а сам об'єкт розміщується у динамічній пам'яті. Однак операція розіменування для класових змінних у Free Pascal не застосовується. Перед використанням об'єкта необхідно виділити для нього пам'ять шляхом створення нового екземпляра класу або присвоєнням існуючого екземпляра змінної типу клас. Після завершення роботи з цим екземпляром класу (об'єктом) пам'ять необхідно звільнити. Оголошення класу в найпростішому випадку провадиться так:

```
type
  <ім'я класу> = class
  <оголошення полів класу>
  <Оголошення методів
класу> end;
```

Розглянемо приклад оголошення класу:

```
type
  THuman = class      // оголошення класу
  name: string; // поля класу
  fam: string;
  function GetData: string; // оголошення методу класу
end;
```

Тут THuman назва класу. Імена класів прийнято починати з літери T (від слова Type), це й необов'язково. Полями класу є name і fam,

а GetData – метод класу (у разі це функція).

Реалізація методу (тіло функції або процедури) міститься після оголошення класу (ключового слова end). Якщо клас створюється всередині модуля (найчастіше саме так і відбувається), то реалізацію методу слід поміщати після ключового слова implementation. У цьому перед ім'ям методу вказується ім'я класу через точку:

```
function THuman.GetData: string; // Реалізація методу
begin
  Result:= name + '' + fam;
end;
```

В даному випадку ми створили клас THuman (людина) з полями name (ім'я), fam (прізвище) та методом GetData, який просто повертає ім'я та прізвище людини.

Зверніть увагу, що всі поля класу доступні методам класу і їх не треба передавати як формальні параметри. Якщо ви вкажете імена полів класу як формальні параметри методу класу, наприклад таким чином

```
function GetData(name, fam: string): string;
```

компілятор видасть помилку. Так само в тілі методу не можна описувати змінні полів класу, наприклад таким чином:

```
function GetData: string;
var
  name: string;
```

5.2.1 Звернення до членів класу.

Після того як клас оголошений необхідно створити екземпляр класу шляхом оголошення змінної типу клас:

```
var
  Person: THuman;
```

Після опису змінної можна звертатися до членів класу. Звернення проводиться аналогічно до полів запису, тобто. через точку, наприклад:

```
Person.name:= 'Віталій';
Person.fam:= 'Петрів';
fname: = Person.GetData;
```

або за допомогою оператора with:

```
with Person do
begin
  name:= 'Олексій';
  fam: = 'Морозів';
  fname: = GetData;
end;
```

Напишемо програму роботи з щойно створеним класом:

```
program project1;
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil;
type

  THuman = class      // оголошення класу
      name: string; // поля класу
      fam: string;
      function GetData: string; // оголошення методу класу
  end;

function THuman.GetData: string; // Реалізація методу
begin
  Result:= name + '' + fam;
end;

var
  Person: THuman; // оголошення змінної типу клас
  fname: string;

begin
  Person:= THuman.Create; // Створення об'єкта
  Person.name:= 'Віталій'; // Присвоєння значень
  поля Person.fam: = 'Петрів';
  fname: = Person.GetData; // виклик методу
  writeln(UTF8ToConsole('Це:' + fname));
  with Person do // інший варіант звернення до членів класу
  begin
    name:= 'Олексій';
```

```
    fam: = 'Морозів';
    fname: = GetData;
  end;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Person.Free; // Звільнення пам'яті (знищення об'єкта)
end.
```

У цій програмі нами залишився не розглянутим оператор

```
Person:= THuman.Create;
```

З допомогою цього оператора на момент виконання програми об'єкт реально створюється (конструюється), тобто. об'єкт розміщується в пам'яті, поля об'єкта ініціалізуються необхідними значеннями, вказаними у спеціальному методі, який називається конструктором (про конструкторів ми поговоримо пізніше), адреса цього об'єкта заноситься в Person (не забувайте, що це покажчик!).

Наприкінці програми пам'ять, виділена об'єкту, звільняється, тобто. об'-єкт знищується.

У програмі можна використовувати скільки завгодно екземплярів класу. У такому прикладі створюється масив об'єктів. Імена та прізвища вводяться з клавіатури.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
```

```
  THuman = class
      name: string;
      fam: string;
      function GetData: string;
      procedure SetData(var f_name, s_name: string);
  end;

function THuman.GetData: string;
begin
  Result:= name + '' + fam;
end;

procedure THuman.SetData(var f_name, s_name: string);
begin
  name: = f_name;
  fam: = s_name;
end;

function GetName(var f_name, s_name: string): boolean;
begin
  Result: = true;
  writeln(UTF8ToConsole('Введіть ім'я'));
  readln(f_name);
  if f_name = '***' then
  begin
    Result: =
    false; exit;
  end;
  writeln(UTF8ToConsole('Введіть прізвище'));
```

```
  readln(s_name);
end;
const kolHuman = 25;
var
  Person: array [1..kolHuman] of THuman;
  i, kolperson: integer;
  fname, sname: string;
begin
  kolperson: = 0;
  for i:= 1 to kolHuman do
  begin
    якщо немає GetName(fname, sname) then break;
    Person[i]:= THuman.Create;
    Person[i].SetData(fname, sname);
    inc(kolperson);
  end;
  for i:= 1 to kolperson do
  begin
    writeln(UTF8ToConsole('Це:'), Person[i].GetData);
    Person[i].Free;
  end;
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Функція GetName не є членом класу. Це звичайна функція, яка організує введення даних. Як параметри передаються рядкові змінні, в які вводяться ім'я та прізвище. Після введення потрібних даних створюється новий об'єкт. До класу THuman додано новий метод

SetData за допомогою якого введені дані заносяться у поля класу. Ознакою закінчення введення служить рядок '***'. Отримавши такий рядок з клавіатури, функція GetName "сигналізує" програмі, що викликає, про те, що введення закінчено. При цьому новий об'єкт, природно, створюватись не буде. Змінна kolperson це лічильник, який відстежує кількість реально створених об'єктів.

5.3. Інкапсуляція

Вище ми зазначали, що під інкапсуляцією розуміється об'єднання даних та логіки їхньої обробки в одне ціле – об'єкт. Крім цього, інкапсуляція передбачає приховування внутрішньої структури та обмеження доступу. Зв'язок із " зовнішнім світом " здійснюється через так званий інтерфейс класу, тобто. набором правил, які регламентують доступом до членам класу. У певному сенсі з класом можна асоціювати якийсь "чорний ящик". Нам відомо, що він ("чорний ящик") може робити. Керувати ним ми можемо лише через його інтерфейс. А як він це робить нам невідомо.

Так само нам важливо знати що робить і вміє робити клас. А як він це робить нас, взагалі кажучи, не має позичати. Цим має займатися програміст – розробник класу. Також нам важливо, як цим класом користуватися (який в цього класу інтерфейс), тобто. які у цього класу є доступні властивості та методи.

Візьмемо, наприклад, телевізор. Що відбувається у нього всередині, коли ми його включаємо відомо "одному аллаху", але нам це не потрібно знати. Можливо, комусь цікаво як влаштований телевізор, але для більшості людей важливо те, що телевізор має відповідні кнопки або пульт з кнопками (інтерфейс) за допомогою яких ми можемо дивитися футбол або хокей (ну, або кому що подобається дивитися).

Принцип приховування інформації – одне із найважливіших принципів ООП. Приховуючи внутрішній устрій класу, ми абстрагуємося від непотрібних деталей. Крім того, це дозволяє захистити членів класу від несанкціонованого доступу ззовні.

У першій програмі розділу 5.2.1. ми зверталися до членів класу безпосередньо, наприклад оператором

```
Person.name:= 'Віталій';
```

Загалом кажучи, пряме звернення до полів не рекомендується. Це може стати джерелом помилок. Напишемо для ілюстрації наступний приклад:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type

  THuman = class
      name: string;
      fam: string;
      birthday: integer;
      function GetData: string;
  end;

function THuman.GetData: string;
var
  s: string;
begin
```

```
  str (birthday, s);
  Result:= name + ' + fam + ' ' + s; end;

var
  Person: THuman;

begin
  Person:= THuman.Create;
  writeln(UTF8ToConsole('Введіть ім'я'));
  readln(Person.name);
  writeln(UTF8ToConsole('Введіть прізвище'));
  readln(Person.fam);
  writeln(UTF8ToConsole('Введіть рік
  народження')); readln(Person.birthday);
  writeln(UTF8ToConsole('Це:'), Person.GetData);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Person.Free;
end.
```

До класу THuman ми додали нове поле birthday (рік народження). Крім того, дещо видозмінили метод GetData. Оскільки ця функція повертає рядок, необхідно вміст поля birthday, що має тип integer, перетворити на рядок. Тепер припустимо, що при введенні року народження було введено неприпустимий символ. Ваша програма аварійно завершиться. Тому для зміни значення поля краще використовувати метод (процедуру або функцію). У другій програмі розділу 5.2.1. з масивом об'єктів ми власне так і вчинили. У методі ви можете організувати контроль над

введеними даними. Перепишемо попередню програму таким чином:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type

  THuman = class
      name: string;
      fam: string;
      birthday: integer;
      function GetData: string;
      procedure SetData(var f_name, s_name,
                        b_year: string);
  end;

function THuman.GetData: string;
var
  s: string;
begin
  str (birthday, s);
  Result:= name + ' + fam + ' ' + s; end;

procedure THuman.SetData(var f_name, s_name,
                         b_year: string);
var
```

```
  code, year: integer;
begin
  name: = f_name;
  fam: = s_name;
  val(b_year, year, code);
  if code = 0 then
    birthday:= year;
end;

var
  Person: THuman;
  fname, sname, year: string;
begin
  Person:= THuman.Create;
  writeln(UTF8ToConsole('Введіть ім'я'));
  readln(fname);
  writeln(UTF8ToConsole('Введіть прізвище'));
  readln(sname);
  writeln(UTF8ToConsole('Введіть рік
  народження')); readln(year);
  Person.SetData(fname, sname, year);
  writeln(UTF8ToConsole('Це:'), Person.GetData);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Person.Free;
end.
```

Тут ми вводимо дані до деяких допоміжних змінних. Лише після закінчення введення методом SetData записуємо значення поля об'єкта

Person. Причому SetData здійснюємо контроль значення року народження шляхом перетворення рядка в число функцією val. Як вам уже відомо, ця функція повертає ненульове значення у параметрі code, якщо рядок не може бути перетворений на число.

Однак, у більшості випадків, доцільно взагалі заборонити доступ до деяких членів класу ззовні. Для цього застосовуються специфікатори доступу.

5.3.1 Специфікатори доступу.

Для реалізації інкапсуляції є такі специфікатори (директиви), які управляють видимістю (доступністю) членів класу:

- private (приватний, кажуть ще приватний) – поля та методи класу недоступні з інших модулів. Це дозволяє повністю приховати всю "кухню" реалізації класу. Однак вони доступні в межах модуля, де описаний цей клас. Більше того, якщо в одному модулі визначено кілька класів, то вони бачать приватні розділи один одного. Це зроблено для зручності розробника даного класу (класів) у цьому модулі. Погодьтеся, безглуздо обмежувати в доступі до "нутрощів" телевізора самого виробника.
- protected (захищений) – поля та методи класу мають обмежену видимість. Вони видно в самому класі, у всіх класах спадкоємців цього класу (в тому числі і в інших модулях) і в програмному коді, розташованому в тому ж модулі, що і клас.
- public (публічний) – вільно доступні з будь-якого місця програми, у тому числі з інших модулів.

Зазвичай, поля класу оголошуються як private, а методи – public. Хоча ті методи, які потрібні тільки для внутрішнього використання, можна помістити в розділ private або protected.

Напишемо таку програму:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type

  THuman = class
    private
      name: string;
      fam: string;
    public
      function GetData: string;
  end;

function THuman.GetData: string;
begin
  Result:= name + '' + fam;
end;

var
  Person: THuman;
  fname: string;

begin
  Person:= THuman.Create;
  Person.name:=
  'Віталій'; Person.fam:=
  'Петрів';
```

```
  fname: = Person.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Person.Free;
end.
```

Ця програма відрізняється від першої програми розділу 5.2.1. тим, що ми ввели в опис класу специфікатори доступу. Тут поля name і fam, як і раніше, залишаться доступними програмі, незважаючи на те, що вони поміщені в private.

Тепер створіть модуль (меню Файл->Створити модуль) і перемістіть опис класу у створений модуль (розділ interface), а реалізацію методу GetData до розділу implementation. Решту програми залиште без змін. Зверніть увагу, Lazarus автоматично додав до оголошення uses ім'я новоствореного модуля.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type

  THuman = class
    private
      name: string;
      fam: string;
    public
```

```
      function GetData: string;
  end;

implementation

function THuman.GetData: string;
begin
  Result:= name + '' + fam;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname: string;

begin
  Person:= THuman.Create;
  Person.name:= 'Віталій';
  Person.fam:= 'Петрів';
  fname: = Person.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Person.Free;
end.
```

У разі спроби компіляції компілятор видасть помилку на операторах:

```
Person.name:= 'Віталій';
Person.fam:= 'Петрів';
```

Для запису значень у поля name і fam необхідно створити метод та помістити його оголошення у розділ public.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  {THuman}
  THuman = class
    private
      name: string;
      fam: string;
    public
      function GetData: string;
      procedure SetData(const f_name, s_name: string);
  end;
implementation

function THuman.GetData: string;
begin
  Result:= name + '' + fam;
end;
```

```
procedure THuman.SetData(const f_name, s_name: string);
begin
  name: = f_name;
  fam: = s_name;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname: string;
begin
  Person:= THuman.Create;
  Person.SetData('Віталій', 'Петрів');
  fname: = Person.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Person.Free;
end.
```

Під час створення класів використовуйте функцію "Автозавершення коду". Після того, як ви написали оголошення методу, натисніть Ctrl+Shift+C та редактор вихідного коду Lazarus автоматично сформує заготівлю тіла вашого методу. Наприклад, в описі класу наберіть

```
procedure SetData(const f_name, s_name: string);
```

та натисніть Ctrl+Shift+C. Lazarus автоматично створить наступний код:

```
procedure THuman.SetData(const f_name, s_name: string);
begin

end;
```

5.3.2 Властивості.

У попередній програмі для доступу до приватних полів ми використовували методи, які помістили в розділ public. Фактично ми організували обмін даними між класом та довкіллям. Але для доступу до полів даних у класі краще використовувати властивості. Властивості описують стан об'єкта, оскільки визначаються полями класу та забезпечені двома спеціальними методами для запису та читання значення поля. Таким чином, властивість є спеціальним механізмом для організації доступу до поля. Визначається трьома складовими: поле, метод читання, метод запису. Перевага якості полягає в тому, що методи читання та запису повністю приховані. Це дозволяє вносити зміни в код класу не змінюючи зовнішній код, що використовує його.

Оголошення властивості має вигляд:

```
property ім'я 1: тип <read ім'я 2> <write ім'я 3>;
```

де `property (властивість) – ключове слово;`
ім'я 1 – ім'я властивості, будь-який дозволений
ідентифікатор; тип – тип поля;
ім'я 2 – ім'я методу читання чи безпосередньо ім'я поля; ім'я

3 - ім'я методу запису або ім'я поля;

Кутові дужки означають, що цей параметр може бути відсутнім. У разі відсутності параметра write властивість стає властивістю "тільки для читання". Змінити значення властивості тоді буде неможливо. Відсутність параметра read призводить до того, що властивість стає властивістю "тільки для запису". Однак це не має сенсу, оскільки значення цієї властивості (для читання) буде недоступним.

Найбільш логічно використовувати властивість з наступними параметрами:

```
property ім'я властивості: тип read ім'я поля write ім'я методу запису;
```

У цьому випадку для звернення до значення властивості програма читає безпосередньо захищене поле, а для запису значення властивості буде викликатися метод запису. У методі перед записом можна організувати контроль даних на відповідність будь-яким критеріям залежно від завдання, що вирішується.

Оголошення властивостей слід розміщувати у розділі public. Під час запису властивості ви також можете скористатися функцією "Автозавершення коду" редактора вихідного коду. Введіть

```
property ім'я якості: тип
```

та натисніть Ctrl+Shift+C. Lazarus автоматично завершить оголошення власності, а також заготівлю тіла методу запису. Крім того, додасть до секції private поле з ім'ям <Fім'я властивості> з відповідним типом і процедуру запису зі стандартним ім'ям <Setім'я властивості>. Імена полів заведено починати з літери F, тому Lazarus формує таке ім'я. Але це не обов'язково. Якщо вас не влаштовують імена, запропоновані Lazarus, ви можете їх просто перейменувати. Наприклад, почніть набирати опис класу:

```
  type
    THuman = class
      private
        name: string;
      public
        property name: string
    end;
```

Натисніть Ctrl+Shift+C. Автоматично буде створено наступний код:

```
type

{THuman}

  THuman = class
    private
      Fname: string;
      name: string;
      procedure Setname(const AValue: string);
    public
      property name: string read Fname write Setname;
  end;

implementation
procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname: = AValue;
```

```
end;
```

Оскільки при використанні функції Автозавершення коду Lazarus завжди додає нове поле в секцію private, логічніше починати опис класу зі специфікатора public і визначення властивостей. У цьому випадку секція private також буде автоматично створена і ви отримаєте опис полів та властивостей та готові шаблони реалізації методів запису. У цьому випадку ми отримали опис поля Fname та заготівлю процедури Setname.

При використанні властивості немає потреби явно викликати процедуру запису. Достатньо записати:

```
Person:= THuman.Create;
Person.name:= AValue;
```

Таким чином, для користувача класу властивість виглядатиме як звичайне поле.

Якщо ви хочете (при використанні функції "Автозавершення коду") для читання також використовувати метод, вам достатньо перед ім'ям поля в описі властивості додати слово Get і знову натиснути на комбінацію клавіш Ctrl+Shift+C. Ви отримаєте заготовку процедури читання, наприклад:

```
property name: string read GetFname write Setname;
……
implementation
function THuman.GetFname: string;
begin

end;
```

Розглянемо приклади роботи із властивостями.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  {THuman}
  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setfam(const AValue: string);
      procedure Setname(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
   end;

implementation
{THuman}
procedure THuman.Setname(const AValue: string);
begin
  if Fname = AValue then exit;
  Fname: = AValue;
end;
procedure THuman.Setfam(const AValue: string);
begin
  if Ffam = AValue then exit;
```

```
  Ffam: = AValue;
end;
function THuman.GetData: string;
begin
  Result:= name + '' + fam;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname: string;
begin
  Person:= THuman.Create;
  Person.name:= 'Віталій';
  Person.fam:= 'Петрів';
  fname: = Person.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Person.Free;
end.
```

У цій програмі ми визначили дві властивості name та fam. Скориставшись функцією автозавершення, ми отримали методи запису даних у поля. У програмі явно ми ці методи не викликали. Ми записали про-

сто

```
  Person.name:= 'Віталій';
  Person.fam:= 'Петрів';
```

Решту роботи (виклик методів запису) за нас виконав компілятор. Зверніть увагу, порівняно з попередньою програмою name та fam це не імена полів, а імена властивостей. У наступному прикладі дані зчитуються з клавіатури. Додано контроль за введенням даних.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  {THuman}

  THuman = class
    private
      Fname: string;
      Ffam: string;
      Fbirthday: integer;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
      procedure Setbirthday(const AValue: integer);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
```

```
      property birthday: integer read Fbirthday write
                                        Setbirthday;
      function GetData: string;
   end;

implementation

{THuman}

procedure THuman.Setname(const AValue: string);
begin
  if Fname = AValue then exit;
  Fname: = AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam = AValue then exit;
  Ffam: = AValue;
end;

procedure THuman.Setbirthday(const AValue: integer);
begin
  if Fbirthday = AValue then exit;
  Fbirthday:= AValue;
end;
function THuman.GetData: string;
var
  s: string;
```

```
begin
  str (birthday, s);
  Result:= name + ' + fam + ' ' + s; end;
end.

program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname, sname, s_year: string;
  code, year: integer;
begin
  Person:= THuman.Create;
  writeln(UTF8ToConsole('Введіть ім'я'));
  readln(fname);
  Person.name:=fname;
  writeln(UTF8ToConsole('Введіть прізвище'));
  readln(sname);
  Person.fam:= sname;
  writeln(UTF8ToConsole('Введіть рік
  народження')); readln(s_year);
  val(s_year,    year,    code);        //    контроль
                 введення
```

```
  if code = 0 then
    Person.birthday:= year
  else
    Person.birthday:= 0;
  fname: = Person.GetData;
  writeln(UTF8ToConsole('Це: '), fname);
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Person.Free;
end.
```

5.4. успадкування

Уявіть собі, що є клас, який вас у принципі задовольняє, але чогось у ньому трохи не вистачає. Що доведеться наново розробляти клас? Звичайно ж, наново створювати клас не потрібно. Досить створити клас на основі існуючого і просто додати до нього члени, що не вистачають (поля, методи і властивості). Це і є спадкування. Новий клас називається спадкоємцем чи нащадком чи дочірнім класом. А старий клас називається батьківським класом чи предком або базовим класом. Оголошується клас спадкоємець наступним чином:

```
type
<ім'я класу спадкоємця> = class(ім'я класу батька)
            <Опис полів, методів та властивостей,
        характерних тільки для класу спадкоємця>
end;
<Реалізація методів>
```

Якщо ім'я класу батька не вказано, за умовчанням клас успадковується

від загального класу TObject. Наш клас THuman із попередніх прикладів є спадкоємцем TObject.

Клас спадкоємець отримує (успадковує) всі поля, методи та властивості класу батька. У той же час створений клас, у свою чергу, може виступати як батько. У Free Pascal існує так звана ієрархія класів, на вершині якого знаходиться клас TObject. Таким чином, клас TObject є прабатьком всіх інших класів.

Наприклад, створимо клас TStudent (студент). Можна створити цей клас з нуля, але оскільки студент є людиною, то логічно буде створити клас на основі класу THuman.

```
program project1;
{$mode objfpc}{$H+}

uses
  CRT, FileUtil;
type

  THuman = class      // оголошення базового класу
    private
      name: string;
      fam: string;
    public
      function GetData: string;
  end;
function THuman.GetData: string;
begin
  Result:= name + '' + fam;
end;
```

```
type

  TStudent = class(THuman) // оголошення класу – спадкоємця
    private
      group: string;
  end;

var

  Student: TStudent;
  fname: string;

begin
  Student:= TStudent.Create;
  Student.name:= 'Віталій';
  Student.fam:= 'Петрів';
  Student.group:= 'ПОВТАС-1/09';
  fname: = Student.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Student.Free;
end.
```

Тут у клас TStudent ми ввели поле group, характерне лише для студента. У разі це група. У той же час, завдяки спадкуванню, будуть доступні поля name (ім'я) та fam (прізвище), а також метод GetData батьківського класу. І клас THuman і клас TStudent є нащадками класу TObject, тому для класу TStudent доступний метод (конструк-

тор) Create класу TObject (докладніше про конструкторів дивіться нижче). Зверніть увагу, якщо ви збираєтеся використовувати лише об'єкт класу TStudent, то створювати екземпляр класу THuman зовсім не обов'язково. Більш того, ви можете створювати клас спадкоємець в іншому модулі, ніж описаний базовий клас.

```
unit Unit1;
{$mode objfpc}{$H+}
interface

uses
  Classes, SysUtils;

type

  THuman = class      // оголошення класу
    protected
      name: string;
      fam: string;
    public
      function GetData: string;
  end;
implementation

function THuman.GetData: string;
begin
  Result:= name + '' + fam;
end;
end.
```

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type

  TStudent = class(THuman) // оголошення класу - спадкоємця
    private
      group: string;
  end;

var

  Student: TStudent;
  fname: string;

begin
  Student:=TStudent.Create;Stude
  nt.name:= 'Віталій';
  Student.fam:= 'Петрів';
  Student.group:= 'ПОВТАС-1/09';
  fname: = Student.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Student.Free;
end.
```

Тут ми описали батьківський клас у окремому модулі Unit1. Зверніть увагу, щоб поля батьківського класу були доступні класу на-

слідчому, ми в класі THuman оголосили полязі специфікатором protected.

Якщо поля батьківського класу захистити специфікатором private, то дочірній клас може отримати доступ до його полів за допомогою його властивостей (зрозуміло, що в батьківському класі мають бути описані відповідні властивості).

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;
type

  {THuman}
  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
   end;
```

```
implementation

{THuman}

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.GetData: string;
begin
  Result:= name + '' + fam;
end;
end.
program project1;
{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;

type
```

```
  TStudent = class(THuman) // оголошення класу - спадкоємця
    private
       group: string;
  end;

var
  Student: TStudent;
  fname: string;

begin
  Student:= TStudent.Create;
  Student.name:= 'Віталій';
  Student.fam:= 'Петрів';
  Student.group:= 'ПОВТАС-1/09';
  fname: = Student.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Student.Free;
end.
```

Напишемо клас TProfessor (викладач). Викладач теж є людиною (чи у вас є сумніви щодо цього?!), тому цілком природно, що цей клас також успадковуватиметься від класу THuman. Так само як студенти "купкуються" до груп, так і викладачі об'єднуються у кафедри. Тому для класу введемо поле kafedra. У наступному прикладі створюються відразу два класи спадкоємця. Наведу лише код основної програми. Клас THuman (модуль Unit1, див. попередній приклад) не зазнає жодних змін.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type
  TStudent = class(THuman) // оголошення класу - спадкоємця
    private
      group: string;
  end;
  type
    TProfessor = class(THuman) // оголошення класу -
    спадкоємця
      private
        kafedra: string;
    end;
var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;
begin
  Student:= TStudent.Create;
  Professor:= TProfessor.Create;
  Student.name:= 'Віталій';
  Student.fam:= 'Петрів';
  Student.group:= 'ПОВТАС-1/09';
  fname: = Student.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  Professor.name:= 'Іван';
  Professor.fam:= 'Іванів';
  Professor.kafedra:='Програмування';
```

```
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Student.Free;
end.
```

5.5. Поліморфізм

В усіх прикладах розділу 5.4. ми використовували метод GetData батьківського класу. Але він повертає лише значення полів name та fam. Тому, незважаючи на те, що ми в програмі вказували значення полів group та kafe-dra, вони на екран не виводилися. Як вивести значення цих полів?

Рішення полягає у написанні відповідного методу у дочірньому класі. Причому ми вільні надавати цьому методу будь-яке ім'я. Наприклад, ми можемо написати методи:

```
function TStudent.A: string;
begin
  Result:= name + ' + fam + ', група ' + group;
end;

function TProfessor.B: string;
begin
  Result:= name + '' + fam + ', кафедра '+ kafedra;
end;
```

Але, давайте згадаємо, що методи, як ми зазначали вище, характеризують деякі дії з обробки даних класу. У цьому контексті ми можемо ввести в розгляд дію "Отримати інформацію про об'єкт". В одному випадку це буде "Отримати відомості про людину", в іншому випадку "Отримати відомості про студента", в третьому – "Отримати відомості про викладача". У всіх трьох випадках це в принципі однотипні дії, хоча й дещо відрізняються по суті. Ми можемо написати дуже коротко – "Отримати відомості". Так ось, повертаючись "до наших баранів", ми можемо констатувати, що функція GetData це і є дія "Отримати відомості". Тому ми можемо використовувати в дочірніх класах методи з таким самим ім'ям, що й у батьківському класі. Звичайно, реалізації цих методів різняться і навіть можуть відрізнятися кардинально, але по суті, якщо ви хочете реалізувати однотипні аспекти поведінки об'єктів, ви можете і повинні давати однакові імена методам. Даючи їм різні імена, ви можете дуже швидко заплутатися, особливо якщо похідних класів досить багато. Звідси, ми можемо напи-

сати:

```
function TStudent.GetData: string;
begin
  Result:= name + ' + fam + ', група ' + group;
end;

function TProfessor.GetData: string;
begin
  Result:= name + '' + fam + ', кафедра '+ kafedra;
end;
```

Таке явище, коли методи класу батька та методи дочірніх класів мають однакові імена, але різні реалізації називають поліморфізмом. Функції THuman.GetData, TStudent.GetData і TProfessor.GetData безперечно відрізняються, оскільки повертають

рядки з різним змістом. В одному випадку просто ім'я та прізвище, в іншому ім'я, прізвище та найменування групи, а в третьому випадку – ім'я, прізвище та назва кафедри. Можна ще більше підкреслити їхню відмінність, якщо у класі TStudent описати тип поля group як integer. У цьому випадку повертатиметься не назва групи, а її номер. Реалізація цього методу буде наступною:

```
function TStudent.GetData: string;
var
  s: string;
begin
  str (group, s);
  Result:= name + ' + fam + ', група ' + s;
end;
```

5.5.1 Раннє зв'язування.

Одноіменні методи, визначені таким чином, називаються статичними. При виклику методу, окрім явно описаних параметрів функції або процедури, неявно передається ще один параметр self - покажчик об'єкт, що викликав метод. Таким чином, компілятор легко визначає об'єкт якого класу викликав метод та організує виклик потрібного методу. Це так зване раннє зв'язування. Метод дочірнього класу як би підміняє метод батьківського класу з тим самим ім'ям. Набагато частіше застосовуються терміни перекриття чи перевизначення. Можна було сказати – метод дочірнього класу перекриває (перевизначає) метод батьківського класу з тим самим ім'ям. При цьому кількість і типи параметрів нового методу дочірнього класу та методу, що перевизначається, можуть не збігатися. Розглянемо приклад, де використовується механізм раннього зв'язування.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type

  {THuman}

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
   end;
implementation

{THuman}

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:=AValue;
end;
```

```
procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.GetData: string;
begin
  Result:= name + '' + fam;
end;
end.

program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;

type

  TStudent = class(THuman)
    private
      group: string;
    public
      function GetData: string;
  end;

function TStudent.GetData: string;
```

```
begin
  Result:= name + ' + fam + ', група ' + group;
end;

  type

    TProfessor = class(THuman)
      private
        kafedra: string;
      public
        function GetData: string;
    end;

function TProfessor.GetData: string;
begin
  Result:= name + '' + fam + ', кафедра '+ kafedra;
end;

var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;

begin
  Student:= TStudent.Create;
  Professor:= TProfessor.Create;
  Student.name:= 'Віталій';
  Student.fam:= 'Петрів';
  Student.group:= '"ПОВТАС-1/09"';
```

```
  fname: = Student.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  Professor.name:= 'Іван';
  Professor.fam:= 'Іванів';
  Professor.kafedra:='"Програмування"';
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Student.Free;
  Professor.Free;
end.
```

Подивіться реалізації методів GetData. У чомусь вони схожі. Особливо якщо ми перепишемо їх у вигляді:

```
function TStudent.GetData: string;
begin
  Result:= name + '' + fam;
  Result: = Result + ', група '+ group;
end;
function TProfessor.GetData: string;
begin
  Result:= name + '' + fam;
  Result:= Result + ', кафедра '+ kafedra;
end;
```

Перші оператори цих функцій збігаються з методом батьківського класу. Це у наших прикладах методи дуже прості. Насправді методи можуть бути дуже складними і налічувати не один десяток, а то й сотень рядків

коду. Що, треба копіювати весь схожий код у кожен із дочірніх методів? Звісно ж, ні! Виявляється, можна викликати метод батьківського класу за допомогою ключового слова, внесеного до нього. Наприклад:

```
function TStudent.GetData: string;
begin
  Result:= inherited GetData; Result:
  = Result + ', група '+ group;
end;
```

Тут спочатку викликається метод батьківського класу, потім додається новий код.

5.5.2 Пізніше зв'язування.

Вище ми бачили, що з статичних методів дозвіл зв'язків, тобто. визначення того, який саме метод слід викликати, відбувається на етапі компіляції. Але бувають ситуації, коли компілятор неспроможна визначити, якого об'єкту (примірнику класу) належить метод. Розглянемо приклад. Нехай нам необхідно, щоб на екран виводилася, крім колишньої інформації, ще й статус людини, тобто. хто він – студент чи викладач. І тому введемо новий метод Status. Цей метод викликатиметься з методу GetData. Під час компіляції невідомо, об'єкт якого класу викликає метод Status. Це визначається етапі виконання, коли точно відомий активний на даний момент об'єкт. Дозвіл зв'язків на етапі виконання називається пізнім зв'язуванням. Для реалізації механізму пізнього зв'язування застосовуються ключові слова virtual та override. Метод батьківського класу позначається ключовим словом virtual. Методи дочірніх класів, що перекривають метод батьківського класу, позначаються ключовим словом

override, а всі ці методи, включаючи батьківський, називаються віртуальними методами. Причому на відміну статичних методів, кількість, тип і порядок слідування параметрів у віртуальних методах повинні збігатися. Розглянемо приклад.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  {THuman}
  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
      функція Status: string; virtual; end;
implementation
{THuman}
procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
```

```
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.Status: string;
begin

end;

function THuman.GetData: string;
begin
  Result:= name + '' + fam + Status;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type

  TStudent = class(THuman)
    private
      group: string;
```

```
    public
      function GetData: string;
      функція Status: string; override;
  end;

function TStudent.Status: string;
begin
  Result:= '- студент';
end;
function TStudent.GetData: string;
begin
  Result:= inherited GetData + ', група '+ group;
end;
type
  TProfessor = class(THuman)
    private
      kafedra: string;
    public
      function GetData: string;
      функція Status: string; override;
  end;

function TProfessor.Status: string;
begin
  Result:= '- викладач';
end;

function TProfessor.GetData: string;
begin
```

```
  Result:= inherited GetData + ', кафедра '+
kafedra; end;

var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;
begin
  Student:= TStudent.Create;
  Professor:= TProfessor.Create;
  Student.name:= 'Віталій';
  Student.fam:= 'Петрів';
  Student.group:= '"ПОВТАС-1/09"';
  fname: = Student.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  Professor.name:= 'Іван';
  Professor.fam:= 'Іванів';
  Professor.kafedra:= '"Програмування"';
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Student.Free;
  Professor.Free;
end.
```

Крім віртуальних методів, можуть бути оголошені і так звані динамічні методи. Вони оголошуються за допомогою ключового dynamickoго слова.

А в класах спадкоємців все того ж ключового слова override. З точки зору спадкування, методи цих двох видів однакові: вони можуть бути перекриті в дочірньому класі лише однойменними методами, які мають той самий тип. Різниця полягає у реалізації механізму пізнього зв'язування. Для віртуальних методів використовується спеціальна таблиця віртуальних методів (ТВМ), а динамічних методів таблиця динамічних методів (ТДМ).

У ТВМ зберігаються адреси всіх віртуальних методів класу, включаючи батьківських класів, навіть якщо вони не перекриті в даному класі. Тому виклики віртуальних методів відбуваються досить швидко, проте ТВМ потребує більшої пам'яті.

З іншого боку, кожному динамічному методу надається унікальний індекс. У ТДМ зберігаються індекси та адреси лише тих динамічних методів, які описані в даному класі, тому ТДМ займає менше пам'яті. При виклику динамічного методу відбувається пошук у ТДМ даного класу. Якщо пошук не дав результатів, пошук триває у ТДМ усіх класів-батьків у порядку ієрархії. Чим більша глибина ієрархії класів, тим повільніше працюватимуть виклики динамічних методів.

Щоб реалізувати динамічні методи нашого прикладу, досить замінити слово virtual, на dynamic.

Подивимося тепер код методу Status у батьківському класі. Фактично в цьому методі нічого не робиться, функція повертає порожній рядок. У разі зручніше оголосити метод як абстрактний з допомогою ключового слова abstract. Реалізація методу, який оголошено як абстрактний, у цьому класі не проводиться, але обов'язково має бути перевизначений у дочірніх класах. Зверніть увагу лише віртуальні методи можуть бути оголошені абстрактними. Для нашого прикладу, щоб оголосити абстрактний метод, додайте в класі THuman в оголошення методу Status після слова virtual через точку з комою ключове слово abstract і видаліть код реалізації методу.

5.5.3 Конструктори та деструктори.

Для створення об'єктів ми використовували метод Create. Це званий конструктор, тобто. спеціальний метод класу, призначений для виділення пам'яті та розміщення в пам'яті екземпляра класу. Крім того, завдання конструктора входить ініціалізація значень полів екземпляра класу. Стандартний конструктор встановлює всі дані нового екземпляра класу в нуль. В результаті всі числові поля та поля порядкового типу набувають нульових значень, рядкові поля стають порожніми, а поля, що містять покажчики та об'єкти набувають значення nil. Під стандартним конструктором мають на увазі конструктор, успадкований класом від класу TObject.

Якщо потрібно ініціалізувати дані екземпляра класу певними значеннями, необхідно написати свій власний конструктор. Допускається використання кількох конструкторів. Бажано, щоб усі конструктори мали стандартне ім'я Create. Це дозволяє перевизначати конструктори, включаючи і стандартний конструктор, для виконання будь-яких корисних дій. Опис конструктора провадиться за допомогою ключового слова constructor. Давайте в останньому прикладі попереднього розділу до класу THuman додамо конструктор. Код програми буде наступним:

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;
type
```

```
   {THuman}

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      constructor Create; // конструктор
      function GetData: string;
      функція Status: string; virtual; abstract;
   end;

implementation

{THuman}

конструктор THuman.Create; // Реалізація конструктора
begin
  Fname:= 'Андрій';
  Ffam:= 'Аршавін';
end;

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
```

```
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.GetData: string;
begin
  Result:= name + '' + fam + Status;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type

  TStudent = class(THuman)
    private
      group: string;
    public
      function GetData: string;
      функція Status: string; override;
  end;
```

```
function TStudent.Status: string;
begin
  Result:= '- студент';
end;

function TStudent.GetData: string;
begin
  Result:= inherited GetData + ', група '+ group;
end;

type

  TProfessor = class(THuman)
    private
      kafedra: string;
    public
      function GetData: string;
      функція Status: string; override;
  end;

function TProfessor.Status: string;
begin
  Result:= '- викладач';
end;

function TProfessor.GetData: string;
begin
  Result:= inherited GetData + ', кафедра '+
kafedra; end;
```

```
var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;
begin
  Student:= TStudent.Create;
  fname: = Student.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  Professor:= TProfessor.Create;
  Student.name:= 'Віталій';
  Student.fam:= 'Петрів';
  Student.group:= '"ПОВТАС-1/09"';
  fname: = Student.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  Professor.name:= 'Іван';
  Professor.fam:= 'Іванів';
  Professor.kafedra:= '"Програмування"';
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Це:' + fname));
  writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
  Student.Free;
  Professor.Free;
end.
```

Під час виконання оператора

```
Student:= TStudent.Create;
```

спрацює конструктор, визначений нами в класі THuman, оскільки він пе-

відкриває стандартний конструктор. В результаті поля Fname та Ffam будуть ініціалізовані значеннями 'Андрій' та 'Аршавін'. Після виконання операторів

```
fname: = Student.GetData;
writeln(UTF8ToConsole('Це:' + fname));
```

на екран буде виведено
"Це: Андрій Аршавін – студент, гурт".

Зверніть увагу, що назва групи виведена не буде. Для ініціалізації значення поля group необхідно написати конструктор у класі TStudent, наприклад таким чином:

```
конструктор TStudent.Create;
begin
  group:= 'Арсенал'; end;
```

Тепер під час виконання оператора

```
Student:= TStudent.Create;
```

буде викликаний конструктор, визначений у класі TStudent і на екран буде виведено:

"Це: – студент, гурт "Арсенал"".

Ми бачимо, що ім'я та прізвище відсутні. Це знову відбувається через те, що конструктор Create дочірнього класу TStudent перекриває конструк-

тор батьківського класу THuman. Щоб викликати конструктора батьківського класу, необхідно використовувати ключове слово взахищеному. Таким чином, реалізацію конструктора класу TStudent необхідно записати у вигляді:

```
конструктор TStudent.Create;
begin
  inherited Create;
  group:= 'Арсенал';
end;
```

Тепер на екран буде виведено рядок так, як ми хотіли: "Це:
Андрій Аршавін – студент, гурт "Арсенал"".

Цілком можливо використовувати конструктори з параметрами. Ось як, наприклад, може виглядати конструктор із параметрами класу THuman:

```
public
      constructor Create(n, f: string);
……
end;
implementation
{THuman}
конструктор THuman.Create(n, f: string);
begin
  Fname: =
  n;
  Ffam: = f;
end;
```

Конструктор класу TStudent:

```
public
      constructor Create(gr: string);
……
end;
constructor TStudent.Create(gr: string);
begin
  inherited Create('Андрій', 'Аршавін');
  group: = gr;
end;
```

Якщо ви зараз викличете конструктор у вигляді

```
Student:= TStudent.Create;
```

то компілятор "сильно лаятиметься"! Необхідно викликати конструктор таким чином:

```
Student:= TStudent.Create(''Арсенал'');
```

Зверніть увагу і на виклик конструктора батьківського класу

```
inherited Create('Андрій', 'Аршавін');
```

Оскільки ми з вами чудово знаємо, що Андрій Аршавін не студент, видозмініть описи класів, щоб на екран виводилося, ну щось на кшталт:
"Андрій Аршавін – футболіст команди "Арсенал"".

Насамкінець зазначимо, що конструктор створює новий об'єкт тільки в тому випадку, якщо перед його ім'ям вказано ім'я класу. Якщо вказати ім'я вже існуючого об'єкта, він поведеться по-іншому: не створить новий об'єкт, а лише виконає код, що міститься в тілі конструктора.

Деструктор має стандартне ім'я Destroy і призначений для ніщо-

ня об'єкта:

```
Student.Destroy;
```

У тілі деструктора зазвичай повинні знищуватися вбудовані об'єкти та динамічні дані, як правило, створені конструктором. Як і звичайні методи деструктор може мати параметри, але ця можливість використовується вкрай рідко. Для оголошення деструктора використовується ключове слово destructor.

Destroy – це віртуальний деструктор класу TObject. Тому при перевизначенні деструктора його необхідно оголошувати з ключовим словом override, наприклад:

```
destructor Destroy; override;
```

У тілі самого деструктора потрібно викликати батьківський деструктор (inherited Destroy). Цим забезпечується формування ланцюжка деструкторів, що сходить до деструктора класу TObject, який і здійснює звільнення пам'яті, зайнятої об'єктом.

Рекомендується замість методу Destroy використовувати метод Free. Цей метод спочатку перевіряє, чи є поточний об'єкт nil і лише, потім викликає метод Destroy. Якщо об'єкти створюються на початку роботи програми, а знищуються в самому кінці, то застосування методу Free є найоптимальнішим.

На цьому ми закінчимо наш короткий екскурс до об'єктно-орієнтованого програмування. У наступному розділі ми ще торкнемося деяких аспектів ООП стосовно створення програм з графічним інтерфейсом.

Докладнішу інформацію про ООП ви можете отримати з [5] та [6]. Хоча

у цих книгах матеріал викладається стосовно Turbo Pascal і Delphi, проте ви можете використовувати їх, особливо [6].

Якщо ви пам'ятаєте, при створенні нашого першого консольного додатка (див. 2.1.10) Lazarus запропонував нам заготівлю коду, який ми з вами просто видалили. Тоді нам цей код був незрозумілий. Ну а тепер ..., тепер ви можете легко розібратися з цим кодом! Так-так, Lazarus створив для нас заготівлю класу з конструктором та деструктором!

Глава 6 Програмування програм із графічним інтерфейсом

У попередніх розділах ми з вами створювали лише консольні додатки. При вивченні основ мови програмування (не тільки Паскаля, а й будь-якої іншої мови), консольні додатки є найбільш зручними, оскільки дозволяють зосередитись на суті тієї чи іншої конструкції чи можливості мови.

Однак, як ви неодноразово мали змогу бачити, інтерфейс користувача був досить примітивний і мізерний. Здебільшого вдавалося збудувати лише просте меню. Можна, звичайно, ціною значних зусиль, створити в консольному додатку і "пристойніший" інтерфейс. Прикладом може бути IDE самого компілятора FPC.

Але нині найбільшу та заслужену популярність завоювала ідея графічного інтерфейсу. Згідно з цією ідеєю, кожна програма працює у своєму власному вікні. Управління вікнами стандартизовано. Стандартний інтерфейс дуже зручний для користувачів і значно полегшує вивчення різних програмних засобів. Запустіть, наприклад, Word та Excel у Windows або текстовий процесор OpenOffice.org та електронні таблиці OpenOffice.org у Linux. Ви побачите багато схожих елементів керування вікнами (кнопки, перемикачі, меню тощо).

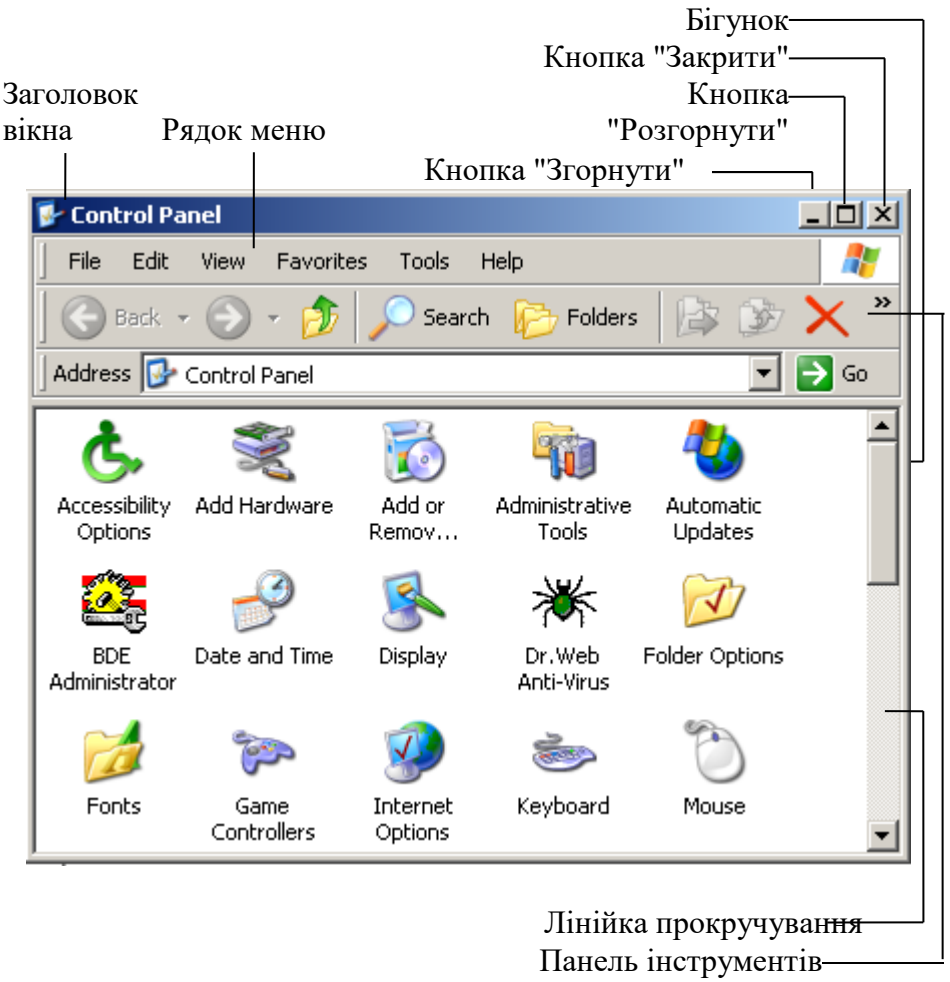
Графічний інтерфейс користувача, інакше GUI (Graphics User Interface) є обов'язковим компонентом більшості програмних продуктів, орієнтованих на роботу кінцевого користувача. Поява графічного інтерфейсу - одне з найважливіших досягнень у галузі розробки програмного забезпечення останніх років. Надалі ми будемо використовувати як термін "графічний інтерфейс користувача", так і термін

"додаток з GUI" або "додаток GUI".

Розглянемо коротко деякі елементи графічного інтерфейсу.

6.1. Елементи графічного інтерфейсу

Як зазначалося, у додатках з графічним інтерфейсом управління вікнами стандартизовано. Практично всі вікна мають одні й самі елементи управління. Розглянемо їх у малюнку 6.1. для операційної системи Windows.



Мал. 6.1 Елементи вікон Windows

Будь-яке вікно має рядок заголовка вякому вказується ім'я програми та ім'я документа.

При натисканні кнопки "Згорнути" активне вікно зменшується до розмірів піктограми та поміщається на панель завдань. Слід пам'ятати, що програма у разі продовжує працювати, але у згорнутому вигляді. При натисканні кнопки Розгорнути вікно розгортається до максимально можливих розмірів. При цьому кнопка "Розгорнути" замінюється на кнопку "Відновити"

Якщо клацнути по цій кнопці, вікно відновить свій початковий розмір. Щоб закрити вікно, натисніть кнопку "Закрити".

Другий рядок вікна називається рядком меню. Якщо клацнути по якомусь пункту, то з'явиться підменю – вертикальне меню в якому можна вибрати відповідні команди.

Панель інструментівдозволяє виконувати деякі дії швидше, використовуючи кнопки на панелі та не звертаючись до пунктів меню. Кнопки на панелі інструментів дублюють деякі команди меню, які найчастіше використовуються.

Лінійка прокручуваннявикористовується у тому випадку, коли вся інформація не міститься у вікні. Якщо натиснути на кнопки , то інформація у вікні зрушуватиметься вгору або вниз.

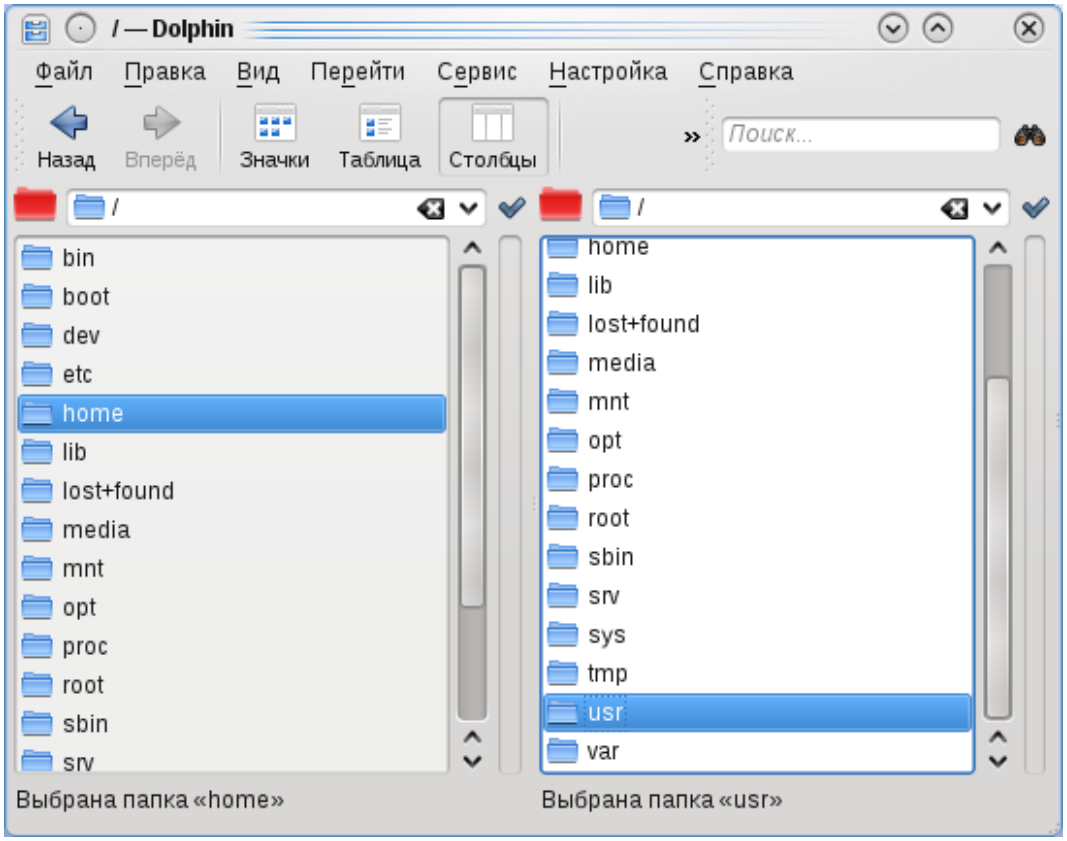
Бігунокпоказує відносне становище у документі. Наприклад, якщо бігунок знаходиться вгорі лінійки прокручування, то ви знаходиться ближче до початку документа, якщо внизу, то ближче до кінця документа. Можна прокручувати текст і за допомогою бігунка. Для цього треба натиснути на бігунок мишею і, не відпускаючи рухати бігунок у потрібному напрямку. Лінійки прокручування бувають вертикальні та горизонтальні.

Меню

У вікні кожної програми може знаходитись рядок меню. Це так зване горизонтальне меню. Кожен пункт цього меню складається у свою чергу з відповідних підменю – вертикальних меню. Для того щоб від-

крити підменю треба клацнути по пункту горизонтального меню. Кожен пункт меню містить або команди, за допомогою яких можна виконати якісь дії, або режими (опції), які можна активізувати або дезактивізувати, або містять ще додаткові заміни (вкладені меню).

Знайдіть схожі елементи у вікні файлового менеджера Dolphin у Linus, рис. 6.2.

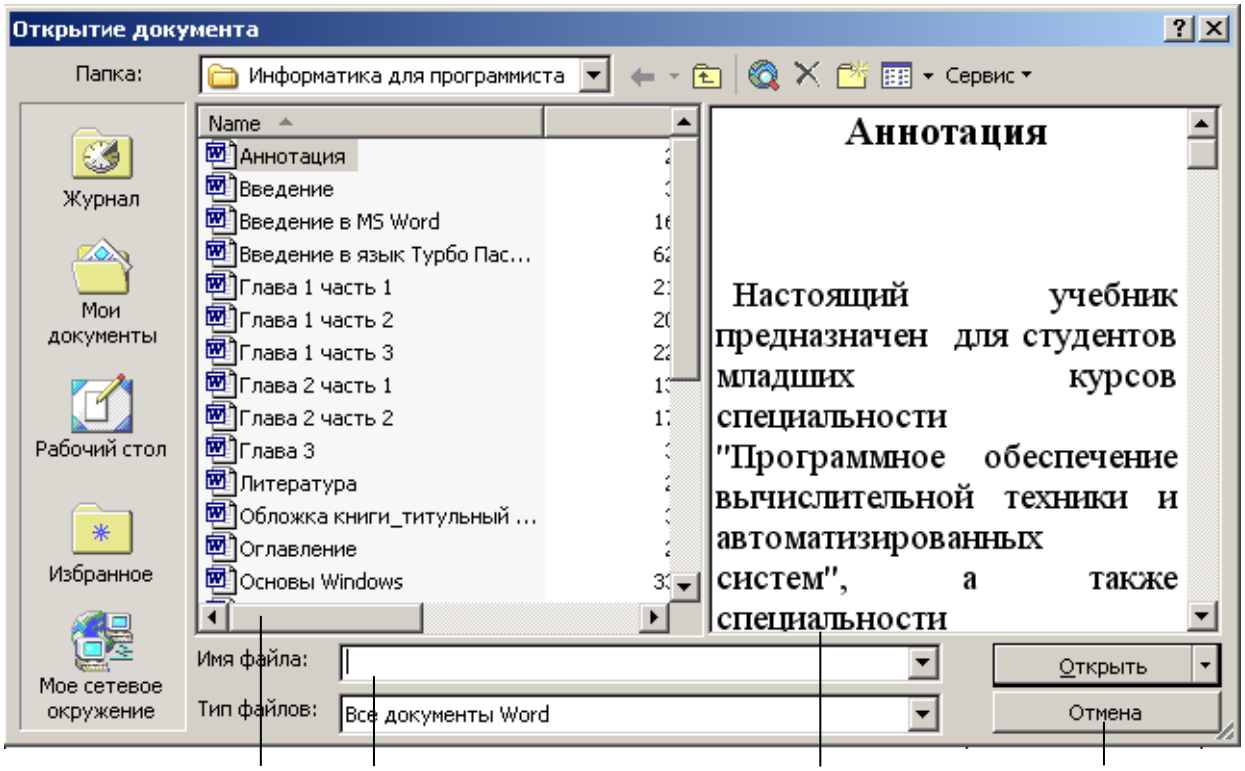


Мал. 6.2. Вікно файлового менеджера Dolphin

Діалогові вікна

Для виконання деяких команд необхідно встановити один або кілька параметрів. Для цього використовуються діалогові вікна. Діалогові вікна містять спеціальні області (які називаються елементами керування або параметрами). Значення параметрів слід вибрати з певного списку або ввести з клавіатури.

Типи параметрів діалогових вікон.



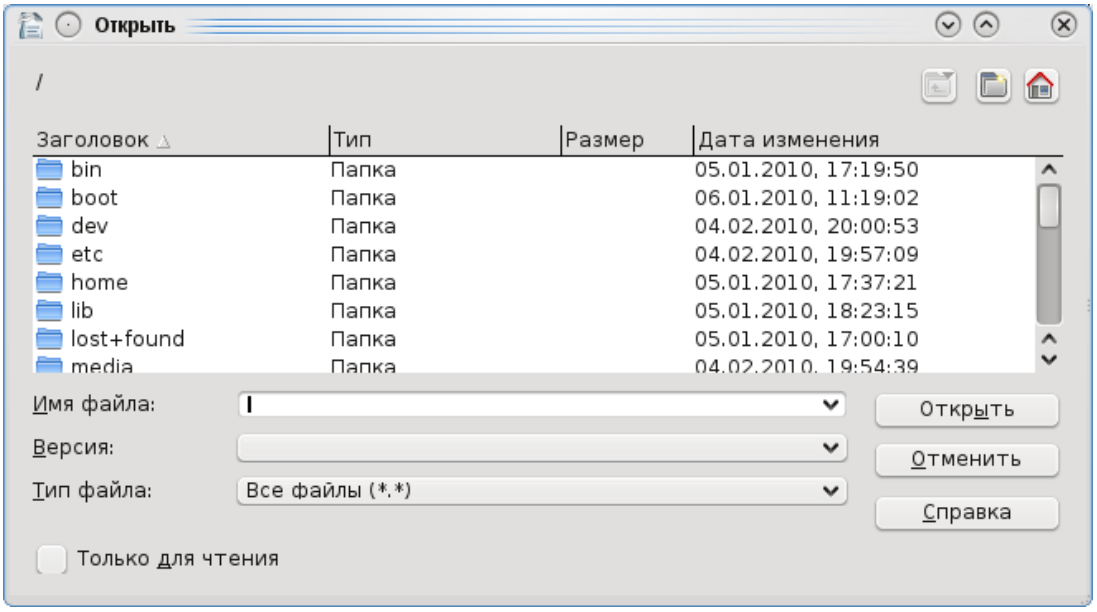
Поле списку Поле введення Поле попереднього перегляду Кнопки команд Мал. 6.3 Діалогове вікно відкриття документа

Кнопки команд- Використовуються для виконання дій. Усі діалогові вікна обов'язково мають кнопки ОК та Скасувати. При натисканні кнопки ОК діалог завершується і команда з вибраними параметрами виконується. При натисканні кнопки Скасувати вікно діалогу просто закривається і команда не виконується. Наявність інших клавіш залежить від виду команди.

Поле введення– використовується для введення параметра з клавіатури.

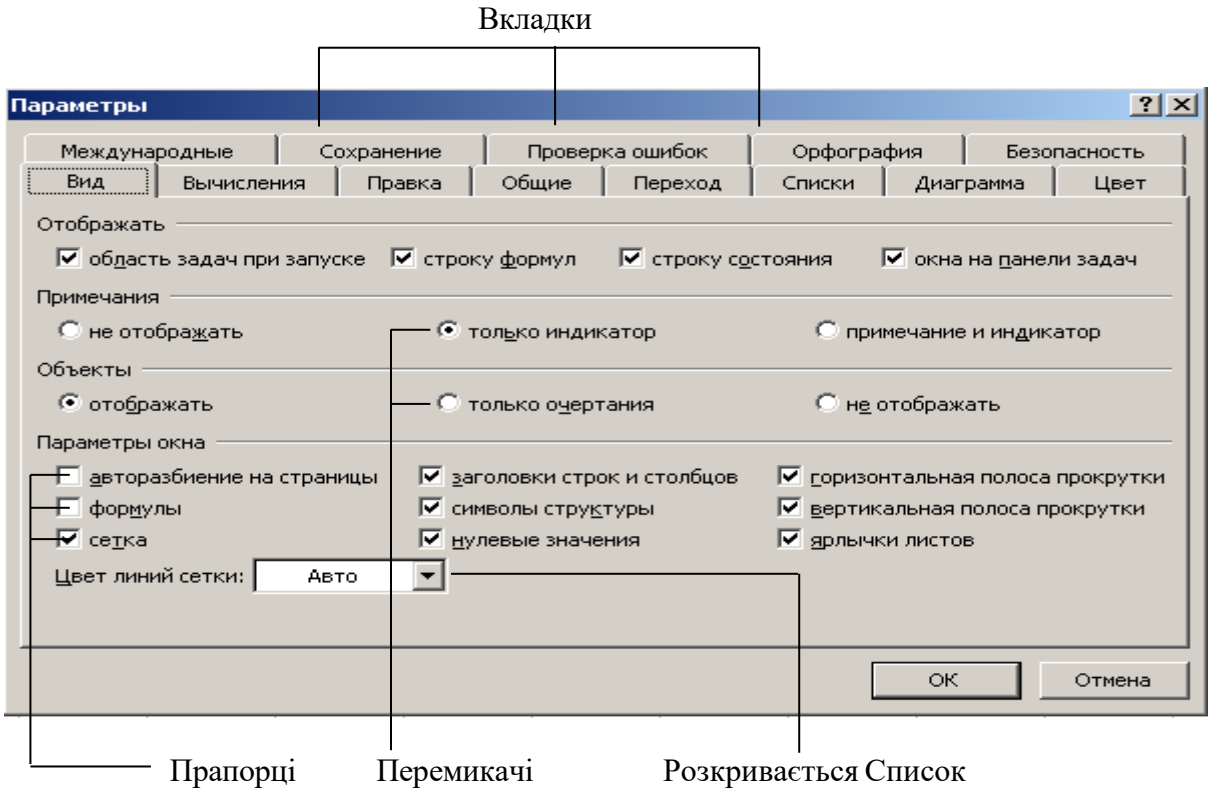
Поле списку– потрібний параметр вибирається зі списку. Вибір здійснюється простим клацанням по потрібному елементу списку. При необхідності можна скористатися лінійкою прокручування.

Знайдіть на рис. 6.4. схожі елементи та параметри діалогових вікон в операційній системі Linux.



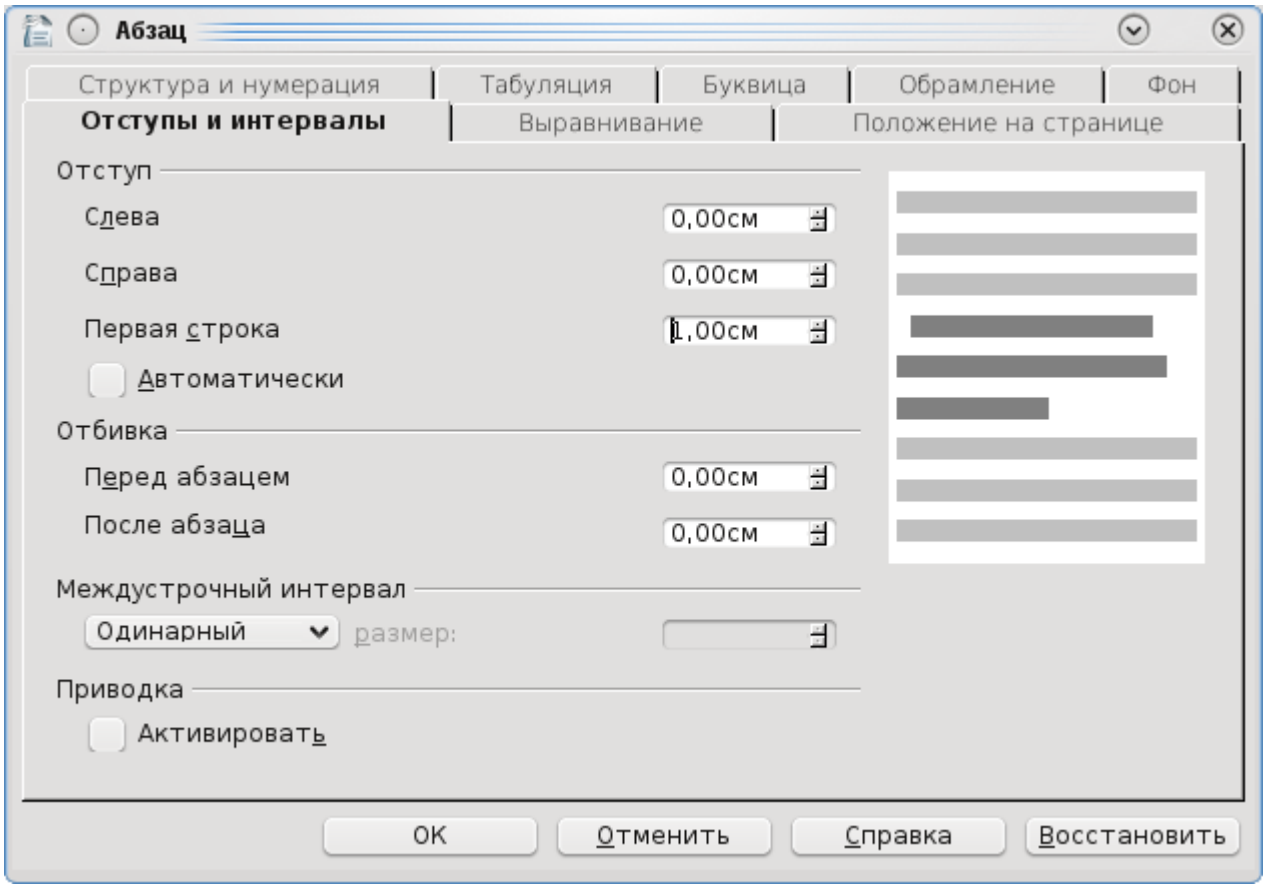
Мал. 6.4. Діалогове вікно відкриття файлу в Linux

Вкладки– деякі команди використовують дуже велику кількість різних параметрів. У цьому випадку діалогові вікна оформляються у вигляді вкладок або підшивок. Щоб вибрати певну групу параметрів, потрібно натиснути на потрібну вкладку. Малюнок 6.5.



Мал. 6.5. Діалогове вікно із вкладками

На рис. 6.6 показано діалогове вікно з вкладками для текстового процесора OpenOffice.org



Мал. 6.6. Діалогове вікно із вкладками для текстового процесора OpenOffice.org

Список, що розкривається– використовується для економії місця на екрані. Щоб розкрити цей список, необхідно натиснути кнопку з трикутником. Надалі робота з цим списком повністю аналогічна до роботи зі звичайним списком.

Прапорець (індикатор)– це поле може мати лише два значення: увімкнено або вимкнено. За допомогою прапорців можна вибрати (включити) деякі режими або опції.

Щоб увімкнути або вимкнути прапорець, потрібно просто клацнути по прапорцю. Угоди тут такі:

 - вимкнено
- увімкнено

Якщо є кілька прапорців (індикаторів), кожен з них може бути включений або вимкнений незалежно від інших прапорців.

Перемикач– кажуть ще радіокнопка, також має лише два значення: увімкнено/вимкнено. Має вигляд кружечка з крапкою або без. Якщо перемикач увімкнено, він відзначається кружечком з точкою всередині. Якщо є кілька перемикачів, то може бути увімкнено лише один перемикач. Таким чином, цей механізм дозволяє вибрати лише один із можливих режимів (опцій). У цьому вся відмінність перемикачів від прапорців.

Також у літературі ви можете зустріти поняття "група незалежних перемикачів" – це в нашій термінології прапорці та "група залежних перемикачів" – по-нашому просто перемикачі.

Як бачимо, стандарти GUI для різних операційних систем і платформ дещо різняться, але в цілому намічається тенденція до їх зближення та уніфікації.

Усі розглянуті елементи графічного інтерфейсу, а також безліч інших можна реалізувати в середовищі Lazarus за допомогою спеціальної бібліотеки LCL (Lazarus Component Library). Ця бібліотека надає програму місту цілу палітру так званих візуальних компонентів, за допомогою яких можна розробляти інтерфейс програми. При цьому програміст звільняється від дрібних операцій і може зосередитися саме на проектуванні інтерфейсу своєї майбутньої програми. Для цього він вибирає необхідні йому візуальні компоненти, налаштовує їх під власні потреби і таким чином створює зовнішній вигляд (інтерфейс) своєї програми. При цьому він ще до компіляції своєї програми бачить результати проектування інтерфейсу користувача.

Оскільки елементи бібліотеки LCL доступні для всіх підтримуваних платформ, GUI-додатки створені на одній платформі (наприклад,

Windows), можуть бути без зміни скомпільовані на іншій платформі (наприклад, OS X або Linux). При цьому слід пам'ятати, що графічний інтерфейс у різних операційних системах реалізовано зовсім інакше. Так, Windows GUI базується на функціях WinAPI, а в Linux на бібліотеках GTK-2 або QT.

6.2. Відмінності між консольними та графічними додатками.женнями

Отже, програма з GUI відрізняється від консольного наявністю графічного інтерфейсу (нагадаю, що консольна програма використовує екран у текстовому режимі, а програма з GUI у графічному режимі). Але між консольним та GUI-додатком існує ще одна, більш істотна різниця. Консольний додаток, отримавши управління, далі повністю сам контролює обчислювальний процес. За потребою запитує дані для роботи (оператор read), виводить результати обчислень на екран (оператор writeln) і завершує роботу з досягнення останнього оператора програми.

Абсолютно інакше працюють GUI-програми. Вони реагують на події. Подія може викликати дію користувача, наприклад, натискання мишею на якусь кнопку в програмі. Подія може викликати операційна система або подія може породити сам додаток. Усі події відслідковуються операційною системою, яка формує на кожен тип події відповідне повідомлення. Це повідомлення передається до додатку, на який додаток повинен реагувати відповідно до алгоритму своєї роботи. Тобто. GUI-додаток повинен вміти обробляти повідомлення операційної системи та вміти генерувати та надсилати свої власні повідомлення.

Програма, після запуску, створює своє вікно і запускає так званий

цикл обробки повідомлень Грубо кажучи, воно чекає на повідомлення від операційної системи. Роль програміста полягає у розробці коду з обробки повідомлень у разі виникнення будь-якої події.

Повідомлення передаються операційною системою саме до того додатку, якому воно призначене. Отже реалізується мультипро- граммность операційної системи, тобто. можливість одночасно запускати та працювати з декількома додатками.

У Lazarus обробка повідомлень замінена на обробку подій. При цьому Lazarus бере на себе всю рутинну роботу з розшифровки численних типів повідомлень та їх не менших параметрів. Таким чином, робота програміста значно полегшується. Програмістові достатньо вибрати ті події, на які реагуватиме його додаток і написати процедуру з обробки відповідної події.

У таблиці 6.1 наведено деякі події та умови, за яких вони виникають.

З таблиці 6.1. Для деякого вікна може бути створено дев'ять процедур обробки подій. Наприклад, процедура обробки подій OnCreate (створення вікна) може в цей час виконати деякі підготовчі операції, такі як відкриття файлів, ініціалізацію деяких змінних і т.д.

Зрозуміло, не обов'язково писати оброблювачі подій для всіх можливих подій. У цьому випадку, якщо відсутній обробник будь-якої події, ця подія просто не буде оброблена вашим додатком. Наприклад, якщо у програмі немає обробника події OnKeyDown, то на натискання клавіш на клавіатурі програма буде реагувати стандартним чином, наприклад при натисканні Alt+F4 вікно програми буде закрито.

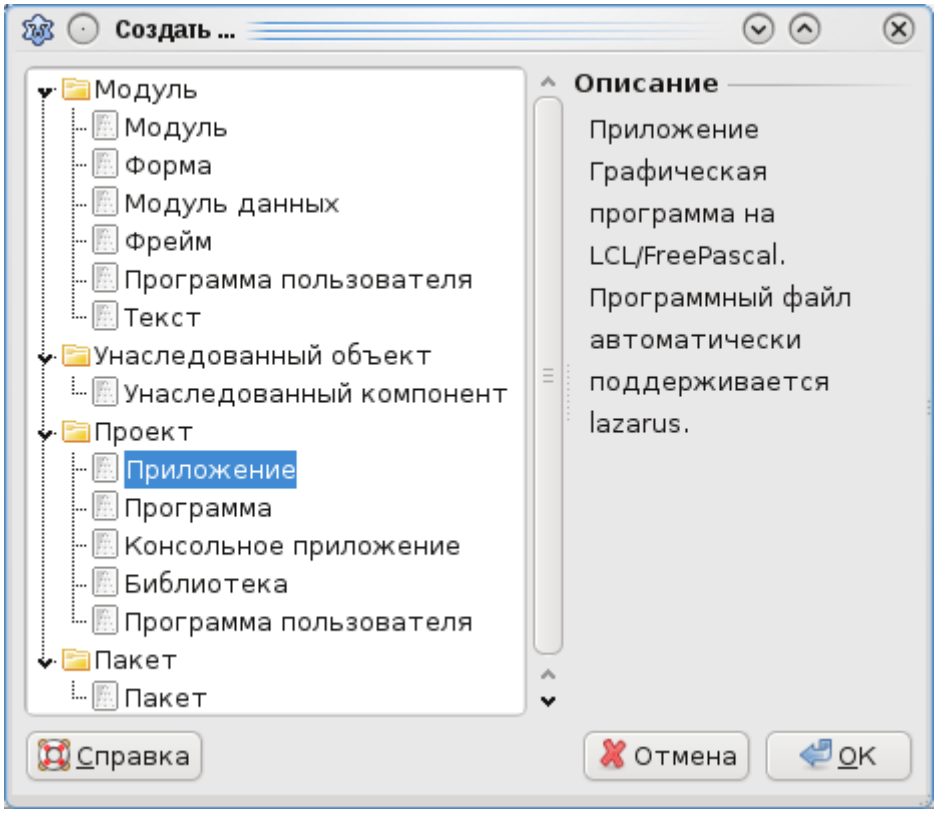
Таблиця 6.1

№ п/п	Подія	Коли виникає
1.	`OnCreate`	Подія виникає при створенні вікна і тільки один раз.
2.	`OnShow`	Ця подія генерується безпосередньо перед тим, коли вікно стане видимим.
3.	`OnActivate`	Ця подія генерується, коли вікно стає активним, тобто коли по-фокус введення.
4.	`OnPaint`	Це подія виникає щоразу під час перемалювання вікна.
5.	`OnReSize`	Це подія виникає щоразу при зміні розмірів вікна.
6.	`OnClose`	Подія виникає при закритий вікно.
7.	`OnKeyDown`	Подія настає при натискання будь-якої клавіші.
8.	`OnKeyPress`	Подія настає при натискання клавіші символу.
9.	`OnKeyUp`	Подія виникає привідпусканні будь-якої клавіши.

6.3. Візуальне програмування у середовищі Lazarus

6.3.1 Створення графічної програми

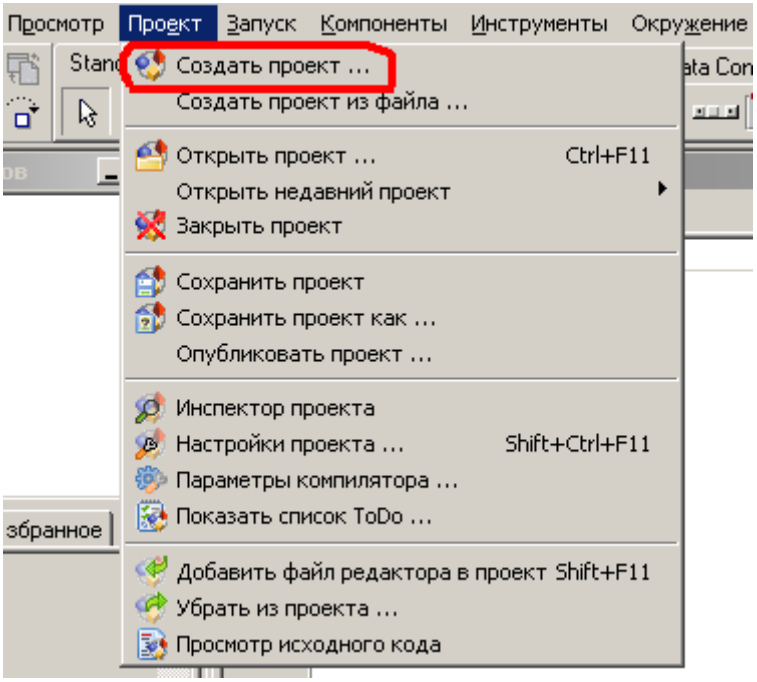
Для створення графічної програми можна у головному меню відкрити пункт Файл-> Створити. У вкладці Проект виберіть пункт Програма, мал. 6.7.



Мал. 6.7. Створення графічної програми

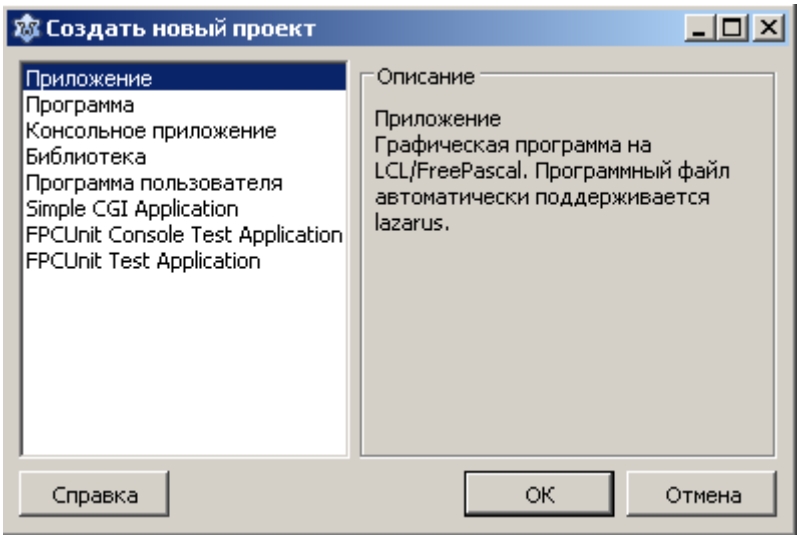
Натисніть кнопку ОК.

Другий спосіб. Виберіть пункт меню Проект, Створити проект… (рис. 6.8).



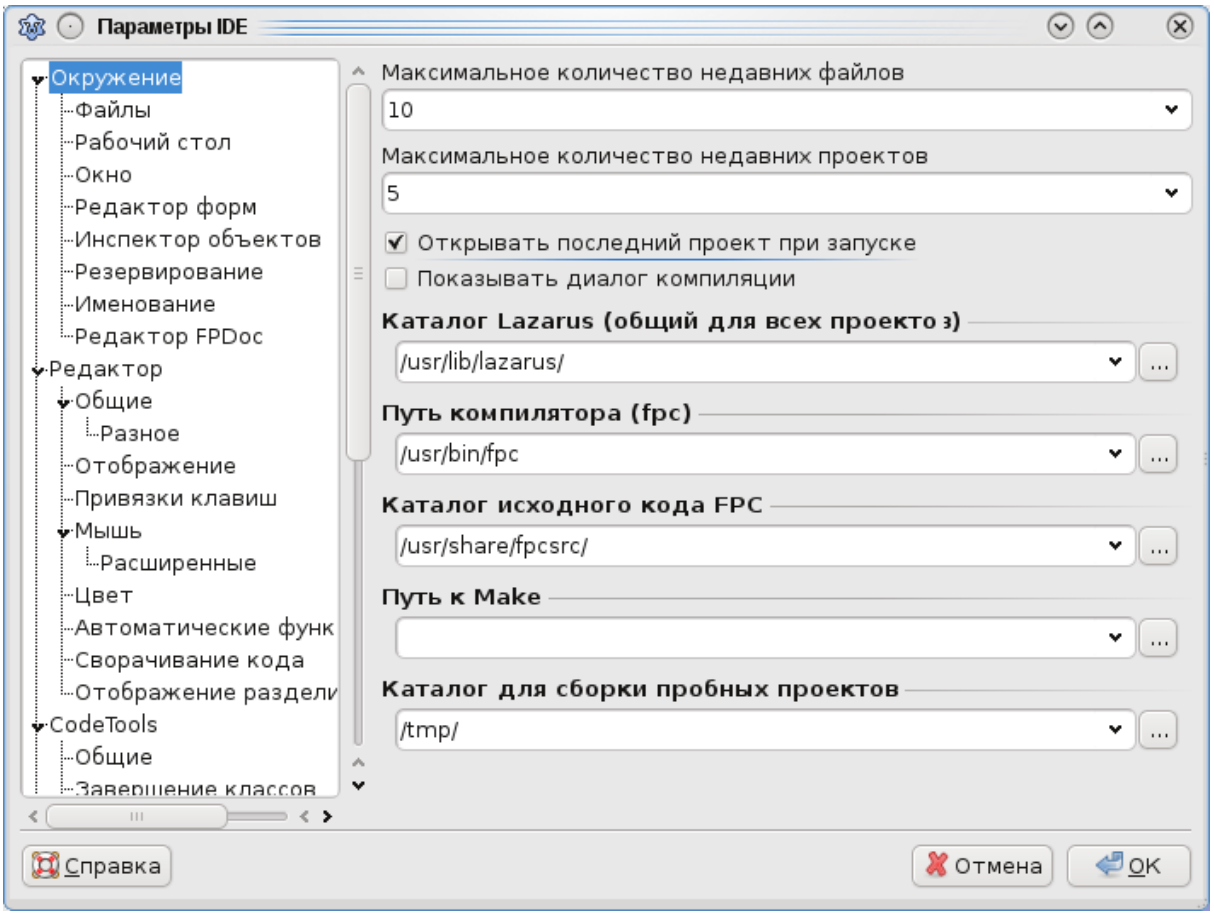
Мал. 6.8. Меню "Проект"

Виберіть Програму та натисніть ОК, мал. 6.9



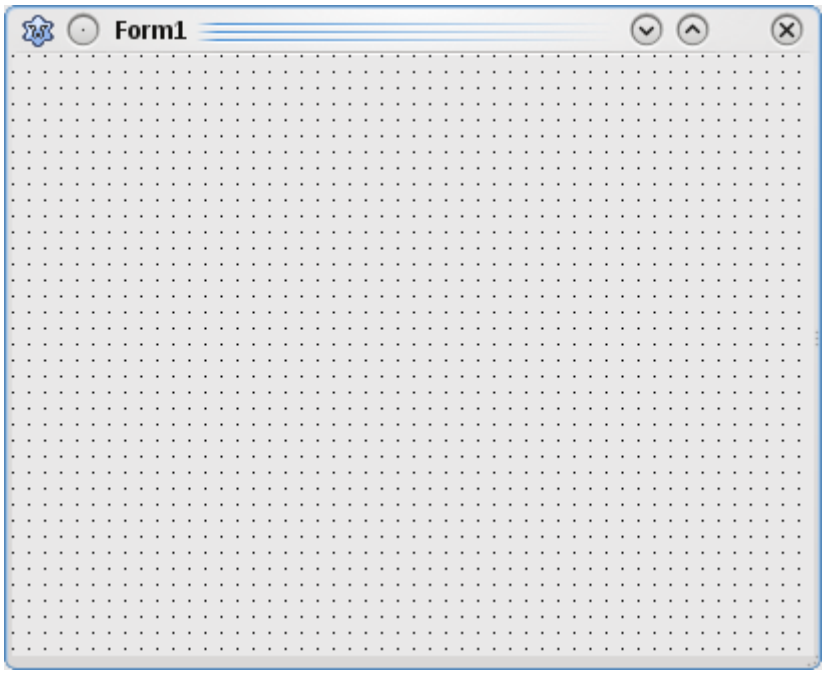
Мал. 6.9. Інший спосіб створення графічної програми

Крім того, якщо в налаштуваннях оточення (Оточення->Налаштування оточення->Файли) зняти галочку "Відкривати останній проект під час запуску" (мал. 6.10.), то Lazarus автоматично створюватиме новий проект GUI-додатка при кожному запуску.



Мал. 6.10. Вікно "Параметри IDE"

На екрані з'явиться пуста форма, рис. 6.11.



Мал. 6.11. Форма нового проекту

Якщо форми не видно, натисніть клавішу F12.

У вікні редактора вихідного коду Lazarus автоматично створить такий код:

```
unit Unit1;
{$mode objfpc}{$H+}
interface

uses
  Classes, SysUtils, LResources, Forms, Controls,
  Graphics, Dialogs;

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

var
```

```
    Form1: TForm1;

  implementation

  initialization
    {$I unit1.lrs}

  end.
```

Ми бачимо, що Lazarus створив для нас модуль із стандартним ім'ям Unit1. На тілі модуля створюється клас TForm1 заснований на базовому класі TForm і визначає об'єкт – стандартне графічне вікно.

Відразу ж збережіть свій проект у потрібній папці. Якщо потрібно, створіть нову папку. При збереженні пам'ятайте, що імена модуля та проекту не повинні збігатися, тобто файл проекту (lpr) та файл модуля (pas) повинні мати різні імена, тому що Lazarus надає ім'я модулю (у вихідному коді) таке саме, як і ім'я файлу модуля, а програмі на ім'я файлу проекту. Це необхідно зробити, інакше компілятор може згодом не знайти модуль посилання на нього у файлі проекту.

Вихідний код основної програми (проекту) буде збережено у файлі з ім'ям <Ім'я проекту>.lpr і має вигляд:

```
program project1;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms, Unit1, LResources
  { you can add units after this };
```

```
{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}

begin
  {$I project1.lrs}
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Тут у оголошенні uses перераховуються модулі, які підключаються до проекту за замовчуванням. Крім того, Lazarus автоматично ввімкнув ім'я щойно створеного модуля. За промовчанням це Unit1. Далі директивою

```
{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}
```

для операційної системи Windows включається файл опису ресурсів. Під ресурсами розуміються ресурси програми: піктограми, курсори, бітові образи та ін.

У частині програми, що виконується, міститься ще одна директива

```
{$I project1.lrs}
```

за допомогою якої підключається файл файлів Lazarus, що автоматично генерується. Зауважте, що це не файл ресурсів Windows.

Останні три оператори

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

реалізують звернення до методів об'єкта Application. В об'єкті Application зібрані дані та підпрограми, необхідні для нормального функціонування програми в середовищі операційної системи. Lazarus автоматично створює об'єкт-програму Application для кожного нового проекту. Метод Initialize відповідає за ініціалізацію (початкове налаштування) програми. Метод CreateForm створює головну форму програми Form1 (вікно програми). Після виклику методу Run здійснюється запуск нашої програми.

Без особливої потреби не слід редагувати код проекту. Тому під час створення проекту цей код не видно. Lazarus "приховує" цей код від зайво цікавих. Але, якщо "дуже хочеться", то можна подивитися його в меню Проект->Переглянути вихідний код проекту або відкрити файл проекту з розширенням .lpr будь-яким текстовим редактором.

А вихідний код нашої програми буде збережений у файлі <ім'я модуля>.pas (за замовчуванням Unit1.pas).

Відкомпілюйте та виконайте свою програму. Ви побачите пусте вікно. Ну, і що тут такого, скажете ви. За великим рахунком ви маєте рацію, нічого особливого. Але якщо вдуматися, то ми з вами щойно, без видимих зусиль, створили повноцінний додаток із графічним інтерфейсом! Вікно вашої програми має всі властивості стандартних графічних вікон. Його можна згорнути, можна розгорнути на весь екран, можна змінювати розміри. Вікно можна переміщувати у будь-яке місце на екрані. Як і будь-яке інше вікно, воно має рядок заголовка і системне меню. Не так уже й мало! І це на основі стандартного класу TForm. У Lazarus є чимало таких стандартних класів, на основі яких можна створювати програми практично будь-якої складності!

Як зазначалося, клас описує певний об'єкт. Щоб мати можливість звертатися в програмі до цього об'єкта, необхідно створити ек-

екземпляр класу. Це робиться за допомогою оголошення

```
var
  Form1: TForm1;
```

Тепер ми можемо працювати з цим об'єктом, звертаючись до нього на ім'я `Form1`.

6.3.2 Форма та її основні властивості

Під час створення нового проекту з'являється порожня форма (рис. 6.8.). Слід зазначити, що форма і вікно програми це не те саме. Форма – це те, що ви бачите під час проектування, а вікно – це те, що бачить користувач під час виконання вашої програми. Таким чином, за допомогою форми ви проектуєте вигляд вікна вашої програми. Крім того, формі відповідає клас, похідний від базового класу TForm.

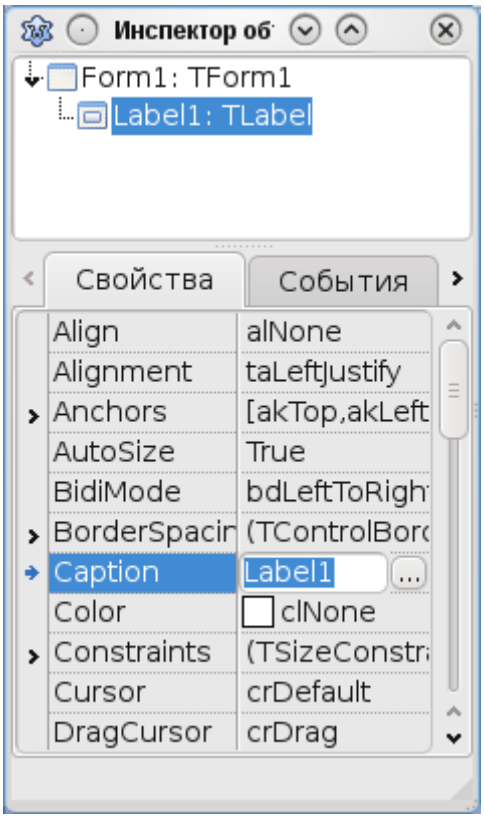
У розділі V ми з вами розглядали специфікатори доступу private, protected і public за допомогою яких можна керувати видимістю членів ззовні.

Є ще один специфікатор – published (опублікований). У цьому розділі містяться властивості, які може встановлювати на етапі проектування, але доступні і під час виконання, тобто. містять так звану RTTI - інформацію (run-time type information). Зазвичай у розділі published розміщуються лише властивості та події.

Усі компоненти Lazarus мають опублікований інтерфейс, який відображається в інспекторі об'єктів.

Розглянемо докладніше деякі властивості форми. Як ви вже зрозуміли, властивості форми, так само як і події, є опублікованою частиною інтерфейсу класу форми. Їх можна буде побачити в інспекторі об'єкт-

тов у вкладці Властивості, рис 6.12.



Мал. 6.12. Вкладка "Властивості" Інспектора об'єктів

Властивість Caption – заголовок вікна, є деяким текстом. За промовчанням надається значення "Form1". Бажано тут зазначати короткий зміст програми, наприклад "Розрахунок заробітної плати". Найчастіше розробники тут виводять назву програми та ім'я документа, пов'язаного з цією програмою.

Властивість Name – ім'я форми у програмі. За цим ім'ям можна звертатися до форми як об'єкта в програмі. За промовчанням надається ім'я Form1. Бажано давати осмислені імена, якщо в програмі є кілька форм. Найменування має підпорядковуватися вимогам мови Паскаль, тобто. Ім'я не повинно містити неприпустимі символи, прогалини тощо. (Див. розділ 2).

Властивість Left встановлює координати верхнього лівого кута вікна по горизонталі.

Властивість Top встановлює координати верхнього лівого кута вікна по вертикалі.

Самі розміри вікна задаються властивостями Height та Width. Розміри задаються у пікселах.

Положення вікна під час запуску визначається властивістю Position, воно може приймати такі значення:

poDesigned – положення вікна та його розміри залишаються такими самими, що й при проектуванні;

poDefault – положення вікна та його розміри визначається автоматично операційною системою;

poDefaultPosOnly – положення вікна визначається автоматично операційною системою, а розміри відповідають установкам під час проектування;

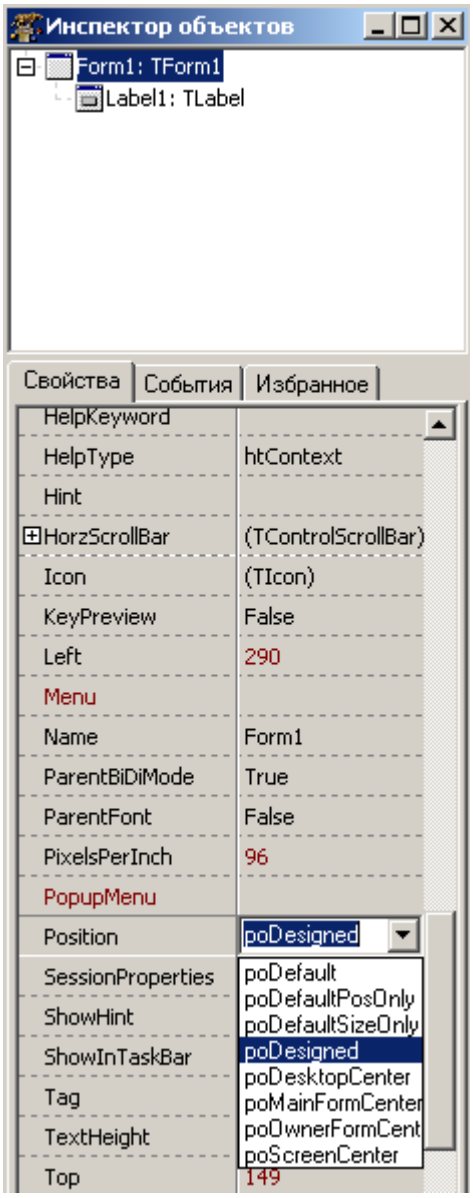
poDefaultSizeOnly – розміри вікна визначається автоматично операційною системою, а положення відповідають установкам під час проектування;

poScreenCenter або poDesktopCenter – вікно виводиться у центрі екрана, розмір визначається під час проектування;

poMainFormCenter – форма відображається в центрі головної форми, розмір визначається при проектуванні, якщо є лише одна головна форма, цей параметр відповідає poScreenCenter;

poOwnerFormCenter – форма відображається у центрі тієї форми, яка є власником цієї форми.

Щоб вибрати потрібне значення, необхідно клацнути на назву властивості. З'явиться список, що розкривається, з якого і можна вибрати необхідне значення, рис. 6.13.



Мал. 6.13. Можливі значення властивості "Position"

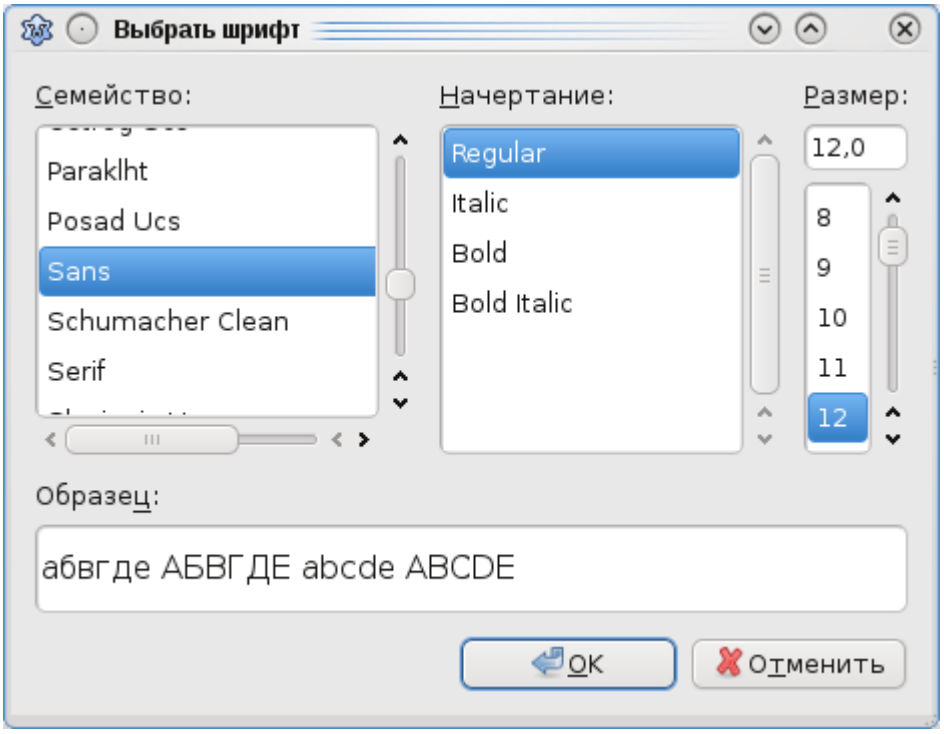
Керувати положенням форми на екрані можна також за допомогою властивості Align. Воно може набувати значень:

```
alNone – становище форми та її розміри не
змінюються; alBottom - форма розташована
внизу екрана; alLeft – форма
розташовується у лівій частині екрана;
alRight – форма розташовується у правій
частині екрана; alTop - форма
розташовується вгорі екрана;
```

```
alClient - форма займає весь екран;
```

Перелічені властивості відносяться до так званих простих властивостей, для завдання яких досить просто ввести одне необхідне значення або вибрати з списку, що розкривається, також тільки одне значення.

Є властивості, які називаються складовими чи складними. Вони помічаються в інспекторі об'єктів знаком "+" або кнопкою з трикрапкою. Наприклад, властивість Font є складовою. Складові якості, як випливає з назви, складаються з кількох значень. На рис. 6.14 показаний приклад завдання значень властивості Font.



Мал. 6.14. Вікно вибору шрифту (Linux)

З іншими властивостями форми ми знайомитимемося надалі за мірою необхідності.

Над об'єктами – екземплярами класу можна робити деякі дії, причому перелік дій визначений у самому класі. Жодних інших дій з ними робити не можна. Ці дії інакше називаються методами.

Наприклад, для класу форми існує метод Show – показати метод

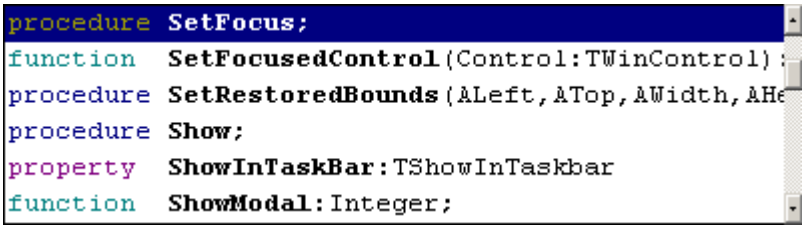
Hide – приховати, метод Close – закрити.

У редакторі вихідного коду Lazarus є чудовий інструмент CodeTools, за допомогою якого можна переглянути всі властивості та методи того чи іншого класу.

Подивимося, наприклад, властивості та методи класу TForm1. Для цього у редакторі вихідного коду після рядків

```
initialization
{$I unit1.lrs}
```

наберіть Form1. і трохи зачекайте. Перед вами з'явиться список всіх властивостей та методів класу TForm1, рис. 6.15.



Мал. 6.15. Вікно "Code Tools"

Користуючись підказками CodeTools, можна значно прискорити процес набору коду програми.

Форма може бути модальною та немодальною. Модальна форма це така форма, яка не дозволяє відкривати інші форми, доки вона сама не буде закрита. До таких форм найчастіше відносяться діалогові вікна. Щоб відобразити форму в модальному режимі, необхідно викликати метод ShowModal.

За замовчуванням головна форма програми відкривається у немодальному режимі.

6.3.3 Компоненти

Суть візуального програмування полягає в тому, що ви з набору компонентів бібліотеки LCL переносите на форму потрібні вам візуальні компоненти, налаштовуєте їх під власні потреби та формуєте дизайн вашої програми. Компоненти як і форма є деякими графічними об'єктами. І кожен компонент реалізований у вигляді класу. Наприклад, компонент TLabel (напис) реалізовано у вигляді класу. Назва компонента відповідає імені класу. Тобто коли ми говоримо про компонент TLabel, ми маємо на увазі клас TLabel.

Компоненти бувають видимими та невидимими. При проектуванні форма виступає як контейнер для компонентів. При цьому форму можна розмістити і невидимі компоненти.

Властивості та методи компонентів також відображаються в інспекторі об'єктів. Щоб побачити їх, достатньо виділити потрібний компонент на формі.

6.3.4 Обробники подій

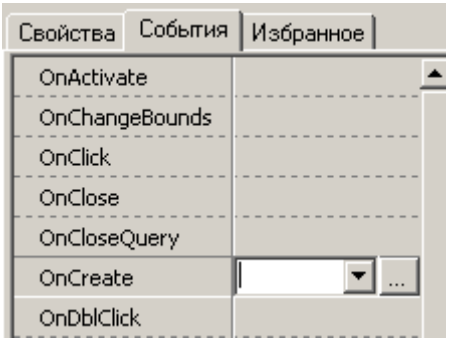
Властивості об'єкта визначають його зовнішній вигляд (розмір, шрифт, колір тощо), а сукупність подій визначають поведінковий бік об'єкта. Обробником події є процедура, яка виконує ті чи інші дії у відповідь на настання події. Тобто. за допомогою цієї процедури (обратника події) реалізується реакція об'єкта на подію, наприклад, на клацання миші.

Таким чином, завдання програміста зводиться до того, щоб визначити необхідні властивості об'єктів у його додатку та написати обробники тих подій, на які повинен реагувати той чи інший об'єкт додатку.

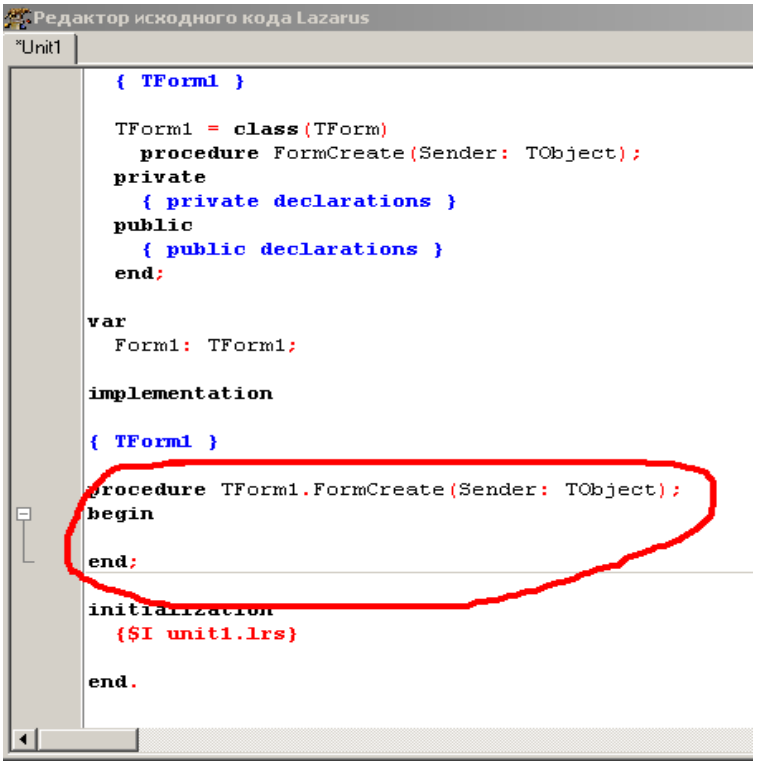
Інспектор об'єктів дозволяє визначити обробники подій, на які має реагувати форма або її компоненти. У вкладці Події в ле-

ній колонці наведено список всіх подій для даного об'єкта. Не обов'язково розробляти обробники для всіх подій. Як ми вже зазначали, якщо для деякої події відсутня його обробник, то додаток просто не реагуватиме на цю подію. Створіть новий проект у Lazarus. В інспекторі об'єктів відкрийте вкладку Події. Виберіть подію OnCreate, мал.

6.16. Ця подія виникає при створенні вікна програми. Клацніть на кнопки з трикрапкою. У редакторі коду з'явиться заготівля коду процедури обробника цієї події, рис. 6.17.



Мал. 6.16. Вкладка "Події"



Мал. 6.17. Заготівля коду процедури оброблювача події

Зверніть увагу, Lazarus автоматично надав процедурі ім'я FormCreate, приєднавши до нього ім'я класу TForm1. В інспекторі об'єктів також з'явилося ім'я процедури FormCreate – обробника події OnCreate.

Перейдіть до редактора вихідного коду, в обробнику події FormCreate введіть наступний код:

```
Form1.Caption:='Моя перша графічна програма';
```

Запустіть свою програму. Ви побачите, що у рядку заголовка вікна замість стандартного Form1, з'явився текст.

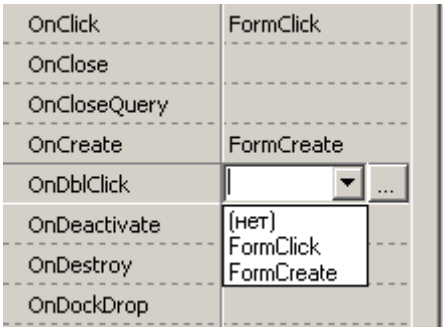
З цього прикладу ми можемо зробити дуже важливий висновок. Виявляється властивості об'єкта можна змінювати динамічно під час виконання програми. Для доступу до властивості об'єкта необхідно вказати ім'я цього об'єкта (у нашому випадку форми Form1) та через точку ім'я властивості (Caption).

Далі в інспекторі об'єктів виберіть подію OnClick. В обробнику події введіть код:

```
Form1.Caption:= 'Навіщо ти на мене натиснув?';
```

Запустіть програму. Клацніть на вікні вашої програми. Ви бачите, що у рядку заголовка вікна текст змінюється. Тобто. можна переконатися, що ваш додаток насправді реагує на натискання миші.

Виберіть тепер, наприклад, подію OnDblClick. Якщо ви розкриєте список, що розкривається, то ви побачите список вже наявних обробників подій, рис. 6.18.

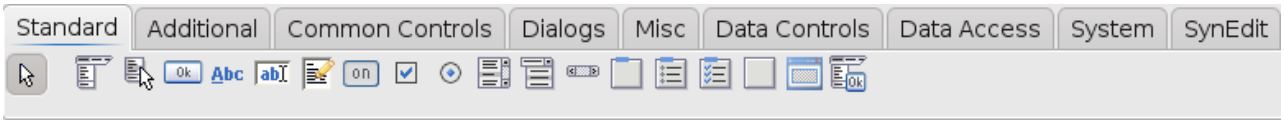


Мал. 6.18. Список вже наявних обробників подій

Якщо вам потрібно, щоб ваша програма реагувала на якусь подію так само, як і на якусь іншу подію, і обробник цієї події вже є, то ви можете просто вибрати зі списку потрібний вам обробник події. У нашому прикладі OnDblClick ви можете вибрати обробник FormClick або FormCreate. Цим можна уникнути невиправданого дублювання коду.

6.3.5 Найпростіші компоненти

Розглянемо деякі компоненти бібліотеки LCL Lazarus. Доступ до компонентів Lazarus організований через палітру компонентів, рис. 6.19.



Мал. 6.19. Палітра компонентів Lazarus

Палітра компонентів складається зі сторінок або вкладок, в яких знаходяться компоненти, згруповані за деякою ознакою. На малюнку 6.19 показано сторінку Standard, в якій розташовані найчастіше використовувані, так звані стандартні компоненти. За назвою сторінки можна зрозуміти, які компоненти знаходяться у тій чи іншій вкладці. Наприклад, зрозуміло, що на сторінці Dialogs знаходяться компоненти для організації діалогу.

га з користувачем.

Відповідно до призначення книги, ми не будемо розглядати всі компоненти (для цього необхідно писати нову книгу приблизно такого ж обсягу, що і ця!).

6.3.5.1. Компонент TLabel

Цей компонент, що його ще називають міткою або написом, дозволяє відобразити деякий текст на екрані. Основні властивості цього компонента:

- Name – ім'я в програмі, що використовується для доступу до компонента.
- Caption – текст, який відображатиметься у полі компонента.
- Color – колір написи фону.
- Font – це властивість є складовою. За допомогою цієї властивості можна налаштувати параметри шрифту для відображення тексту напису: колір літер (Color), розмір шрифту (Size), назва шрифту (Name) і т.д.
- Alignment – вирівнювання тексту напису. Тут taLeftJustify - по лівому краю, taCenter - по центру, taRightJustify - по правому краю.
- Height – висота напису пікселів.
- Width – ширина напису пікселів.
- AutoSize – автоматична зміна розміру напису залежно від довжини тексту. True – вертикальний та горизонтальний розмір напису залежить від довжини тексту, False – розміри компонента встановлюються вручну, вирівнювання тексту виконується властивістю Alignment.
- WordWrap – якщо має значення True, відбувається перенесення тексту за словами, тобто. текст напису відображається у декілька рядків, інакше текст виводиться в один рядок.

- Visible – визначає видимість компонента під час виконання програми. За умовчанням має значення True, тобто. компонент видно в момент

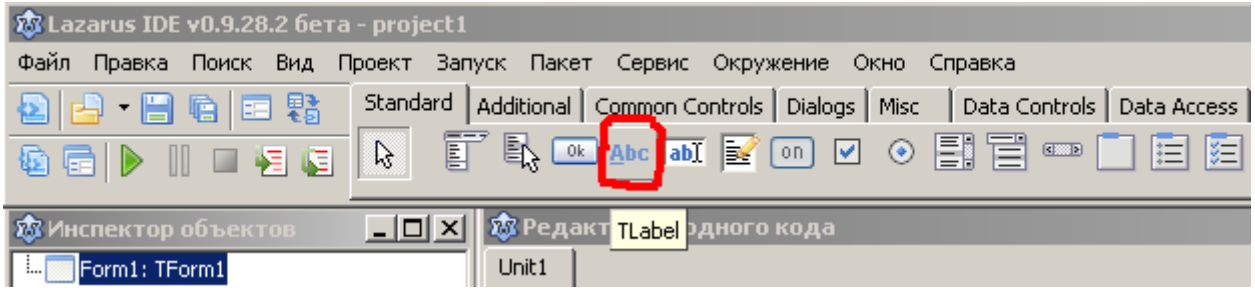
виконання програми.

Такі властивості компонента TLabel, що найчастіше використовуються. В інспекторі об'єктів ви знайдете ще безліч властивостей, розібратися з якими ви можете самостійно.

Найбільш важливими властивостями є Name і, звичайно, Caption.

У Caption ви задаєте текст напису, а в Name ім'я, яке використовуватиметься для доступу до компоненту в програмі.

Розглянемо приклад. Практично в будь-якій книзі, присвяченій програмуванню для початківців, розглядається приклад, якому навіть присвоєно ім'я – Hello World. Ця проста програма просто виводить текст привітання на екран. Чи не відступимо від цієї традиції і ми. Давайте напишемо цю програму. Запустіть Lazarus, створіть новий проект. Якщо ви в налаштуваннях оточення прибрали галочку "Відкривати останній проект під час запуску", то новий проект буде створено автоматично. Перенесіть на форму компонент TLabel. Це робиться дуже легко та просто. У палітрі компонентів на сторінці Standard (за замовчуванням відкрито цю сторінку) знайдіть компонент TLabel, рис. 6.20.

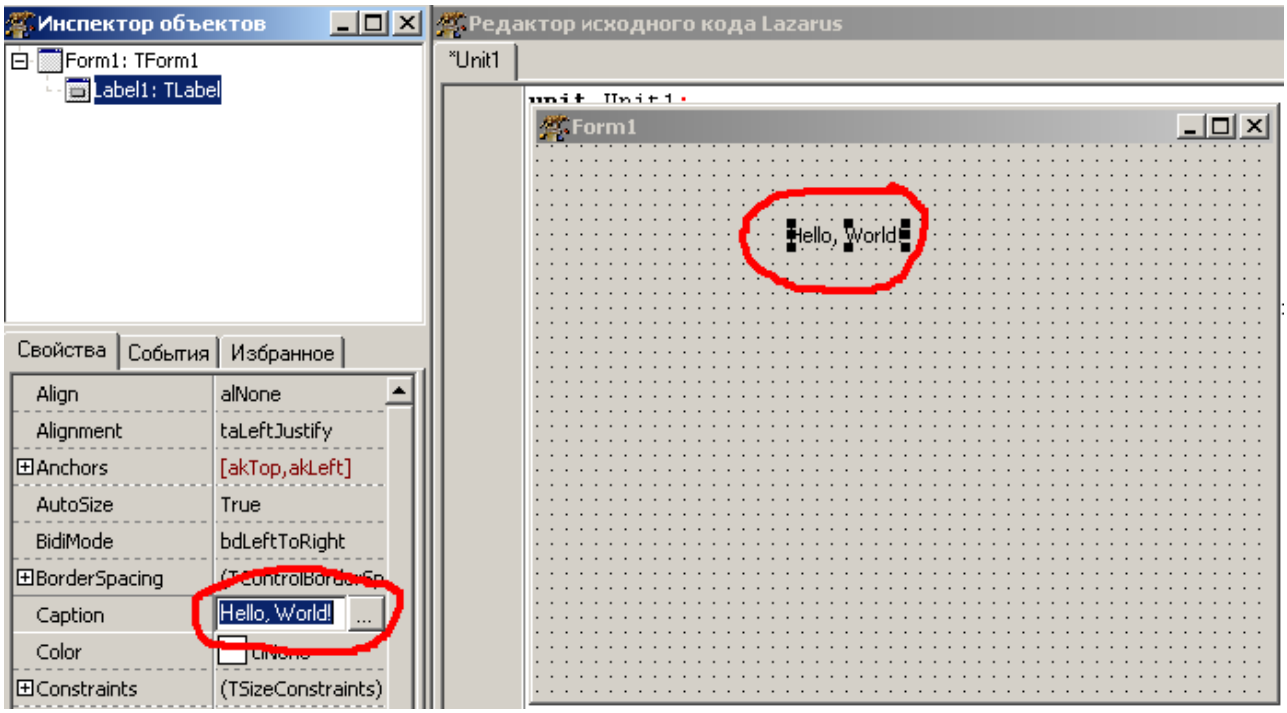


Мал. 6.20. Сторінка "Standard" палітри компонентів

Крім того, якщо трохи потримати покажчик миші над компонентом, виведеться підказка з ім'ям компонента. Натисніть на піктограму компонента, а потім натисніть на вільному місці форми. Вибраний компонент буде перенесений на форму. Він буде також автоматично ви-

діленим, тому всі його властивості будуть доступні в інспекторі об'єктів.

Натисніть на властивості Caption, його значення (за замовчуванням Label1) буде виділено синім кольором. Можете одразу набирати текст напису. Наберіть Hello, World! Цей текст з'явиться у вікні введення властивості Caption, а й одразу буде видно у полі тексту компонента, рис. 6.21.



Мал. 6.21. Приклад введення тексту напису

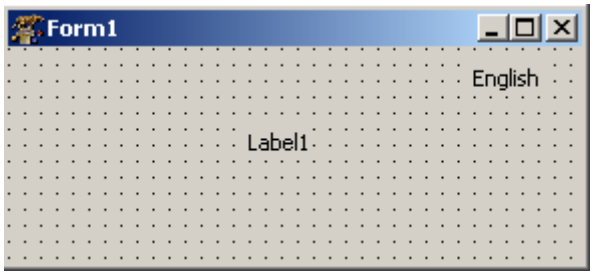
Подивимося, які зміни відбулися у коді порівняно з програмою з порожнім вікном (див. розділ 6.3.1). Ми бачимо, що Lazarus додав до опису класу форми TForm1 екземпляр класу TLabel – Label1, причому автоматично, без нашої участі! Як тільки ми додаємо на форму той чи інший компонент, Lazarus відразу додасть екземпляри класів відповідних компонентів, формуючи таким чином опис класу форми вашого додатка. Натисніть клавішу F9. Після компіляції та запуску вашої програми на екран буде виведено вікно з написом "Hello, World!", Мал. 6.22.

Мал. 6.22. Вікно графічного застосування з компонентом TMemo

Розглянемо ще одну цікаву властивість, яка досить часто застосовується. Це властивість Visible. За допомогою цієї властивості можна керувати видимістю компонента під час роботи програми.

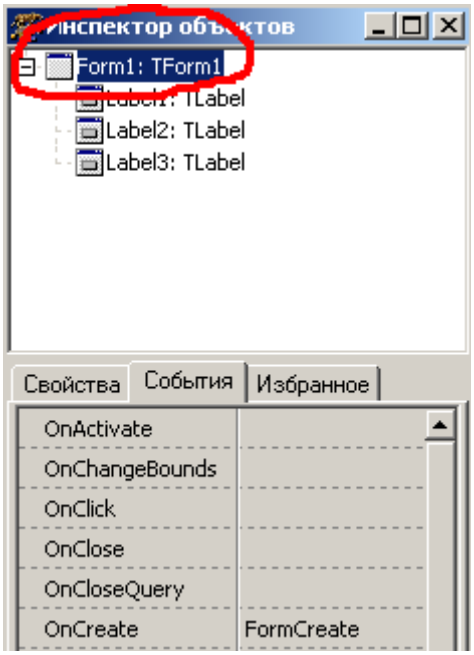
Давайте створимо додаток, в якому динамічно змінюватимемо властивості компонента, а також подивимося послідовність дій для розробки обробника подій.

Створіть новий проект. Перенесіть на форму три компоненти TLabel. Компонент Label1 розмістіть у центрі форми. Властивість Caption залиште без зміни. Компонент Label2 розмістіть у верхньому правому кутку форми, приблизно так, як показано на малюнку 6.23.



Мал. 6.23. Форма програми

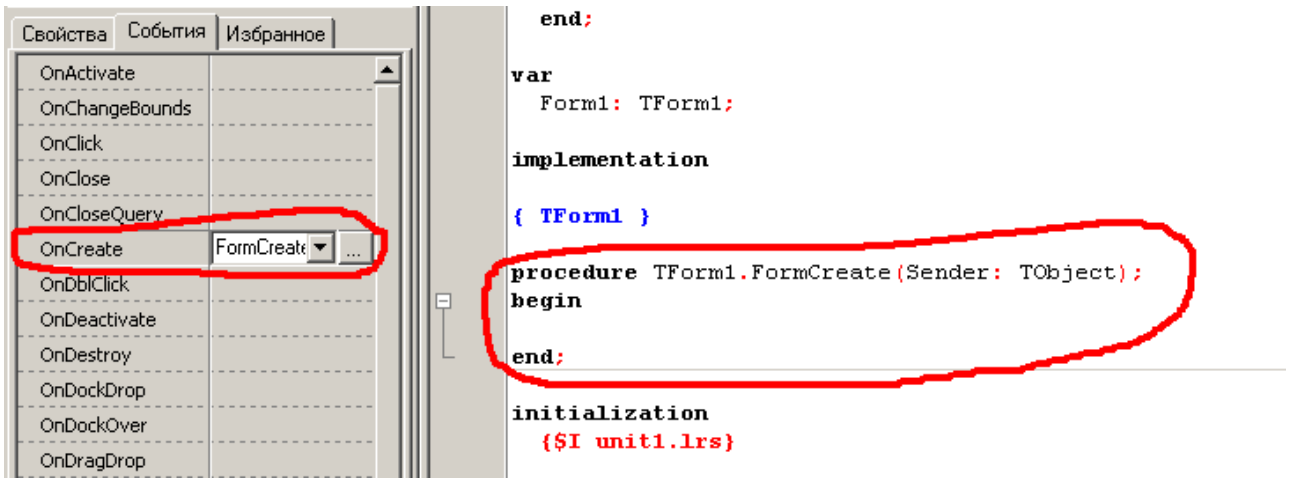
Властивість Caption змініть на "English". Компонент Label3 розмістіть прямо на компонент Label2 так, щоб він закривав його (виявляється так можна робити!). Властивість Caption змініть на "Українську". Виділіть форму. Найпростіше це зробити в інспекторі об'єктів, рис. 6.24.



Мал. 6.24. Інспектор об'єктів. Події форми

Перейдіть на вкладку Події, виберіть подію OnCreate. Натисніть кнопку з трикрапкою або двічі клацніть по полю з трикутником. Lazarus відразу додасть до опису класу форми відповідний метод, а в редакторі вихідного коду з'явиться заготівля для реалізації цього методу – обробника події, рис. 6.25.

Мал. 6.25. Заготівля обробника події



Оброблювач подій це звичайна процедура. У обробнику події OnCreate між операторами begin та end введіть наступний код (для зручності наводжу текст процедури повністю:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Label1.Caption:= 'Привіт, Світ!';
  Label2.Visible: = true;
  Label3.Visible:= false;
end;
```

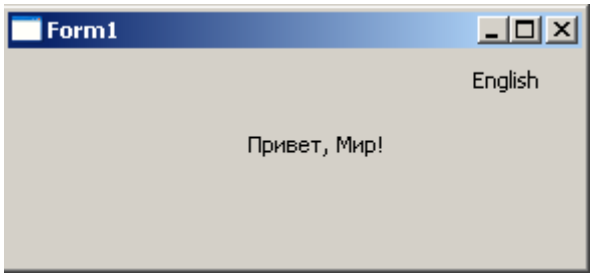
Виділіть компонент Label2, знову ж таки простіше буде через інспектор об'єктів. У вкладці Події виберіть подію OnClick. У його обробнику введіть код:

```
procedure TForm1.Label2Click(Sender: TObject);
begin
  Label2.Visible:= false;
  Label3.Visible: = true;
  Label1.Caption:= 'Hello, World!!';
end;
```

Тепер виділіть компонент Label3, виберіть подію OnClick. У його обробнику введіть код:

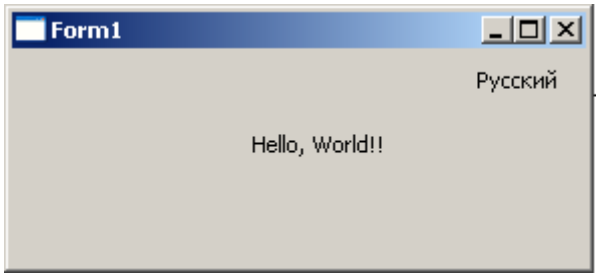
```
procedure TForm1.Label3Click(Sender: TObject);
begin
  Label2.Visible: = true;
  Label3.Visible:= false;
  Label1.Caption:= 'Привіт, Світ!';
end;
```

Запустіть програму, рис. 6.26.



Мал. 6.26. Вікно програми

Якщо натиснути на напис English, вітання виведеться англійською, а напис зміниться на Російський, рис. 6.27.



Мал. 6.27. Вікно програми після натискання напис "English"

Якщо ми знову натиснемо на напис Російська, текст привітання знову виведеться російською мовою, а текст напису зміниться на English.

Вам, мабуть, доводилося бачити програми, в яких реалізовано багатомовний інтерфейс. Перемикання з однієї мови на іншу в них реалізовано приблизно за таким же методом, який ми з вами щойно реалізували. За традицією, що вже встановилася нашої з вами, подивимося ще раз код програми і спробуємо знайти недоліки. Цей код настільки простий, що, здається, тут нічого й покращувати. Загалом, ви маєте рацію. Єдине зауваження, яке можна зробити скоріше стосується стилю програмування. Я маю на увазі методи іменування об'єктів у програмі. Є багато різних рекомендацій при виборі імен об'єктів. Зокрема, для іменування змінних багато хто використовує так звану угорську нотацію, тобто правила написання імен змінних. Сенс цих правил у тому, щоб на ім'я змінної можна було визначити її тип. Для цього використовують пре-

Фікси, наприклад, у багатьох наших програмах для позначення покажчиків ми використовували префікс "p" (pHead, pCurrent і т.д.). Виходячи з цього, для змінних цілого типу можна застосовувати префікс "i", для змінних логічного типу префікс "b" і т.д.

Для компонентів також можна використовувати префікси, наприклад для `TLabel` доречно використовувати префікс `"lb"`.

Існують і інші способи іменування, тим більше, що є чимало і затятих противників угорської нотації. Не стану їх описувати. За бажанням ви можете знайти все це в Інтернеті.

У принципі, якихось жорстких вимог щодо цього немає. Головне, ви повинні знайти, виробити свій, власний стиль, зрозумілий не лише вам, а й іншим. Ви використовуватимете префікси, не використовуватимете, це ваша справа. Основний принцип тут – зрозумілість, тобто всі позначення мають бути зрозумілими і не лише вам, а й іншим.

Уявіть собі, що ви написали якусь досить велику програму. Через кілька місяців, а можливо років, вам чи комусь іншому (звичайно, з вашого дозволу!) знадобилося її модернізувати. Якщо у вашій програмі будуть позначення, типу "aaa" або "x123", то у вашій програмі буде важко розібратися не тільки тому іншому, а й вам самому!

До хорошого стилю також належить наявність у програмі коментарів. І тут не повинно бути крайнощів. Сухі односкладові коментарі типу "x – це змінна" одна крайність. Численні, багатослівні коментарі чи не після кожного оператора інша крайність. Важливо знайти золоту середину!

Не забувайте також про використання відступів під час запису коду програми. У всіх наших програмах цієї книги ми використовували відступи та коментарі.

Повернімося до нашої програми. Тут використовуються три компоненти `TLabel`. Lazarus присвоїв їм за умовчанням імена `Label1`, `Label2` та

Label3. Зрозуміло, що це є написи. Тому застосування слова Label цілком логічно та виправдано. Але далі йде проста нумерація! А якщо у програмі 10, 20, 100 написів? Незабаром ви заплутаєтеся, що означає, наприклад Label5 і, наприклад, Label12.

Тому при невеликій кількості однакових компонентів цілком допустимо використовувати імена за умовчанням, але все ж таки краще, особливо якщо кількість однакових компонентів більше трьох, вибирати інформативні імена, які відображають суть того чи іншого об'єкта. При цьому не бійтеся вигадувати імена у російському контексті, хоча записувати їх доводиться англійськими літерами. Наприклад, ім'я Stroka буде зрозуміле будь-якому російськомовному програмісту. Хоча, якщо ви працюватимете, ну скажімо, у Microsoft, то за цей ідентифікатор вас трохи пожурять і попросять змінити на інше, більш зрозуміле ім'я!

Виходячи з цих міркувань, перепишемо нашу програму у вигляді:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   LResources,   Forms,
             Controls,Graphics, Dialogs, StdCtrls,
  Windows;
type
  {TForm1}
  TForm1 = class(TForm)
    L_Hello: TLabel; // тут виводитиметься саме вітання
    L_English: TLabel; // для перемикання англійською
    мовою L_Russian: TLabel; // для перемикання
    російською мовою procedure FormShow(Sender: TObject);
```

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

```
    procedure L_RussianClick(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{TForm1}
procedure TForm1.FormShow(Sender: TObject);
begin
  L_Hello.Caption:= 'Привіт, Світ!';
end;

procedure TForm1.L_EnglishClick(Sender: TObject);
begin
  L_English.Visible:= false;
  L_Russian.Visible:= true;
  L_Hello.Caption:= 'Hello, World!';
end;

procedure TForm1.L_RussianClick(Sender: TObject);
begin
  L_English.Visible:= true;
  L_Russian.Visible:= false;
  L_Hello.Caption:= 'Привіт, Світ!';
end;
initialization
```

```
  {$I unit1.lrs}
end.
```

Повторюю, ви вільні вибирати будь-які імена для своїх об'єктів у програмі, зокрема, і тут ви можете вигадати інші імена для компонентів, більш логіч- ні, інформативні та зрозумілі на ваш погляд. Цей приклад лише ілюстрація до того, що було сказано вище.

Чи є відмінності під час створення власних класів у програмах з графічним інтерфейсом проти консольними додатками?

Давайте реалізуємо останній приклад із розділу V з конструкторами та віртуальними функціями у вигляді GUI-додатку. Для цього створіть новий проект і помістіть на форму чотири компоненти TLabel. Код програми:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls,
  Unit2;
type
  {TForm1}
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    procedure FormShow(Sender: TObject);
  private
    {private declarations}
```

```
  public
    { public declarations }
  end;

type
  TStudent = class(THuman)
    private
      group: string;
    public
      constructor Create(gr: string);
      function GetData: string;
      функція Status: string; override;
  end;

type
  TProfessor = class(THuman)
    private
      kafedra: string;
    public
      constructor Create(kaf: string);
      function GetData: string;
      функція Status: string; override;
  end;
var
  Form1: TForm1;
  Student: TStudent;
  Professor: TProfessor;
  fname: string;
```

```
implementation
function TStudent.Status: string;
begin
  Result:= '- студент';
end;

constructor TStudent.Create(gr: string);
begin
  inherited Create('Андрій', 'Аршавін');
  group: = gr;
end;

function TStudent.GetData: string;
begin
  Result:= inherited GetData + ', група '+ group;
end;

constructor TProfessor.Create(kaf: string);
begin
  inherited Create('Олексій', 'Попов');
  kafedra: = kaf;
end;

function TProfessor.Status: string;
begin
  Result:= '- викладач';
end;
function TProfessor.GetData: string;
begin
```

```
  Result:= inherited GetData + ', кафедра '+
kafedra; end;

{TForm1}

procedure TForm1.FormShow(Sender: TObject);
begin
  Student:= TStudent.Create(''Арсенал'');
  fname: = Student.GetData;
  Label1.Caption:= 'Це:' + fname;
  Professor:= TProfessor.Create('"Телеканал Росія 2"');
  fname:= Professor.GetData;
  Label2.Caption:= 'Це:' + fname;
  Student.name:= 'Віталій';
  Student.fam:= 'Петрів';
  Student.group:= '"ПОВТАС-1/09"';
  fname: = Student.GetData;
  Label3.Caption:= 'Це:' + fname;
  Professor.name:= 'Іван';
  Professor.fam:= 'Іванів';
  Professor.kafedra:= '"Програмування"';
  fname:= Professor.GetData;
  Label4.Caption:= 'Це:' + fname;
  Student.Free;
  Professor.Free;
end;
initialization
  {$I unit1.lrs}
```

```
end.

unit Unit2;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
  type

  {THuman}

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      constructor Create(n, f: string);
      function GetData: string;
      функція Status: string; virtual; abstract;
   end;
implementation
{THuman}
конструктор THuman.Create(n, f: string);
begin
  Fname: = n;
```

```
  Ffam: = f;
end;

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.GetData: string;
begin
  Result:= name + '' + fam + Status;
end;
end.
```

Ми бачимо, що ніякої різниці немає, за винятком того, що реалізації методів класів слід поміщати в розділ implementation модуля.

6.3.5.2. Кнопки TButton, TBitBtn та TSpeedButton

Кнопка є елементом керування, призначеним для запуску певних дій або команд. Клацніть по кнопці мишею викликає подію OnClick в обробнику якого програміст і ініціює виконання ка-

будь-яких дій, команд та процедур. На панелі компонентів є два різновиди кнопок. На сторінці Standard є компонент TButton, а на сторінці Additional компонент TBitBtn.

Розглянемо властивості, властиві лише кнопці. Такі його властивості, як Name, Left, Top, Height, Width, Font, Visible та ін мають той же сенс, що і для розглянутого раніше компонента TLabel.

- Caption – текст, який відображається на кнопці.

- Enabled – ознака доступності кнопки. За промовчанням має значення True, тобто кнопка доступна. Якщо False, то кнопка на даний момент недоступна. Напис на кнопці має бляклий вигляд, натискання на кнопку не призводить до жодних дій, навіть якщо є обробник події OnClick.

- Default – якщо встановлено значення True, натискання клавіші Enter

буде еквівалентно клацанню по кнопці мишею.

- Cancel – якщо встановлено значення True, то натискання клавіші Esc буде еквівалентно клацанню кнопки мишею.

Компонент TBitBtn відрізняється від TButton тим, що на ньому можна відображати піктограми. Наведені вище властивості для TButton мають місце і для TBitBtn. Крім цього, цей компонент має і свої особливі властивості.

- Kind – задає тип кнопки. Є кілька визначених типів кнопки з готовою піктограмою та текстом:
  - ✓ bkAbort – з текстом "Прорвати".
  - ✓ bkAll – з текстом "Все".
  - ✓ bkCancel – з текстом "Скасувати".
  - ✓ bkClose – з текстом "Закрити".
  - ✓ bkCustom – довільний текст, який встановлюється програмістом.
  - ✓ bkHelp – з текстом "Довідка".
  - ✓ bkIgnore – з текстом "Пропуск".

- ✓ bkNo – з текстом "Ні".
- ✓ bkNoToAll – з текстом "Ні для всіх".
- ✓ bkOK – з текстом "ОК".
- ✓ bkRetry – із текстом "Повтор".
- ✓ bkYes – з текстом "Так".
- ✓ bkYesToAll – з текстом "Так для всіх".

• Glyph – якщо вас не влаштовують малюнки, ви можете вибрати інші. Відкриється діалогове вікно, необхідно вказати шлях до цього малюнку.

• Margin – визначає відстань від краю кнопки до малюнка (у пікселах). За промовчанням -1. У цьому випадку малюнок та текст розташовуються в центрі.

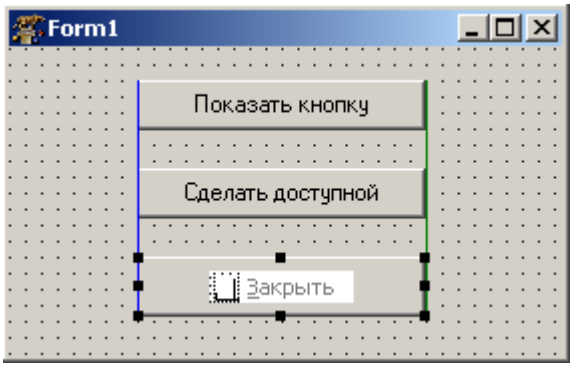
• Layout - Визначає положення малюнка на кнопці. Можна вибрати:

- ✓ blGlyphLeft – ліворуч.
- ✓ blGlyphRight – справа.
- ✓ blGlyphBottom – знизу.
- ✓ blGlyphTop – зверху.

• Spacing – задає відстань у пікселах між малюнком та текстом кноп-

ки.

Розглянемо приклад. Покладіть на порожню форму дві кнопки TButton та одну TBitBtn. Для кнопки Button1 встановіть властивість Caption="Показати кнопку", для кнопки Button2 встановіть властивість Caption="Зробити доступною". Для кнопки BitBtn1 встановіть властивість Kind = bkClose, Enabled = False, Visible = False, рис. 6.28.



Мал. 6.28. Форма застосування

В обробник події OnClick для Button1 введіть код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  BitBtn1.Visible:= true;
end;
```

В обробник події OnClick для Button2 введіть код:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  BitBtn1.Enabled:= true;
end;
```

Відкомпілюйте та запустіть програму. Ви бачите, що кнопка BitBtn1 відразу після запуску не видно. Натисніть кнопку "Показати кнопку". Кнопка BitBtn1 буде видно, але недоступна. Тепер натисніть кнопку "Зробити доступною". Кнопка BitBtn1 стане доступною. Закрийте за допомогою програми.

Кнопки TSpeedButton (вона знаходиться на сторінці Additional) зазвичай використовуються як швидкі кнопки в панелях інструментів. на

їх також можна помістити деяке зображення. Зрозуміло, їх можна використовувати як звичайні кнопки. Відмінною особливістю цих клавіш є можливість фіксації натиснутого стану. Для цього використовуються властивості GroupIndex та AllowAllUp.

За промовчанням властивість GroupIndex має значення 0. У цьому випадку кнопка поводиться як звичайна кнопка. Якщо значення GroupIndex > 0, то при натисканні на кнопку вона залишиться у натиснутому стані. Якщо властивості AllowAllUp присвоїти значення = true, то при повторному натисканні на кнопку вона повернеться до нормального стану.

Якщо привласнити властивості GroupIndex кількох кнопок однакове значення > 0, то при натисканні наступної кнопки вона залишиться в натиснутому стані, а попередня кнопка повернеться у звичайний віджатий стан (властивість AllowAllUp має мати значення = true для всіх цих кнопок).

Дізнатися в якому стані знаходиться кнопка (натиснутою або віджатий) можна за якістю Down. Якщо Down = true, то кнопка натиснутому стані.

6.3.6 Організація введення даних. Однорядкові редактори TEdit, TLabeledEdit

Будь-яка програма у своїй роботі використовує якісь вихідні дані. Найчастіше ці дані вводяться користувачем із клавіатури.

Для організації введення використовуються компоненти TEdit, TLabeledEdit та ін. Сигналом завершення введення та початку обробки введених даних є зазвичай натискання користувачем кнопки. Для завершення введення також часто використовується клавіша Enter.

6.3.6.1. Компонент TEdit

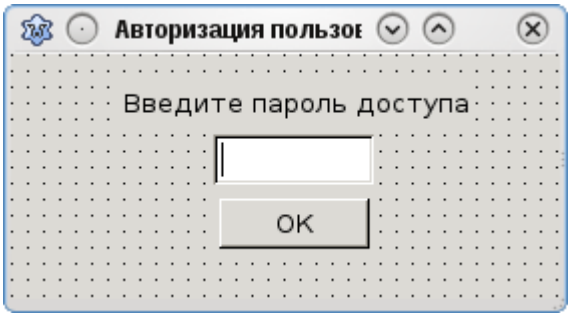
За допомогою цього компонента можна вводити та редагувати деякий

текст. Цей компонент носить також назву однорядковий редактор.

- Text – значення, що вводиться, міститься в цій властивості у вигляді рядка символів. За промовчанням містить рядок "Edit1".
- AutoSelect – якщо встановлено значення True, то під час передачі фокуса на компонент, весь текст у полі редагування буде виділено.
- ReadOnly – якщо встановлено значення True, то текст у полі редагування доступний лише для читання, при цьому зберігається можливість копіювання тексту до буфера обміну.

Розглянемо приклад, консольний варіант якого ми розглядали розділ 3.3.1.3. Зробимо цей приклад у вигляді графічного докладання. Як ви пам'ятаєте, приклад імітував випадок авторизації користувача для входу в систему.

Перенесіть форму компонент TEdit, очистіть властивість Text. Також нам знадобляться кнопка та напис. Розташуйте їх на формі так, як показано на малюнку 6.29, відповідно змінивши властивості Caption форми, написи та кнопки.



Мал. 6.29. Форма застосування

У редакторі вихідного коду в обробник події OnCreate форми введіть наступний код:

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

```
begin
  passw:= '1234';
```

```
  n: = 1;
end;
```

А в обробник OnClick кнопки наступний код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if passw <> Edit1.Text then
  begin
    Edit1.SetFocus;
    if n = 1 then
    begin
      ShowMessage('Неправильний пароль!');
      inc(n);
    end
    else
    begin
      if n = 3 then
      begin
        ShowMessage('Вам відмовлено у
        доступі'); Close();
      end
      else
      begin
        Stroka:= 'Ви' + IntToStr(n) + 'рази ввели
неправильний пароль';
        Stroka:= Stroka + ' Після 3-ї спроби вам буде
        відмовлено в
доступі';
        ShowMessage(Stroka);
```

```
        inc(n);
      end
    end
  end
  else
  begin
    ShowMessage('Ви успішно
    авторизувалися!'); Close();
  end;
end;
```

У розділі опису змінних модуля, додайте наступні описи змінних:

```
var
  Form1: TForm1;
  passw: string[4];
  Stroka: string;
  n: integer;
```

Зверніть увагу – щоб введені змінні були доступні в обробниках, ми помістили їх оголошення в розділі interface.

Запустіть програму і поекспериментуйте з введенням різних паролей. Правильний пароль "1234".

У програмі ми використовували функцію IntToStr(n), яка переводить ціле число у її рядкове представлення та стандартну процедуру ShowMessage для виведення повідомлення з кнопкою ОК. Єдиним параметром цієї процедури є рядок символів.

Чи можна покращити програму?

Багато користувачів звикли завершувати введення в однорядковому редакторі

натисканням клавіші Enter. Для того, щоб реалізувати цю можливість, просто встановіть у кнопки властивість Default рівним True в інспекторі об'єктів. Проте врахуйте, якщо у вас у програмі кілька кнопок, то спрацює та кнопка, яка має у цей момент фокус уведення. Відстежувати фокус при великій кількості компонентів стає досить обтяжливим. Вихід полягає в написанні обробника компонента Edit1 на подію OnKeyPress. Цей обробник має параметр Key, якому система передає код натиснутої клавіші. Код клавіші Enter дорівнює #13 (див. табл. 3.4. розділ 3.3.1). Звідси обробник виглядатиме так:

```
procedure TForm1.Edit1KeyPress(Sender: TObject;
                                var Key: char);
begin
  if Key <> #13 then exit;
    if passw <> Edit1.Text then
  begin
    Edit1.SetFocus;
    if n = 1 then
    begin
      ShowMessage('Неправильний пароль!');
      inc(n);
    end
    else
    begin
      if n = 3 then
      begin
        ShowMessage('Вам відмовлено у
        доступі'); Close();
      end
```

```
      else
      begin
        Stroka:= 'Ви' + IntToStr(n)+ 'разу ввели
        неправильний
пароль';
        Stroka:= Stroka + ' Після 3-ї спроби вам буде
        відмовлено в
доступі';

        ShowMessage(Stroka);
        inc(n);
      end
    end
  end
  else
  begin
    ShowMessage('Ви успішно
    авторизувалися!'); Close();
  end;
end;
```

Від обробника Button1Click код відрізняється лише одним першим оператором

```
    if Key <> #13 then exit;
```

На жаль, тут має місце майже повне дублювання коду. Щоб уникнути цього, створимо процедуру і в обробниках просто її викликатимемо. Остаточно програма матиме вигляд:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
```

```
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs,
  StdCtrls;

type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure FormCreate(Sender: TObject);
    procedure avtorization;
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;
  passw: string[4];
  Stroka: string;
  n: integer;

implementation
{TForm1}
```

```
procedure TForm1.avtorization;
begin
  if passw <> Edit1.Text then
  begin
    Edit1.SetFocus;
    if n = 1 then
    begin
      ShowMessage('Неправильний пароль!');
      inc(n);
    end
    else
    begin
      if n = 3 then
      begin
        ShowMessage('Вам відмовлено у
        доступі'); Close();
      end
      else
      begin
        Stroka:= 'Ви' + IntToStr(n) + 'рази ввели
неправильний пароль';
        Stroka:= Stroka + ' Після 3-ї спроби вам буде
        відмовлено в
доступі';
        ShowMessage(Stroka);
        inc(n);
      end
    end
  end
  else
```

```
  begin
    ShowMessage('Ви успішно
    авторизувалися!'); Close();
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  passw:= '1234';
  n: = 1;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  avtorization;
end;
procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key <> #13 then exit;
  avtorization;
end;
initialization
  {$I unit1.lrs}

end.
```

6.3.6.2. Компонент TLabeledEdit

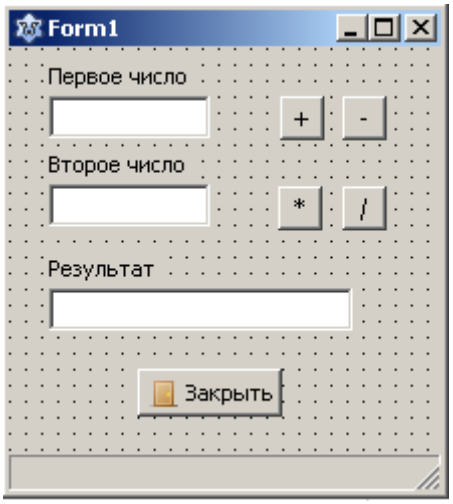
На панелі компонентів на сторінці Additional є ще один різновид однорядкового редактора TLabeledEdit. Це фактично той же

самий однорядковий редактор, але з приєднаним до нього написом. У багатьох випадках для зручності користувача над вікнами редагування необхідно поміщати деякий текст, який пояснює, що за інформацію слід вводити користувачеві в цьому вікні. У цих випадках зручніше використовувати TLabeledEdit.

Досить часто необхідно організовувати введення числової інформації. При цьому завжди необхідно пам'ятати, що дані вводяться у символьному вигляді, тобто у властивості Text компонентів TEdit та TLabeledEdit міститься рядок символів. Якщо введено число, то для роботи з ним як із числом, необхідно цей рядок символів перетворити на внутрішнє уявлення числа. Для цього використовуються функції StrToInt – для перетворення рядка на ціле число та StrToFloat – для перетворення на речове число. З іншого боку, щоб вивести числові дані, наприклад результати обчислень, необхідно виконати зворотне перетворення чисел у їх рядкове представлення. Для цього застосовуються функції IntToStr – для перетворення цілого числа на рядок і FloatToStr – для перетворення речовинного числа на рядок.

Розглянемо приклад.

Створіть новий проект. Помістіть на форму три компоненти TLabeledEdit, чотири кнопки TSpeedButton, одну кнопку TBitBtn і компонент TStatusBar, розташовану на сторінці Common Controls, приблизно так, як показано на рис. 6.30.



Мал. 6.30. Форма застосування

Для LabeledEdit1 розкрийте властивість EditLabel та встановіть властивість Caption = "Перше число". Так само для LabeledEdit2 установіть властивість Caption = "Друге число" і для LabeledEdit3 Caption = "Результат".

Для SpeedButton1 встановіть властивість Caption = "+". Для SpeedButton2 властивість Caption = "-", для SpeedButton3 Caption = "*" і SpeedButton2 властивість Caption = "/".

Встановіть для всіх кнопок SpeedButton властивості GroupIndex=1, властивості AllowAllUp=true.

У обробник події OnShow для Form1 введіть код:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  LabeledEdit1.SetFocus;
end;
```

У обробник події OnKeyPress для LabeledEdit1 введіть код:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
```

```
begin
  if Key = #13 then
  begin
    LabeledEdit2.SetFocus;SpeedB
    utton1.Down:= false;
    SpeedButton2.Down:= false;
    SpeedButton3.Down:= false;
    SpeedButton4.Down:= false;
    exit;
  end;
end;
```

У обробник події OnClick для SpeedButton1 введіть код:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
              + StrToInt(LabeledEdit2.Text));
  StatusBar1.SimpleText:= 'Складання';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

В обробник події OnClick для SpeedButton2 введіть код:

```
procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
              - StrToInt(LabeledEdit2.Text));
```

```
  StatusBar1.SimpleText:= 'Віднімання';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

В обробник події OnClick для SpeedButton3 введіть код:

```
procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
              * StrToInt(LabeledEdit2.Text));
  StatusBar1.SimpleText:= 'Умноження';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

У обробник події OnClick для SpeedButton4 введіть код:

```
procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                    div StrToInt(LabeledEdit2.Text));
  StatusBar1.SimpleText:= 'Поділ націло';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

Тут новим для нас є компонент TStatusBar, який дозволяє виводити у вікні так званий рядок стану. У цьому рядку додат-

ня зазвичай виводить різну допоміжну інформацію. У нашій програмі буде виводитись інформація про останню виконану операцію.

У рядку стану можна визначити одну чи кілька незалежних панелей. Якщо ви хочете мати лише одну панель, то встановіть властивість SimplePanel = true. У цьому випадку для виведення тексту в рядок стану використовуйте властивість SimpleText. Якщо вам необхідно мати кілька панелей, встановіть властивість SimplePanel = false. Тоді для виведення тексту в потрібну панель використовуйте властивість Panels[k], де номер k панелі (нумерація з 0) або підсвойство Items, наприклад:

```
StatusBar1.Panels[0].Text:=     'Який     текст';
StatusBar1.Panels.Items[0].Text:= 'Який текст';
```

Відкомпілюйте та запустіть програму. Випробуйте роботу кнопок.

У процесі введення інформації користувачі досить часто припускаються помилок. У більшості випадків ці помилки мають випадковий і ненавмисний характер, але можуть спричинити збій або навіть аварійне завершення програми. Тому програміст повинен передбачати відповідні заходи захисту своїх програм від таких помилок.

У прикладі ми реалізували дії для цілих чисел. При введенні чисел користувач може, наприклад, ввести неприпустимий символ або замість цілого ввести дійсне число. У цій програмі жодного контролю за введення немає. У разі програма, зазвичай, аварійно завершується. У
2.1.14 та 2.1.25 ми вже стосувалися цього питання.

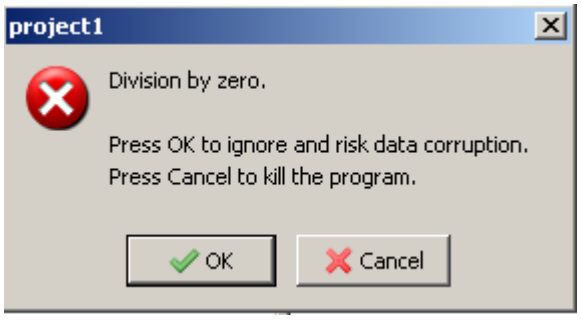
При виникненні будь-якої помилкової ситуацій (кажуть ще виняткову ситуацію) генерується так званий виняток. До виникнення винятків можуть призвести не лише дії користувача, а й низка інших обставин. Наприклад, звернення до неіснуючого файлу,

вання до пристрою, який у цей момент не готовий, вихід за межі виділеної динамічної області пам'яті, поділ на нуль, переповнення розрядної сітки та ін. Програміст зобов'язаний і для таких випадків передбачити якісь дії, щоб не допустити краху своєї програми.

Відвернемося на деякий час від вивчення компонентів та розглянемо деякі способи виявлення та організації відповідної реакції програми у разі виникнення будь-яких помилкових ситуацій.

6.3.7 Обробка винятків. Компонент TMaskEdit. Організація контролю введення даних

Для обробки стандартних винятків у Lazarus є спеціальні класи. Якщо у програмі не передбачено оброблення якогось виключення, воно обробляється глобальним обробником. Він забезпечує стандартну реакцію на виняток, що виник, - виводить попередження на екран з коротким описом причини, що викликало виняток, рис. 6.31.

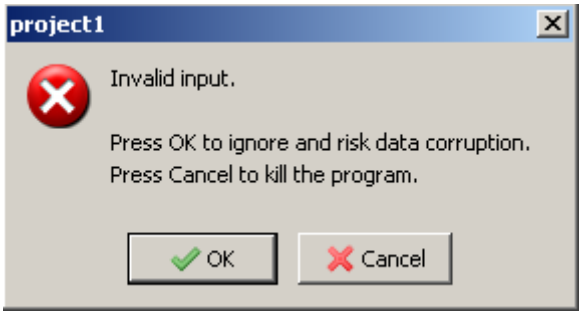


Мал. 6.31. Стандартна реакція на виключення

Тут зазначено, що відбулася спроба поділу на нуль та пропонуються варіанти дій – продовжити виконання програми або завершити її.

Найчастіше стандартної реакції виявляється недостатньою. Поясню прикладом. Припустимо, у програмі створюється тимчасовий файл, який після завершення програми має бути видалено. Нехай під час роботи

ти програми з файлом сталася помилка. Скажімо, було прочитано неприпустимий символ. Обробку такої помилки програмістом не було передбачено. Тоді настане стандартна реакція, рис. 6.32.



Мал. 6.32. Стандартна реакція на виключення

Користувач закриває програму, натиснувши кнопку Cancel. А тимчасовий файл, який мав бути знищений, залишиться на диску! Існує дуже багато програм, які залишають після своєї роботи всяке "сміття" на диску, якщо вони завершуються аварійно. І це відбувається через те, що програміст не передбачив відповідних дій щодо очищення сміття у разі виникнення виняткових ситуацій.

У таких випадках набагато краще, якщо програміст організує свою власну обробку виключення, в якій і виконає очищення сміття. Крім того, програміст знає логіку роботи своєї програми, знає – де саме стався виняток і, в більшості випадків, чому це сталося, тому він може видати користувачеві більш розгорнутий опис причини помилки, до того ж російською мовою, а також передбачити такі дії, які могли б користувачеві продовжити роботу (якщо це можливо).

Розглянемо деякі класи винятків.

- `EConvertError – помилка перетворення типів. Найчастіше відбувається при перетворенні рядка на число.`
- `EDivByZero – спроба розподілу цілого числа на нуль.`
- `EZeroDivide – спроба поділу речового числа на нуль.`

- ERangeError – вихід межі допустимого діапазону індексу чи порядкового типу.
- EIntOverflow – переповнення в операціях із цілими числами, тобто. спроба присвоєння змінної цілого типу значення більша за допустиме.
- EOverflow – переповнення в операціях із речовими числами.
- EUnderflow – втрата значних розрядів в операціях із речовими числами.
- EAccessViolation – спроба звернутися до недійсної адреси в пам'яті. Найчастіше виникає через неправильну роботу з покажчиками.

Для обробки винятків використовують дві конструкції. Перша конструкція має вигляд:

```
try
< Потенційно"небезпечні" оператори,  під час виконання яких можуть виникнути виняткові ситуації >
except
on клас виключення 1 do <оператор обробки виключення 1>; on клас виключення 2 do <оператор обробки виключення 2>;

……

on клас виключення n do <оператор обробки виключення

n>; else

<Оператори обробки інших винятків>

end;
```

Дана конструкція означає, що після ключового слова try і до ключового слова except випливають оператори, при виконанні яких можливе виняток.

Після except слідують оператори, які

утворюють секцію обробки винятків. Ознакою кінця секції є ключове слово end. Усередині секції програміст вказує класи виключень (говорять ще типи винятків) після слова on, а потім після ключового слова do оператор обробки виключення, причому оператор може бути складним. Після необов'язкового else слідують оператори обробки виключень, які не увійшли до переліку on. Якщо програмісту потрібно лише встановити сам факт виключення, незалежно від типу, він може просто записати обробник виключення після слова except.

Друга конструкція має вигляд:

```
try
< Потенційно "небезпечні" оператори, під час виконання яких можуть виникнути виняткові ситуації >
finally
< оператори, які мають бути виконані у будь-якому випадку, незалежно від того, відбулося виключення чи ні >
end;
```

У чому їхня відмінність? У конструкції try..except якщо при виконанні операторів секції try виник виняток, то керування передається в секцію except, де відбувається обробка виключення. Якщо ж винятку не сталося, то оператори блоку просто просто пропускаються. У конструкції try..finally оператори будуть виконані незалежно від того, сталося виняток чи ні.

Розглянемо приклад:

```
try
  num: = StrToInt (Stroka);
except
```

```
on EConvertError do
   ShowMessage('Помилка перетворення рядка на ціле
число'); end;
```

Тут повідомлення буде виведене лише в тому випадку, коли неможливо перетворити рядок символів на ціле число, тобто коли виникне виключення EConvertError.

Конструкції try..except та try..finally можуть бути вкладені одна в одну на необмежену глибину. Розглянемо реалізацію прикладу з файлом.

```
Procedure Test;
var
  F: TextFile;
  number: integer;
  s: string;
begin
  AssignFile(F, 'Data.txt');
  Rewrite(F);
  s:= '12#4';//       У файл свідомо записується
  Writeln(F, s); // помилковий рядок символів Reset (F);
  try // початок секції (блоку)
    try..except Readln(F, s);
    try // початок секції try..finally
      number:= StrToInt(s);
    finally
      CloseFile(F); // ці два оператори будуть виконані
      DeleteFile('Data.txt'); // у будь-якому випадку
```

```
      end;    // кінець секції try..finally except
      on EConvertError do
         ShowMessage('Помилка перетворення');
    end; // кінець секції try..except
  end;
```

Повернемося до прикладу, в якому здійснюється введення двох цілих чисел і виконуються чотири арифметичні дії (додавання, віднімання, множення і поділ націло), рис. 6.30. Модифікуємо програму, додавши до неї обробку винятків. Перепишіть обробники події OnKeyPress для LabeledEdit1 та подій OnClick кнопок SpeedButton1, SpeedButton2, SpeedButton3 та SpeedButton4 у вигляді:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
  if Key = #13 then
  begin
    try
      StrToInt(LabeledEdit1.Text);
    except
      on EConvertError do
      begin
        ShowMessage('Помилка перетворення!
                    Ймовірно, '+' Ви помилилися
                    при введенні числа');
        exit;
      end;
    end;
```

```
    LabeledEdit2.SetFocus;SpeedB
    utton1.Down:= false;
    SpeedButton2.Down:= false;
    SpeedButton3.Down:= false;
    SpeedButton4.Down:= false;
    exit;
  end;
end;

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  try
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                  + StrToInt(LabeledEdit2.Text));
  except
    on EConvertError do
    begin
      ShowMessage('Помилка перетворення! Ймовірно,
                  '+' Ви помилилися при введенні
                  другого числа');
      exit;
    end;
  end;
  StatusBar1.SimpleText:= 'Складання';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;

procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
```

```
  try
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                  - StrToInt(LabeledEdit2.Text));
  except
    on EConvertError do
    begin
      ShowMessage('Помилка перетворення! Ймовірно,
                  '+' Ви помилилися при введенні
                  другого числа');
      exit;
    end;
  end;
  StatusBar1.SimpleText:= 'Віднімання';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;

procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  try
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                  * StrToInt(LabeledEdit2.Text));
  except
    on EConvertError do
    begin
      ShowMessage('Помилка перетворення! Ймовірно,
                  '+' Ви помилилися при введенні
                  другого числа');
exit;
    end;
```

```
end;
```

```
  StatusBar1.SimpleText:= 'Умноження';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;

procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  try
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                div StrToInt(LabeledEdit2.Text));
  except
    on EConvertError do
    begin
      ShowMessage('Помилка перетворення! Ймовірно,
                  '+' Ви помилилися при введенні
                  другого числа');
      exit;
    end;
    on EDivByZero do
    begin
      ShowMessage('Помилка! Відбувся розподіл на нуль.
                  Ймовірно, '+' Ви помилилися при введенні
                  другого числа');
      exit;
    end;
  end;
  StatusBar1.SimpleText:= 'Поділ націло';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

Тепер при введенні неприпустимого символу для цілого числа, а також при введенні числа 0 як другий операнд програма перехоплює виключення і реагує відповідним чином.

Майте на увазі, якщо ви запускаєте програму з середовища Lazarus, то виключення перехоплюватиме налагоджувач. Тому краще запускати програму з командного рядка або в налаштуваннях оточення для налагоджувача додайте потрібні типи винятків до списку ігнорування.

Обробку винятків можна застосовувати і в консольних додатках. Згадаймо програму із 2.1.14. Організуємо контроль введення даних, використовуючи механізм винятків.

```
program int_operations_control;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  A, B, C: integer;
begin
  writeln(UTF8ToConsole('Введіть два числа'));
  readln(A, B);
  writeln('A=', A, 'B=', B); C:
  = A + B;
  writeln(UTF8ToConsole('Демонстрація додавання, C='),
  C); C: = A * B;
  writeln(UTF8ToConsole('Демонстрація множення, C='),
  C); try
    C: = A div B;
    writeln(UTF8ToConsole('Демонстрація поділу націло,
    C='),
                              C);
```

```
  except
  on EDivByZero do
    begin
      writeln(UTF8ToConsole('Помилка!! Поділ на нуль.'));
      writeln(UTF8ToConsole('Натисніть будь-яку
      клавішу')); readkey;
      exit;
    end;
  end;
  C: = A mod B;
  writeln(UTF8ToConsole('Залишок від поділу,
  C='), C); C: = A - B;
  writeln(UTF8ToConsole('Демонстрація віднімання,
  C='), C); writeln(UTF8ToConsole('Натисніть будь-яку
  клавішу')); readkey;
end.
```

Порівняйте цю програму із програмою з 2.1.25.

Контроль та обробку помилок із застосуванням механізму винятків зручніше використовувати, коли робота програми не залежить від дій користувача, тобто. всі необхідні дані від користувача вже отримані, програма перейшла безпосередньо до обробки даних – йдуть обчислення, відбувається читання та запис у файли, звернення до різних зовнішніх пристроїв тощо. На етапі введення даних, коли користувач у режимі діалогу вводить якісь дані з клавіатури, зручніше використовувати інший спосіб контролю. Оскільки обробка винятків відбувається після завершення введення користувачем, тобто цим відбувається лише констатація факту, що помилка користувачем вже здійснено. Доводиться організовувати повторне введення, причому з того місця, де користувач помилився. А це призведе до ускладнення.

ня коду. Набагато розумніше та ефективніше при введенні даних просто не давати користувачу зробити помилку. Наприклад, якщо користувач повинен вводити лише цілі числа, простіше контролювати введені користувачем символи і, якщо буде спроба ввести неприпустимий для цілих чисел символ, просто ігнорувати цей символ.

Для цього зручніше використовувати інший різновид компоненту TEdit – TMaskEdit зі сторінки Additional.

6.3.7.1. Компонент TMaskEdit

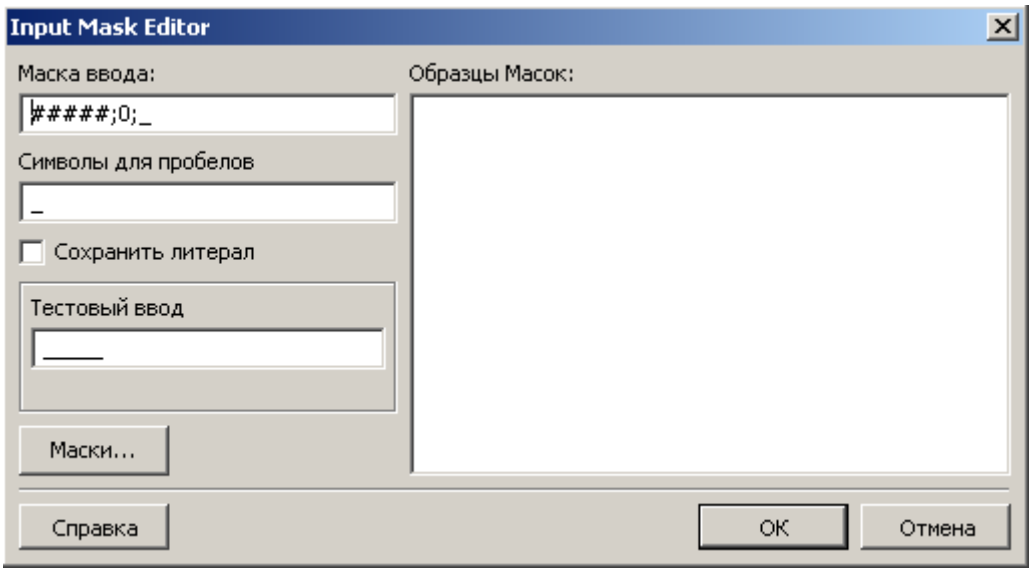
У цьому компоненті є властивість EditMask, за допомогою якого можна встановити маску введення. Маска це якийсь шаблон, який визначає які символи може вводити користувач у вікні введення. Неприпустимі символи ігноруються. Маска складається з трьох частин, між якими ставиться крапка з комою (;). У першій частині маски – шаблон записуються символи (табл. 6.2), які вказують які символи можна вводити в кожній позиції.

Таблиця 6.2

Символ	Шаблон введення
!	Означає, що в EditText символи, що відсутні, передаються пробілами. У разі відсутності символу пробіли розміщуються в наприкінці.
0	Означає, що у цій позиції має бути цифра.
9	Це означає, що в цій позиції може бути цифра або нічого.
#	Означає, що у цій позиції може бути цифра, знак «+», знак "-" або нічого.

Далі через точку з комою (;) записується 1 або 0 залежно від того, треба чи ні, щоб символи, що додаються маскою, включалися до властивості Text компонента. У третій частині маски вказується символ-заповнювач, який використовується для позначення позицій, у яких ще не здійснено введення. Встановити потрібну маску можна прямо у властивості EditMask, ввівши потрібні символи маски або в редакторі масок, відкрити який можна натиснувши

кнопку з трійкою, рис. 6.33.



Мал. 6.33. Вікно редактора масок

Прочитати результат введення можна або у властивості Text, яка, залежно від виду другої частини маски, включає або не включає символи маски, або у властивості EditText, що містить введений текст разом з символами маски.

Отже, давайте застосуємо для нашого прикладу компонент TMaskEdit. Видаліть з форми LabeledEdit1 та LabeledEdit2. Перенесіть на них два компоненти TMaskEdit. Також вставте у форму два компоненти TLabel. Загалом відновіть зовнішній вигляд форми як на малюнку 6.30.

Встановіть такі властивості MaskEdit1: •

```
AutoSelect = true
```

- `EditMask = # 9999; 0;`
- `TabOrder = 0`
- Властивість Text залиште порожнім.

Як символ заповнювача у вікні "Символи для пробілів" в редакторі масок введіть пробіл замість символу підкреслення.

Встановіть властивості MaskEdit2:

- AutoSelect = true•

EditMask = # 9999; 0;

- TabOrder = 1
- Властивість Text залиште порожнім.

Встановіть символ заповнювач такий самий, як і для MaskEdit1.

У LabeledEdit3 та BitBtn1 властивості TabStop присвойте значення false.

Обробники подій запишіть у вигляді:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  MaskEdit1.SetFocus;
end;
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  LabeledEdit3.Text:= IntToStr(StrToInt(MaskEdit1.Text)
                    + StrToInt(MaskEdit2.Text));
  StatusBar1.SimpleText:= 'Складання';
  MaskEdit1.SetFocus;
  MaskEdit1.SelectAll;
end;
procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  LabeledEdit3.Text:= IntToStr(StrToInt(MaskEdit1.Text)
                    - StrToInt(MaskEdit2.Text));
  StatusBar1.SimpleText:= 'Віднімання';
  MaskEdit1.SetFocus;
  MaskEdit1.SelectAll;
end;
```

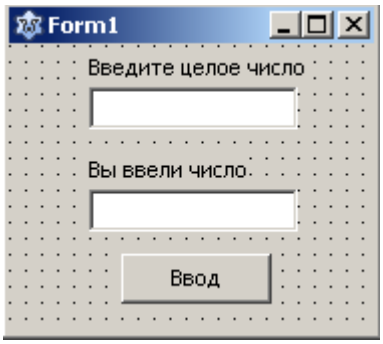
```
procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  LabeledEdit3.Text:= IntToStr(StrToInt(MaskEdit1.Text)
                    * StrToInt(MaskEdit2.Text));
  StatusBar1.SimpleText:= 'Умноження';
  MaskEdit1.SetFocus;
  MaskEdit1.SelectAll;
end;
procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  try
  LabeledEdit3.Text:= IntToStr(StrToInt(MaskEdit1.Text)
                  div StrToInt(MaskEdit2.Text));
  except
  on EDivByZero do
    ShowMessage('Помилка! Відбувся поділ на
      нуль.' + 'Вірогідно Ви помилилися при
      введенні другого числа');
  end;
  StatusBar1.SimpleText:= 'Поділ
  націло'; MaskEdit1.SetFocus;
  MaskEdit1.SelectAll;
end;
```

Як бачите, обробку винятків на можливі помилки перетворення типів ми прибрали, оскільки MaskEdit не дозволить користувачеві ввести неприпустимі символи. Але в обробнику операції поділу виняток, що виник при спробі поділу на нуль, ми, природно, залишили.

Перехід від MaskEdit1 до MaskEdit2 та назад здійснюйте клавішею Tab. Втім, можете додати обробник OnKeyPress для MaskE-

```
dit1:

procedure TForm1.MaskEdit1KeyPress(Sender: TObject;
                                   var Key: char);
begin
  if Key = #13 then
  begin
    MaskEdit2.SetFocus;SpeedButt
    on1.Down:= false;
    SpeedButton2.Down:= false;
    SpeedButton3.Down:= false;
    SpeedButton4.Down:= false;
    exit;
  end;
end;
```

Давайте спробуємо самі реалізувати контроль введення для компонента TLabeledEdit. Реалізуємо повний контроль введення цілих чисел з урахуванням знаку. Для цього створимо просте додаток. Помістіть на форму два компоненти TLabeledEdit та одну кнопку TButton, рис. 6.34.



Мал. 6.34. Форма застосування

Встановіть властивість `TabOrder=0` і `TabStop=true` для

LabeledEdit1 і TabOrder=1, TabStop=true для TButton1, а для LabeledEdit2 властивість TabStop=false. Це для того, щоб при натисканні кнопки Tab фокус автоматично переміщався з LabeledEdit1 на кнопку і назад. Крім того, встановіть властивість AutoSelect для LabeledEdit1 рівним true.

Вводитимемо числа у вікні LabeledEdit1, потім введене число просто виведемо в LabeledEdit2, але не "відразу"! Спочатку перетворимо його у внутрішнє уявлення цілого числа для того, щоб "виловити" помилки. Якщо будуть помилки під час введення, то природно, перетворення не відбудеться і виникне виняток EConvertError. Для налагодження та тестування нашої програми нам вистачить і стандартної реакції системи, тому обробку винятку ми робити не будемо. Додайте лише виняток EConvertError в список ігнор через меню**Оточення->Параметри>Відладчик->Виключення мови**щоб зручніше було запускати наш додаток з середовища Lazarus.

Отже, із чого почнемо? Завдання цілком зрозуміле. Необхідно забезпечити введення користувачем лише цифр від 0 до 9 та знака мінус (-) (знак плюс (+) перед числом зазвичай не ставиться). Для цього найпростіше скористатися певним у Паскаль типом – множиною і оператором in для визначення належності символу до допустимих.

У LabeledEdit1 для події OnKeyPress напишіть наступний обробник:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
{ дозволяємо лише цифри, знак мінус та кл.
  BackSpace } if not (Key in ['0' .. '9', '-
  ' , #8])
```

```
  then Key:= #0;
end;
```

В операторі if, якщо користувач натискає на клавіші відмінні від цифр, клавіші BackSpace і знаку (-) параметру Key присвоюється нульовий символ.

Для події OnClick кнопки напишіть наступний оброблювач:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  LabeledEdit1.SetFocus;
  n:= StrToInt(LabeledEdit1.Text);
  LabeledEdit2.Text:= IntToStr(n);
end;
```

Не забудьте оголосити змінну n цілого типу:

```
var
  Form1: TForm1;
  n: integer;
```

Запустіть програму. Спробуйте ввести символи, відмінні від цифр і мінус знак. Ви бачите, що неприпустимі символи просто не відображаються у вікні введення. І що? Завдання вирішено? На жаль, ні!

Введіть кілька цифр, потім знак мінус. Наприклад, "2010-". Якщо натиснути кнопку, отримайте повідомлення про помилку. Тому що знак числа ставиться перед першою цифрою. Або введіть спочатку два або більше знаки (-), потім цифри, наприклад, "--2010". Знову отримаєте помилку.

Ви скажете, ну не може бути, щоб користувач припускався таких помилок! Не настільки ж користувач дурний і тупий! І ви маєте рацію! Йдеться про випадкові помилки. Ми протягом цієї книги неодноразово говорили про це. Програма має бути захищена від будь-яких випадкових та ненавмисних

помилок користувача!

Значить після введення цифр знак (-) теж повинен бути заборонений. Крім того, необхідно дозволити введення знака числа лише один раз, причому спочатку числа! Для цього введемо логічну змінну, назвемо її sign. Спочатку її значення дорівнюватиме false. Як тільки користувач натиснув хоча б один допустимий символ, включаючи знак числа, змінної sign присвоюємо значення true.

Обробник для LabeledEdit1 переписуємо в наступному вигляді, заодно дозволимо користувачеві закінчувати введення натисканням клавіші Enter:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
  if Key = #13 then
  begin
    sign:= false;
    n:= StrToInt(LabeledEdit1.Text);
    LabeledEdit2.Text:= IntToStr(n);
  end;
{ дозволяємо лише цифри, знак мінус та кл.
  BackSpace } if not (Key in ['0' .. '9',
  '-' , #8])
  begin
    Key:= #0;
    exit;
  end;
  if (Key = '-') and (sign)
  then Key:= #0;
  sign:= true;
```

```
end;
```

Оператор

```
if (Key = '-') and (sign)
then Key:= #0;
```

не дозволить користувачеві ввести знак числа після значних цифр і, крім того, двічі ввести знак (-).

Тепер, здається, все. Але ні, ще не все! Уявіть ситуацію – користувач хотів запровадити негативне число, але забув запровадити знак мінус, а цифри вже запровадив. Спробуйте ввести кілька цифр, потім повернути курсор на початок рядка та спробуйте ввести (-). У вас не вийде! Навіть якщо ви видалите введені цифри кнопкою Delete або BackSpace. Тому що sign має значення true. Ситуація не така проста, як здається. Щоб ввести від'ємне число, потрібно спочатку натиснути кнопку "Введення" або на клавішу Enter, тобто. Користувач буде змушений ввести число, яке не хотів вводити!

Доведеться контролювати положення курсору. Компонент має властивість SelStart, яка вказує на поточне положення курсору. Використовуючи цю властивість і функцію Pos, перепишемо обробник так:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
  if Key = #13 then
  begin
    sign:= false;
    n:= StrToInt(LabeledEdit1.Text);
    LabeledEdit2.Text:= IntToStr(n);
    exit;
```

```
  end;
{ дозволяємо лише цифри, знак мінус та кл.
  BackSpace } if not (Key in ['0' .. '9',
  '-' , #8])
  begin
    Key:= #0;
    exit;
  end;
  if (Key = '-') and (sign)
  then
  begin
    if (LabeledEdit1.SelStart <> 0)
    then
    begin
      Key:= #0;
      exit;
    end
    else
      if (Pos('-', LabeledEdit1.Text) <> 0)
      then Key:= #0;
  end;
  sign:= true;
end;
```

Обробник працює наступним чином – якщо поточний символ (-) і цифри введені, то перевіряємо позицію курсора. Якщо курсор встановлений на початок і мінусу немає, то дозволяємо вводити знак.

Уф! Нарешті має все заробити як слід. Але ж є нюанс! Спробуйте ввести лише один знак (-) і більше нічого. Знову отримаєте

помилку. Але це вже занадто, ви вигукнете! Кому спаде на думку вводити один тільки мінус. Ну, а раптом, випадково, користувач натиснув на Enter? І ще. А якщо, користувач нічого не ввівши, знову ж таки випадково, натисне кнопку або Enter, тобто. буде введено порожній рядок?

У вас немає відчуття, що ми пишемо програму для американців? (Як говорив М. Задорнов - "Ну, тупі ...!).

Зробимо так, якщо довжина рядка дорівнює одиниці і це знак (-), забороняємо введення.

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
  if Key = #13 then
  begin
    sign:= false;
    if Length(LabeledEdit1.Text) = 0 // якщо порожній рядок
    then exit;
    if (Length(LabeledEdit1.Text) = 1) and
       (LabeledEdit1.Text = '-')
    then exit;
    n:= StrToInt(LabeledEdit1.Text);
    LabeledEdit2.Text:= IntToStr(n);
    exit;
  end;
{ дозволяємо лише цифри, знак мінус та кл.
  BackSpace } if not (Key in ['0' .. '9',
  '-' , #8])
  begin
    Key:= #0;
```

```
    exit;
  end;
  if (Key = '-') and (sign)
  then
  begin
    if (LabeledEdit1.SelStart <> 0)
    then
    begin
      Key:= #0;
      exit;
    end
    else
      if (Pos('-', LabeledEdit1.Text) <> 0)
      then Key:= #0;
  end;
  sign:= true;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  LabeledEdit1.SetFocus;sign
  := false;
  if Length(LabeledEdit1.Text) = 0 // якщо порожній рядок
  then exit;
  if (Length(LabeledEdit1.Text) = 1)and
     (LabeledEdit1.Text = '-')
  then exit
  else
  n:= StrToInt(LabeledEdit1.Text);
```

```
  LabeledEdit2.Text:= IntToStr(n);
end;
```

Між іншим, компонент TMaskEdit при встановленій масці на введення тільки чисел теж не "ловить" одиночний мінус і "мовчки, ковтає" порожній рядок!

Чому такі ситуації слід контролювати? Уявіть собі, що користувач має вводити велику кількість числових даних. Через деякий час після початку введення користувач починає втомлюватися і, чисто ма- шинально, вводить порожній рядок або одиночний мінус. Ваша програма аварійно завершується через помилку перетворення (EConvertError)! Вся пророблена користувачем робота пішла "коту під хвіст". Як же він вас проклинатиме! Саме вас – розробника програми!

Цей приклад ще раз демонструє, як треба послідовно покращувати та "доводити" алгоритм і програму до досягнення необхідної функціональності. Нехай сам алгоритм, можливо, не є оптимальним. Не в цьому суть! Це конкретний (хоча й простий) приклад того, як треба розробляти реальні програми. Вже неодноразово зазначав, що з першого разу неможливо написати програму, що повністю правильно працює. Якщо так можна виразитися, цей приклад показує метод "послідовних наближень" розробки програм.

А тепер спробуємо реалізувати контроль за введенням дійсних чисел. Зрозуміло, що треба додати перевірку на символи точки (.) та коми (,). Використання точки або коми для відокремлення цілої частини числа від дробової залежить від налаштувань операційної системи Windows. Хоча Lazarus спокійно перетворює рядок на число і з точкою, і з комою. А ось Delphi "лається", якщо роздільник не збігається з налаштуваннями Windows. Тож будемо враховувати обидва роздільники. Для визначення який роздільник використовується системою, служить глобальна змінна DecimalSeparator, значенням якої є

ється символ роздільника. При цьому вчинимо так, якщо користувач ввів "не той" роздільник, то просто замінимо його на потрібний. Отже, наша програма "володітиме" вже деяким інтелектом! При реалізації застосуємо для різноманітності дещо інший спосіб, ніж у попередній програмі. Використовуватимемо оператор вибору case .. of. Грунтуючись на досвіді розробки попередньої програми, враховуватимемо не лише одиничний мінус, а й одиночний роздільник, і комбінацію "-" та роздільник. Знову ж таки для різноманітності використовуємо компонент TEdit. Код програми:

```
unit Unit1;
{$mode objfpc}{$H+}

interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls;
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
  private
    {private declarations}
```

```
  public
    { public declarations }
  end;
var
  Form1: TForm1;

implementation
{TForm1}
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SetFocus;
  if (Edit1.Text <> '') and // якщо порожній рядок
     (Edit1.Text <> '-') and // якщо одиночний мінус
     (Edit1.Text <> DecimalSeparator) and {якщо одиночний
роздільник}
     (Edit1.Text <> '-' + DecimalSeparator) {якщо "-" і
роздільник}
  then Edit2.Text:= FloatToStr(StrToFloat(Edit1.Text));
  Edit1.Clear;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  case Key of
    '0'..'9', # 8:; // якщо цифри та кл. BackSpace,
    введення дозволено
    #13: // якщо натиснута клавіша
      <Enter> begin
        if (Edit1.Text <> '') and
```

```
        (Edit1.Text <> '-') and
        (Edit1.Text <> DecimalSeparator) and
        (Edit1.Text <> '-' + DecimalSeparator)
    then
      Edit2.Text:=FloatToStr(StrToFloat(Edit1.Text));
    Edit1.SetFocus;
    Edit1.Clear;
  end;

'.',',': // якщо роздільники
begin
  if Key <> DecimalSeparator then
  Key := DecimalSeparator; // Заміна
  роздільника if Pos(DecimalSeparator,
  Edit1.Text) <> 0 then Key:= #0;
end;

'-' : // якщо знак мінус
begin
if (Edit1.SelStart <> 0) // якщо одиночний мінус
then
begin
  Key:= #0;
  exit;
end
else
  if (Pos('-', Edit1.Text) <> 0) // два мінуси
  then Key:= #0;
end;
```

```
  else
  Key := #0;
  end;
end;

initialization
  {$I unit1.lrs}

end.
```

І, нарешті, найпростіший спосіб контролю за введенням числових даних це використання функції val(). Функція була розглянута нами у 3.3.1.4.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls;
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
```

```
                              var Key: char);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
  code: integer;
  value: real;
implementation
{TForm1}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SetFocus;val(Edit1.Text
  , value, code); if code <> 0
  then exit;
  Edit2.Text:= FloatToStr(value);
end;
procedure TForm1.Edit1KeyPress(Sender: TObject;
                                 var Key: char);
begin
if Key = #13 then
  begin
    Edit1.SetFocus;val(Edit1.Text
    , value, code); if code <> 0
    then exit;
    Edit2.Text:= FloatToStr(value);
```

```
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

6.3.8 Спеціальні компоненти для введення чисел

Якщо необхідно вводити послідовність чисел із заздалегідь визначеним кроком зміни значень, то зручніше використовувати спеціальні компоненти TSpinEdit і TFloatSpinEdit, розташовані на вкладці Misc. Основні властивості цих компонентів:

- Increment – крок збільшення числа. Зрозуміло, для TSpinEdit крок може бути лише цілим, а для TFloatSpinEdit може бути й дробовим.
- MaxValue – максимально можливе значення числа.
- MinValue – мінімально можливе значення числа.
- ReadOnly – змінити значення користувач неспроможна, правда програмно змінити значення можна.
- Value – вказує на поточне значення. Під час проектування можна задати початкове значення числа, з якого розпочинається введення.
- DecimalPlaces – тільки для TFloatSpinEdit, вказує кількість розрядів після коми.

Вводити числа можна не лише за допомогою кнопок прокручування, але й з клавіатури. При цьому забезпечується контроль за введенням неприпустимих символів.

У компоненті TSpinEdit не можна ввести число поза встановленим діапазоном як за допомогою кнопок прокручування, так і з клавіатури. У

компоненті TFloatSpinEdit якщо введення здійснюється за допомогою кнопок, то перехід за межі вказаного під час проектування діапазону блокується, тобто. наприклад,

натискаючи на кнопку збільшення не можна ввести число більше MaxValue або натискаючи кнопку зменшення, не можна ввести число менше MinValue. Але можна ввести числа поза цим діапазоном з клавіатури. Однак, якщо після цього натиснути на будь-яку з кнопок прокручування, то у вікні введення з'явиться число, перше з можливих у встановленому діапазоні! Наприклад, ви встановили MaxValue = 10.0, з клавіатури ввели число 24.5. Після цього натиснули кнопку збільшення, у вікні компонента з'явиться число 10.0. Якщо натиснути на кнопку зменшення, то з'явиться число 9.5.

Згадаю ще один компонент TUpDown із вкладки Common Controls, який дозволяє вводити цілі числа. Основні властивості у нього практично такі самі, як і у TSpinEdit, лише по-іншому названі. Наприклад, властивість MaxValue у TUpDown називається Max, MinValue у TUp-Down називається Min, властивість Value називається Position. Якщо його використовувати "поодинці", то поточне значення (Position) буде не видно користувачеві. Тому його найчастіше використовують разом із TEdit. Для цього є властивість Associate. Використовуючи список, що розкривається, можна вказати потрібний компонент. У цьому випадку TUpDown "притискається" до вибраного компонента і обидва компоненти перетворюються на один, але візуально! У програмі звертатися до них треба окремо за своїми іменами. Властивістю AlignButton можна керувати розташуванням TUpDown навколо компонента (ліворуч, праворуч, зверху, знизу). Також є властивість Orientation, що дозволяє керувати положенням кнопок TUpDown (вертикально або горизонтально). І, нарешті, властивість Wrap визначає, що відбуватиметься, якщо користувач спробує ввести значення більше максимального або менше мінімального. При Wrap=false введення числа за допомогою кнопок поза вказаним діапазоном (Min..Max) блокується, а при Wrap=true відбувається автоматичний перехід з Min на Max при натисканні кнопки на зменшення та з Max на Min при натисканні кнопки на збільшення. При введенні з клавіатури слід-

дує пам'ятати, що цей компонент абсолютно не захищений від введення неприпустимих символів.

В цілому можна сказати, що розглянуті компоненти зручні лише у випадку, якщо числа, що вводяться, мають постійний крок збільшення і відомий діапазон допустимих значень. При введенні випадкових чисел, особливо якщо вони досить далеко "відстоять" один від одного за значенням, їх застосування може завдати користувачеві лише незручності.

Отже, ми з вами розглянули способи організації введення та контролю чисел даних. І хоча ми можемо бути тепер впевнені, що користувач ввів числа, а не щось інше, на жаль, при виконанні програми все ще можуть з'явитися помилки. Для пошуку та локалізації помилок виконують спеціальні дії, які називаються тестуванням та налагодженням програми.

6.3.9 Тестування та налагодження програми

Які бувають види помилок?

Помилки можна поділити на три групи. До першої відносяться так звані синтаксичні помилки. Це помилки, допущені безпосередньо в момент кодування, тобто. запису (набору в редакторі вихідного коду) тексту програми. Такі помилки найлегше знаходити та виправляти, оскільки компілятор сам вкаже на такі помилки.

До другої групи помилок відносяться так звані логічні помилки або їх називають помилками часу виконання. Іноземні програмісти їх називають bugs – жучки. Логічні помилки виявляються під час запуску програми. У більшості випадків логічні помилки призводять до отримання невірних результатів. Дуже часто програма завершується аварійно. Програміст отримує лише повідомлення про помилку. Якщо логічна помилка тривіальна, то за повідомленням можна визначити, де саме криється помилка. Однак бувають помилки, які перебувають з великими труднощами.

Третя група помилок – це алгоритмічні помилки. Це такі помилки, при яких зовні правильно і без помилок працююча програма видає докорінно невірні результати або вирішує зовсім не завдання.

Насправді досить важко відразу визначити, якої групи належить та чи інша помилка – до логічної чи алгоритмической. Якщо програма видає неправильні результати (але працює!), слід передусім перевірити сам алгоритм розв'язання задачі. І якщо виявляється помилка в алгоритмі, то ця алгоритмічна помилка. Якщо алгоритм правильний, то помилку слід визнати логічною.

Процес виявлення алгоритмічних помилок називається тестуванням. При тестуванні задається деякий ряд вхідних даних, званих модельними даними, для яких заздалегідь відомий результат (згадайте завдання обчислення значень синуса, розд. 2.1.26) або для яких можна "вручну" обчислити потрібний результат. Наприклад, якщо це програма нарахування заробітної плати, то дуже часто, на етапі впровадження, розрахунок заробітної плати здійснюється паралельно за допомогою програми та бухгалтерії. Або, якщо програма сортує якісь дані, можна просто подивитися на отримані результати та визначити, правильно програма працює чи ні.

Процес пошуку, виявлення та виправлення логічних помилок називається налагодженням. В основному процес налагодження зводиться до визначення в коді програми місця, де "сидить" помилка (жучок!). Для налагодження програми Lazarus є кілька корисних інструментів.

✓ **Точки зупинки (Breakpoints).**Виконання програми припиняється після досягнення деякого місця (рядка коду). Точку зупинки задає сам програміст. Для цього треба в редакторі вихідного коду просто клацнути мишею по полю, де показані номери рядків коду навпроти потрібного рядка. Точка зупинки буде виділена червоним кольором і позначена галочкою, рис. 6.35. При запуску програми відладчик виконає всі оператори до точки зупинки та зупинить виконання програми. Далі ви можете продовжити виконання

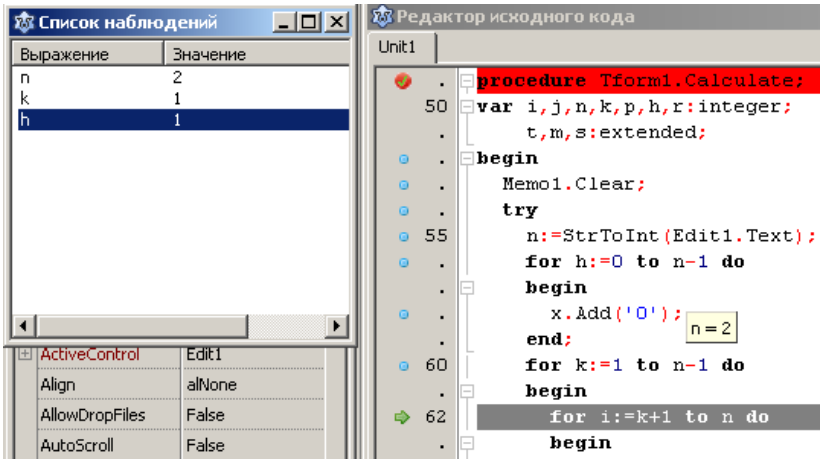
програми покроково.

✓ **Покрокове виконання із входом.**При натисканні клавіші F7 будуть виконані всі оператори поточного рядка (тому рекомендується розташовувати в рядку строго по одному оператору, щоб мати можливість контролювати виконання кожного оператора окремо). Кожне наступне натискання клавіші F7 викликає виконання оператора у наступному рядку. Оператор, що виконується в даний момент, буде виділений сірим кольором, а її номер позначений зеленою стрілкою, рис. 6.35. Якщо у коді зустрічається виклик процедури або функції, то будуть виконані всі оператори тіла цієї процедури (функції) також у покроковому режимі.

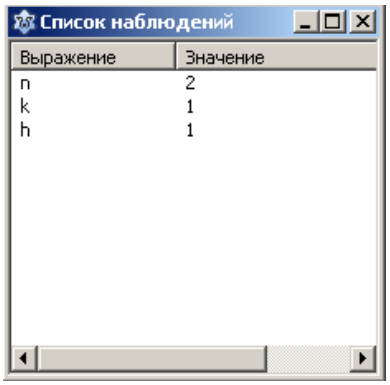
✓ **Покрокове виконання в обхід.**При кожному натисканні клавіші F8 будуть виконані всі оператори поточного рядка. Якщо у коді зустрічається виклик процедури чи функції, то будуть виконані всі оператори тіла процедури (функції) без входу, тобто. "відразу" і покажчик буде встановлений на наступному після виклику підпрограми рядку.

✓ **Вікно спостережень.**Мабуть, найголовніший і найкорисніший інструмент налагодження. У цьому вікні можна спостерігати поточні значення змінних у процесі виконання програми. Цей засіб допоможе вам простежити за життєвим циклом змінних, що вас цікавлять, і побачити, як змінюються їх значення після виконання того чи іншого оператора. Відкрити це вікно можна командою меню Вид-> Вікна налагодження -> Вікно спостережень. Ще простіше відкрити комбінацією клавіш <Ctrl+Alt+W>. Вигляд вікна спостережень показано на малюнку 6.36. Щоб додати нову змінну, клацніть правою клавішею у вікні спостережень, у діалоговому вікні введіть ім'я змінної або вираз, рис. 6.37. Можна призначити постійне розташування вікна спостережень у налаштуваннях оточення, наприклад, оскільки показано на рис. 6.35. Вікно спостережень має таку саму ширину, як і інспектор об'єктів і частково його перекриває. Це логічно, тому що в момент налагодження інспектор об'єктів нам не потрібен і вікно спостережень не заважає працювати в редакторі

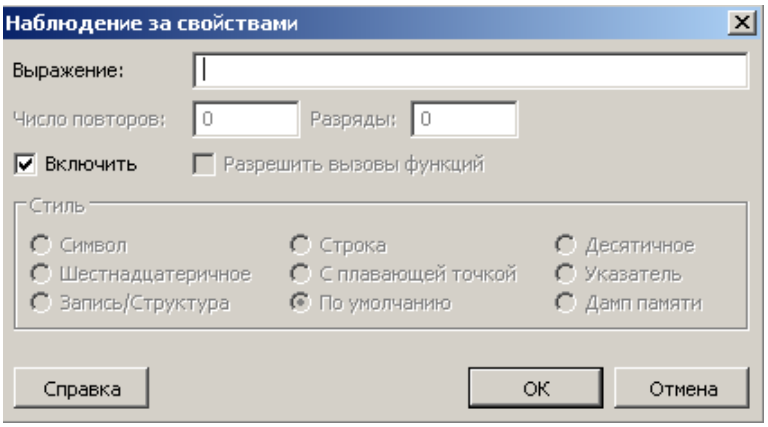
вихідного коду. Побачити значення змінної можна також, просто навівши мишу на цю змінну в редакторі вихідного коду. На малюнку 6.37. показано момент, коли миша була наведена на змінну n у 56 рядку редактора коду.



Мал. 6.35. Покрокове виконання програми



Мал. 6.36. Вікно списку спостережень



Мал. 6.37. Вікно додавання нової змінної до списку спостереження

Вважають, що налагодження передує тестуванню. Тобто про-

граміст, розробляючи програму, спочатку налагоджує її (виявляє та "вбиває" всіх "жучків"), потім приступає до тестування. Але, взагалі кажучи, тестування та налагодження взаємопов'язані процеси. Найчастіше при тестуванні виявляються помилки, які виправляються... налагодженням, і навпаки, під час налагодження виявляються багато алгоритмічних помилок. Тому це словосполучення практично завжди вживають разом – тестування та налагодження.

В принципі, тестування починається тоді, коли ваша програма стала цілком працездатною, тобто перестала "вилітати", "зависати", "зациклюватися" і так далі і почала видавати результати. Чи правильні ці результати виявляються тестуванням. Але, крім правильності результатів, під час тестування дуже важливо перевіряти "неприпустимі", "неможливі" і навіть "дурні" дії користувача. Якщо він не повинен вводити число 0, ви обов'язково повинні перевірити, як поведеться ваша програма при введенні саме цього самого нуля і т.д. У наведених нами прикладах ми, по суті, цим і займалися!

6.3.10 Компоненти відображення та вибору даних

Ця група компонентів дозволяє наочно відображати інформацію, а також легко здійснювати вибір потрібних даних. Інформація у цих компонентах міститься у вигляді списку (набору) рядків. Для роботи зі списками рядків розроблено спеціальний клас TStrings. Цей клас є абстрактним та інкапсулює методи для роботи з наборами рядків. Від цього класу наслідуються спеціалізовані класи в компонентах відображення та вибору даних, які забезпечують доступ та обробку набору рядків відповідно до функціональності компонента. Оскільки методи додавання та видалення рядків у класі TStrings не реалізовані та оголошені як абстрактні,

всі класи спадкоємці у компонентах відображення та вибору перекривають ці методи. Для введення, редагування та виведення багаторядкових даних зручніше використовувати компонент TMemo.

6.3.10.1. Компонент TMemo

Вікно компонента поводиться як традиційний текстовий редактор типу " Блокнот " , тобто. доступні всі стандартні функції редагування (виділення, копіювання, вставка, видалення тощо).

Інформація в TMemo міститься у вигляді сукупності (масиву) рядків типу TStrings. Кожен елемент масиву містить один рядок. Доступ до окремого рядка здійснюється за допомогою властивості Lines за її номером (індексом). Індекс вказується, як і належить для масивів, у квадратних дужках. Нумерація рядків починається із нуля. Загальна кількість рядків міститься у властивості Lines.Count.

Якщо рядок не вміщується повністю у вікні, то можна встановити властивість WordWrap= true і тоді частина рядка, що не вмістилася, буде автоматично перенесена на наступний рядок.

Також можна встановити смуги прокручування властивістю ScrollBars. Можливі значення властивості:

- ssNone – встановлено за замовчуванням, смуги прокручування відсутні.
- ssVertical – встановити вертикальну смугу прокручування.
- ssHorizontal – встановити горизонтальну смугу прокручування.
- ssBoth – встановити і вертикальну та горизонтальну смуги.
- ssAutoVertical – вертикальна смуга у вікні компонента помітна, але не доступна, доки вікно не заповниться по вертикалі.
- ssHorizontal – горизонтальна смуга у вікні компонента

помітна, але не доступна, доки вікно не заповниться по горизонталі.

- ssAutoBoth – поєднує два попередні значення.

Заборонити користувачеві редагування можна, встановивши властивість ReadOnly=true.

Додавання нового рядка при введенні даних з клавіатури здійснюється натисканням клавіші Enter, при цьому властивість WantReturns має бути рівним true. Якщо WantReturns=false, то для переходу на новий рядок необхідно натиснути Ctrl+Enter.

Властивість SelStart вказує на початок виділеного тексту, а SelLength – довжину виділеного тексту (кількість символів).

У властивості Text весь набір рядків представляється як одного рядка з роздільниками повернення каретки і перенесення рядка (#13#10) між рядками.

Подивимося, як програмно заповнювати поле введення компонента. Щоб додати рядок у TMemo, необхідно скористатися методами:

- function Add(const S: string): integer; – додає рядок S в кінець набору рядків TMemo та повертає її індекс.
- procedure Append(const S: string); – просто додає рядок S до кінця набору рядків.

Щоб додати цілий набір рядків, наприклад, з іншого компонента TMemo, можна застосувати методи:

- procedure AddStrings(TheStrings: Tstrings); – де The-Strings набір рядків типу TStrings, додає цей набір рядків до існуючого.
- procedure Assign(Source: TPersistent); – повністю очищає вміст TMemo та завантажує новий набір рядків із Source.

Для вставки рядка довільне місце списку рядків існує метод

procedure Insert(Index: integer; const S: string); – де Index номер (індекс) рядка куди вставляється рядок S. При цьому старий рядок не зникає, а зсувається вниз разом з усіма рядками нижче (їх індекси автоматично збільшуються на одиницю).

Замінити вміст якогось рядка можна простим оператором присвоєння, наприклад, щоб замінити вміст рядка з індексом до достатньо записати оператор:

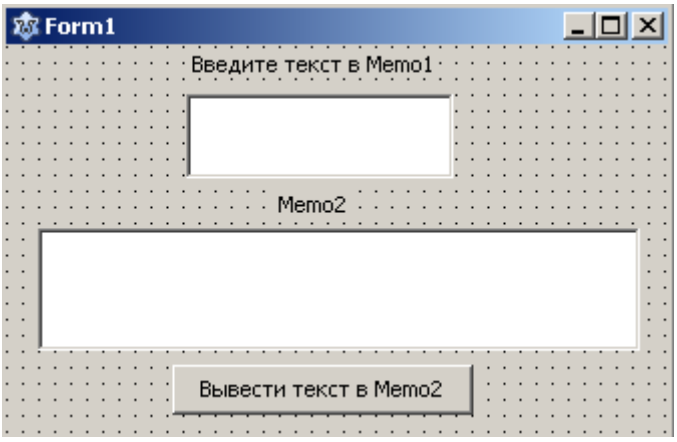
```
Memo1.Lines[k]:='Вміст рядка, що замінюється';
```

Додавати рядки оператором присвоювання теж можна, але за однієї умови, не можна застосовувати цей спосіб під час створення форми, тобто. в обробці OnCreate форми.

І, нарешті, щоб видалити рядок застосовується метод

```
Delete (Index: integer);
```

Давайте застосуємо отримані знання простому прикладі. Створіть новий проект. Покладіть на форму два компоненти TMemo, два написи та кнопку приблизно так, як показано на малюнку 6.38.



Мал. 6.38. Форма з компонентами TMemo

У обробник OnCreate форми введіть код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Memo1.Lines.Append('Це довгий рядок,'+ // метод Append
                     'не міститься у вікні Memo1');
  Memo1.Lines.Add('А це короткий рядок'); // метод Add
```

```
end;
```

У обробник OnClick кнопки введіть код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: integer;
begin
  Memo1.Lines [2]: = 'Акінфєєв'; // додавання
  рядківMemo1.Lines[3]:= 'Аршавін';//      оператором
  присвоєння Memo1.Lines.Delete(2);       //видалення рядка
  Memo1.Lines.Insert(2, 'Гус Іванович'); // вставка рядка Memo2.Clear;
                          //очищення набору рядків Memo2
  for i:= 0 to Memo1.Lines.Count - 1 do// додавання рядків
  Memo2.Lines.Append(Memo1.Lines[i]);// у циклі
  //Memo2.Lines.Add(Memo1.Lines[i]);// можна застосувати
  Add
  {додавання всього вмісту Memo1}
  Memo2.Lines.AddStrings(Memo1.Lines);// 1-й спосіб
  Memo2.Lines.Assign(Memo1.Lines);// 2-й, з очищенням
end;
```

Поекспериментуйте з введенням тексту за різних значень властивостей WordWrap, WantReturns, ReadOnly та ScrollBars.

Для встановлення параметрів шрифту є властивість Font. У ньому є такі підсвойства, як Name – назва шрифту, Size – розмір, Color – колір, Style – стиль шрифту та інші.

Властивість Aligment – служить для вирівнювання тексту у вікні TMemo. Воно має значення:

taLeftJustify - вирівнювання по лівому краю.

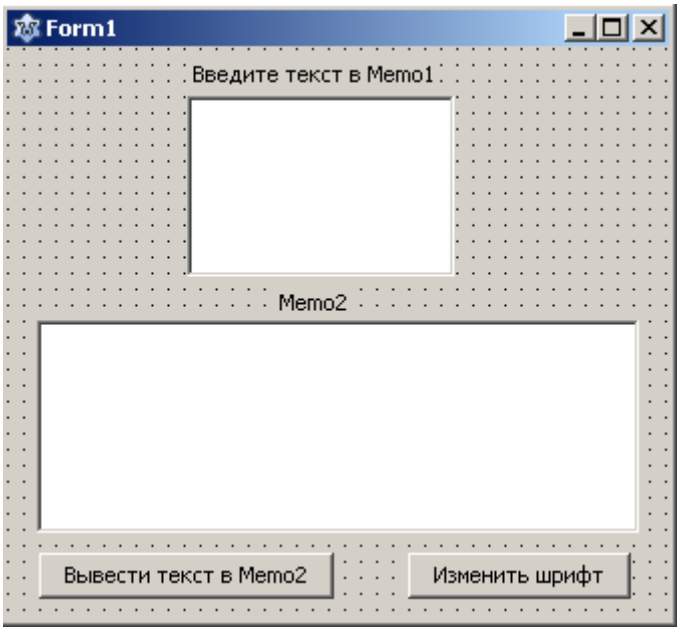
taCenter – вирівнювання центром.

taRightJustify – вирівнювання праворуч.

Властивість Font також доступна у програмі. Для попереднього прикладу додайте форму ще одну кнопку, як показано на рис. 6.39.

Напишіть для цієї кнопки наступний обробник:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Memo2.Font.Size:= 12;
  Memo2.Font.Color:= clBlue;
end;
```



Мал. 6.39. Форма застосування

Після запуску програми та натискання на кнопку "Змінити шрифт" текст в Memo2 змінить розмір і колір.

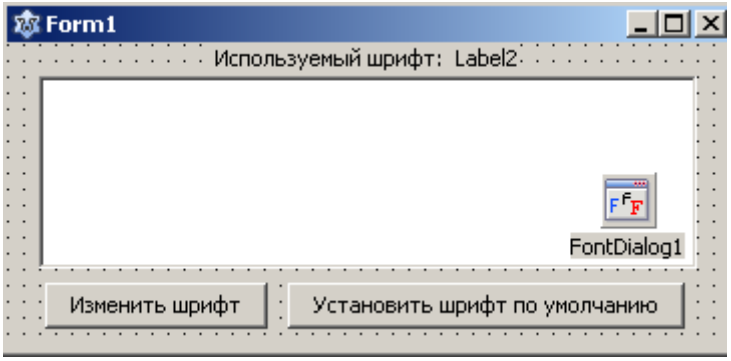
Виникає питання, а чи не можна надати користувачеві право змінювати атрибути шрифту? Для цього існує спеціальний компонент вибору властивостей шрифту TFontDialog. Він розташований на сторінці Dialogs панелі компонентів. Компонент є не візуальним, тому що під час

виконання користувач побачить не сам компонент, а стандартний діалог, що викликається при зверненні до цього компонента. Тому його можна розмістити у будь-якому місці форми. Показ діалогу ініціюється спеціальним методом компонента Execute. Це функція, яка повертає true, якщо в результаті діалогу користувач вибрав якісь параметри шрифту. Вибрані параметри запам'ятовуються у властивостях компонента TFont-Dialog. Якщо користувач нічого не вибрав і натиснув на кнопку Скасувати або клавішу Esc, то функція повертає false.

Таким чином, для виклику діалогу необхідно записати:

```
if FontDialog1.Execute then
Memo1.Font.Assign(FontDialog1.Font);
```

Створіть нову програму. Перенесіть на форму компонент TMemo, TFontDialog, дві кнопки та два написи, як показано на рис. 6.40.



Мал. 6.40. Форма програми з додатковим компонентом TFontDialog

У обробник кнопки "Змінити шрифт" введіть код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if FontDialog1.Execute then
```

```
  Memo1.Font.Assign(FontDialog1.Font);Label2.Capt
  ion:= Memo1.Font.Name + ', '
                + IntToStr(Memo1.Font.Size);
  Memo1.SetFocus;
end;
```

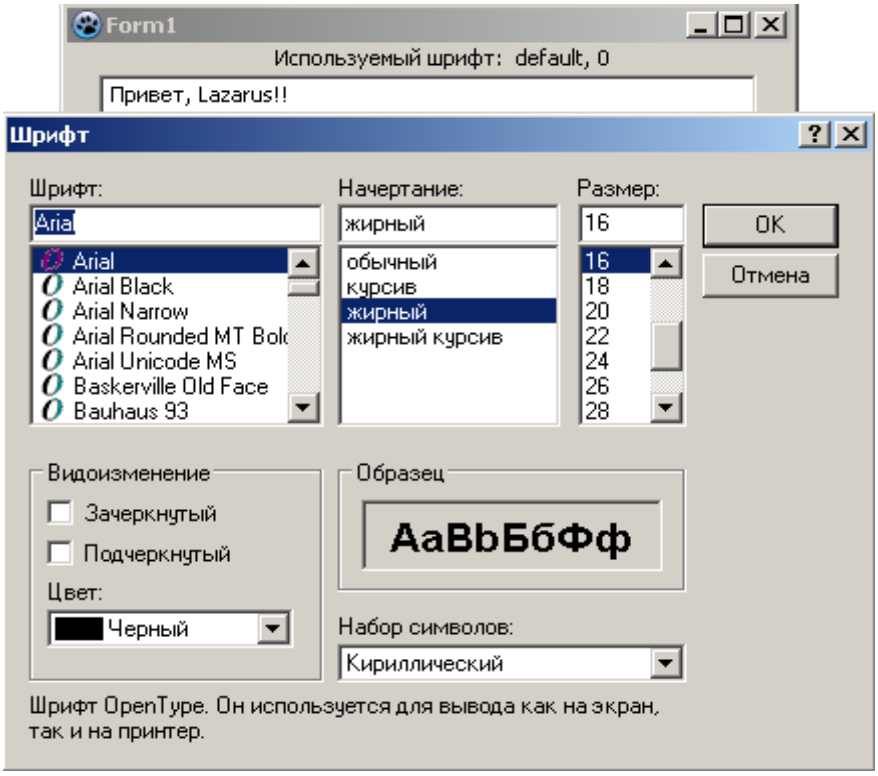
У обробник кнопки "Встановити шрифт за замовчуванням" введіть код:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Memo1.Font.SetDefault;
  Label2.Caption:= Memo1.Font.Name + ', '
                + IntToStr(Memo1.Font.Size);
end;
```

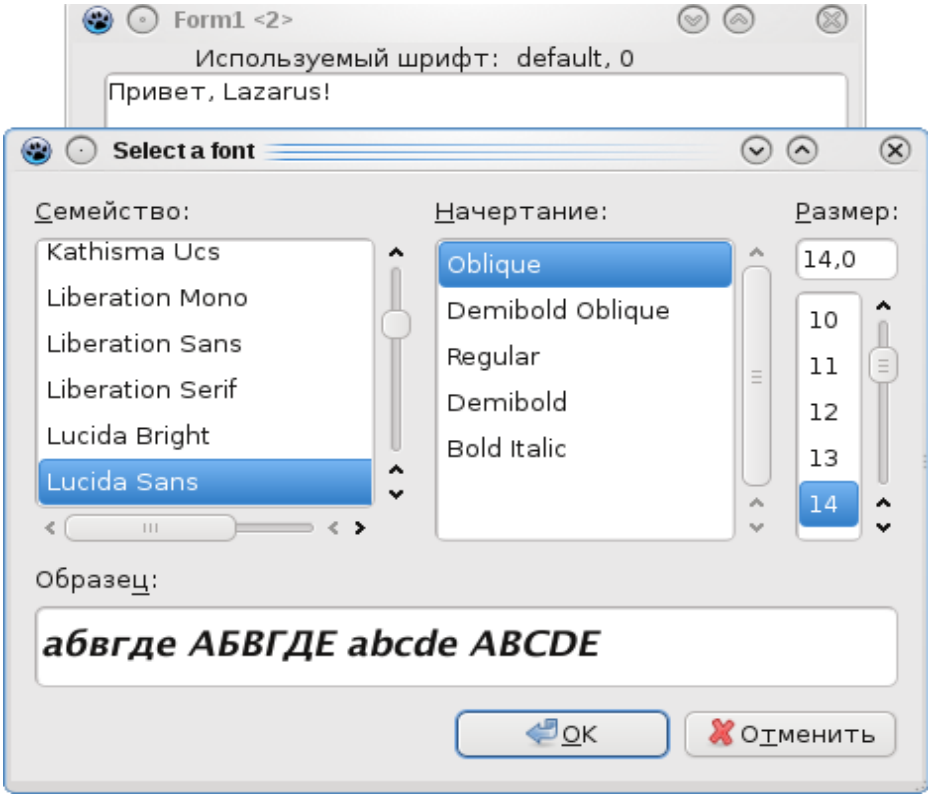
І, нарешті, в обробник OnCreate форми введіть код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Label2.Caption:= Memo1.Font.Name + ', '
                + IntToStr(Memo1.Font.Size);
end;
```

Тепер користувач може сам на власний розсуд встановлювати параметри шрифту, рис. 6.41, 6.42.



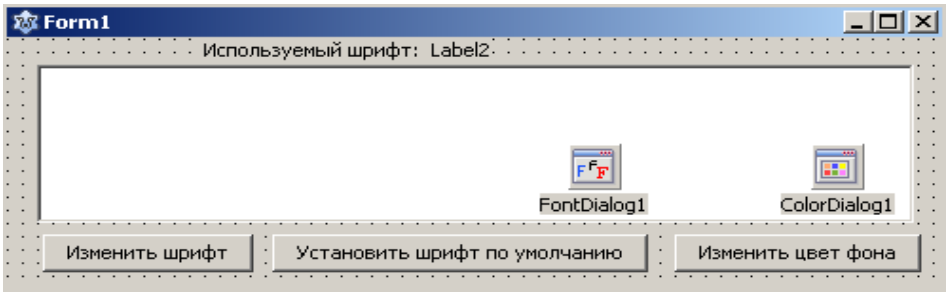
Мал. 6.41. Стандартний діалог вибору шрифту у Windows.



Мал. 6.42. Стандартний діалог вибору шрифту на Linux.

Так само користувач може встановлювати колір фону вікна TMemo.

Для цього використовується компонент вибору кольору TColorDialog, розташований у тій самій вкладці Dialogs. Видозмініть щойно розглянутий приклад. Додайте на форму компонент TColorDialog та кнопку, рис. 6.43.

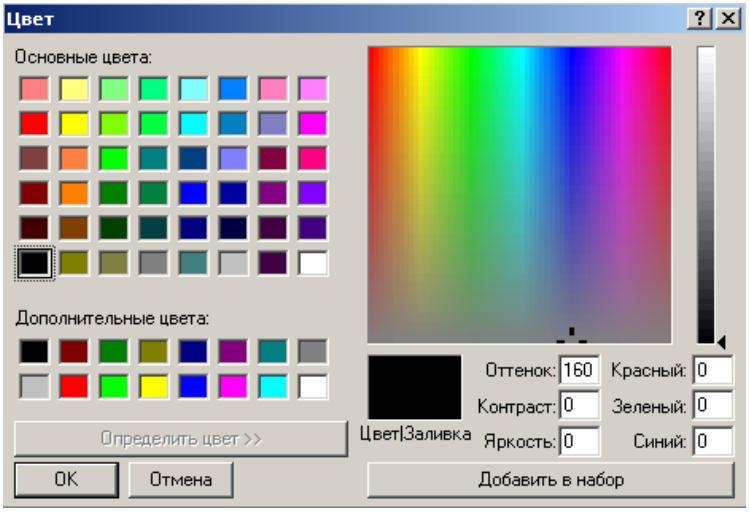


Мал. 6.43. Форма застосування

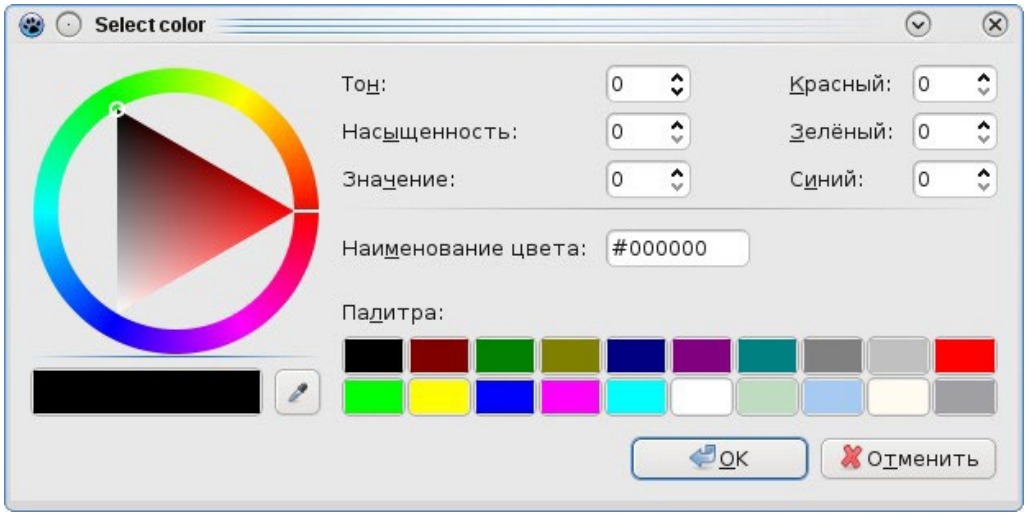
Напишіть наступний обробник для кнопки "Змінити колір тла":

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  if ColorDialog1.Execute then
  Memo1.Color:= ColorDialog1.Color;
end;
```

Тепер користувач може вибирати та встановлювати потрібний йому колір тла, рис. 6.44, 6.45.



Мал. 6.44. Стандартний діалог вибору кольору фону у Windows.



Мал. 6.45. Стандартний діалог вибору кольору фону в Linux.

Для подальшого нам знадобиться вміння працювати із класом TStringList. Клас TStringList має широкий набір властивостей та методів для роботи зі списками рядків. Розглянемо основні властивості та методи цього класу.

Властивості класу:

Таблиця 6.3

`Count: integer;`	Кількість рядків у наборі.
`Strings[Index: Integer]: string;`	Рядок з індексом Index
`Text: string;`	Весь набір рядків подається у вигляді одного рядка з роздільниками повернення каретки та перенесення рядки (#13#10) між рядками.
`Duplicates: TDuplicates;`	Дозволяє або не дозволяє додавання однакових рядків. Якщо Duplicates= dupIgnore – однаковий рядок не додається, при Duplicates= dupAccept – додавання однакових рядків у набір рядків дозволено, при Duplicates= dupError – при спробі додавання однакових рядків генерується виняток `EListError`
`Sorted: boolean`	Якщо Sorted = true – рядки набору автоматично сортуються за абеткою.

Методи класу:

Таблиця 6.4

`function Add(const S: string): integer;`	Додає рядок у набір даних та повертає її індекс
`procedure AddStrings(Strings: TStrings);`	Додає до поточного набору новий набір рядків
`procedure Append(const S:string) ;`	Додає до поточного набору новий набір рядків, але не повертає індекс вставленого рядка
`procedure Assign(Source: TPersistent);`	Знищує колишній набір рядків та завантажує з Source новий набір. У у разі невдачі виникає виняток EconvertError
`procedure Clear;`	Очищує набір даних та звільняє пов'язану з ним пам'ять
`procedure Delete(Index: integer);`	Знищує елемент набору з індексом Index і звільняє пов'язану з ним пам'ять
`function Equals(Strings:TStrings): boolean;`	Порівнює рядково поточний набір даних із набором Strings і повертає True, якщо набори ідентичні
`function IndexOf(const S:string): integer;`	Для рядка S повертає її індекс або -1, якщо такого рядка в наборі немає
`procedure Insert(Index: integer; const S: strings;`	Вставляє рядок у набір і привласнює-ет їй індекс Index
`procedure LoadFromFile (const FileName: string;`	Завантажує набір із файлу
`procedure LoadFromStream (Stream: TStream);`	Завантажує набір із потоку
`procedure Move(Curindex, Newindex: integer;`	Переміщує рядок із положення Curindex в положення Newindex
`procedure SaveToFile(const FileName: string;`	Зберігає набір у файлі
`procedure SaveToStream (Stream: TStream);`	Зберігає набір у потоці
функція `Find(const S: string; var Index: integer):Boolean;`	Пошук у відсортованому наборі рядків рядка S. Якщо такий рядок є, то його індекс міститься у параметрі Index. Якщо набір рядків не відсортований, необхідно використовувати функцію IndexOf
`Sort;`	Сортує рядки списку, властивість Sorted якого встановлена у false, у зростаючій алфавітній послідовності. Якщо Sorted = true, список сортується автоматично.

Як бачите, властивості та методи класу TStringList практично збігаються (крім деяких) з TMemo. Тільки рядки в TStringList містяться у властивості Strings, а в TMemo у властивості Lines.

Задавати значення рядкам за допомогою оператора присвоєння можна лише після того, як рядок створено, наприклад, методами Add або AddStrings.

Досить часто TMemo заповнюють вмістом якогось текстового файлу або набір рядків TStringList формується із записів текстового файлу. Для цього існує метод

```
LoadFromFile(const FileName: string);
```

де FileName ім'я файлу, включаючи шлях до нього. Якщо файл розташовано в одній папці з програмою, шлях не обов'язково вказувати.

Так само, набір рядків можна зберегти у вигляді текстового файлу мето-

буд
ино
к
```
SaveToFile(const FileName: string);
```

При роботі з текстовими файлами слід мати на увазі, що при виведенні в TMemo файлу з російським текстом, у Windows літери будуть відображатися некоректно. Це викликано тим, що текстові файли Windows мають кодування CP-1251. Необхідно спочатку перетворити вміст файлу на кодування UTF-8. Це можна зробити так:

```
procedure Ansi_UTF8;
var
  tfile: TStringList;
  str: string;
begin
```

```
  tfile:= TStringList.Create; // Створення
  списку рядків tfile.LoadFromFile('File in
  Ukrainian.txt'); str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  Memo1.Lines.Add(str);
  tfile.Free;
end;
```

Функція SysToUTF8() перетворює рядок із системного кодування (в даному випадку CP1251) на кодування UTF-8.

Набір рядків TStringList "невидимий" для користувача, на відміну від набору рядків TMemo. Зазвичай за допомогою TStringList проводять різні операції з набором рядків, а в TMemo виводять остаточно сформовані (наприклад, відсортовані) рядки. Деякі операції з набором рядків можна робити і безпосередньо в TMemo, але зазвичай так не роблять. Так як, якщо ви будете робити якісь операції з рядками в TMemo, а кількість рядків досить велика, то під час виконання вашого додатка у вікні буде щось швидко "миготіти". Зрозуміло, це недобре. Отже, операції з рядками в TMemo "на очах" у користувача є ознакою поганого тону!

Тут ми створюємо список рядків типу TStringList, методом LoadFromFile завантажуємо файл. Для цього створіть текстовий файл з ім'ям 'File in Ukrainian' з російським текстом у тій же папці, що й ваш проект. У прикладах, що містяться на прикладеному до книги диску, такий файл є. Використовуючи властивість Text, скопіюємо вміст списку в один рядок. Потім, якщо програма виконується у Windows, використовуючи функцію SysToUTF8(), перетворимо отриманий рядок на UTF-8. І тільки після цього додаємо в

Memo1. Якщо ви використовуєте Linux, то застосовувати функцію SysToUTF8() не треба, тому що рядок вже в UTF-8. В принципі, нічого страшного не станеться, якщо ви застосуєте функцію SysToUTF8() до рядка, який вже в кодуванні UTF-8. Функція просто поверне вихідний рядок, не перетворюючи його. Якщо хочете, ви можете усунути директиви компілятора

```
{$IFDEF WINDOWS}
{$ENDIF}
```

Зверніть увагу, що після завершення роботи з TStringList методом Free пам'ять, виділена набору рядків, звільняється.

Так само, при збереженні списку рядків TMemo у файл необхідно зворотне перетворення, наприклад, таким чином:

```
procedure UTF8_Ansi;
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create; // Створення списку рядків
  str:=Memo1.Text;
  {$IFDEF WINDOWS}
    str:= UTF8ToSys(str);// перетворення на системне
    кодування
  {$ENDIF}
  tfile.Add(str);
  tfile.SaveToFile('File in Ukrainian.txt');
  tfile.Free;
```

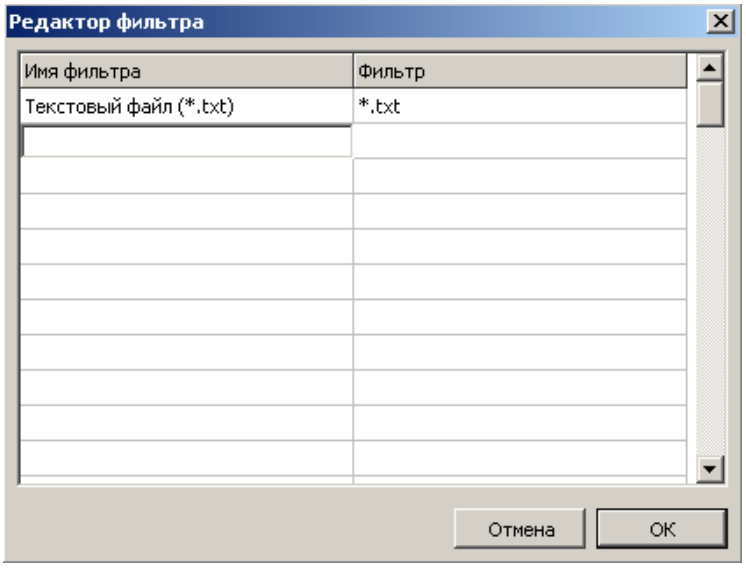
```
end;
```

У цією програмі завантажується файл з фіксованим ім'ям

'File in Ukrainian.txt', що знаходиться в тій же папці, що й сама програма. Збереження проводиться у файл із тим самим ім'ям.

Звичайно, можна надати можливість користувачеві самому вибирати файл для відкриття та задавати ім'я файлу при збереженні. Для цього використовуються компоненти TOpenDialog та TSaveDialog із вкладки Dialogs. Розглянемо деякі властивості цих компонентів:

- DefaultExt – вказує значення розширення файлу за умовчанням. У нашому випадку вкажіть DefaultExt=.txt
- Filter – задає шаблони фільтра під час вибору файлів. Якщо у вікні цієї властивості натиснути кнопку з трикрапкою, з'явиться вікно редактора фільтрів, рис. 6.46.



Мал. 6.46. Редактор фільтрів

Задайте фільтр як показано на малюнку. Тоді значення властивості дорівнює Filter= 'Текстовий файл (*.txt)|*.txt', тобто. Ім'я фільтра та сам фільтр розділені вертикальною лінією. Якщо ви поставите ще один фільтр, то вони будуть розділені крапкою з комою. Таким чином, значення якості Filter можна задавати програмно.

- FilterIndex – визначає номер фільтра, який буде за замовчуванням

показаний користувачеві на момент відкриття діалогу.

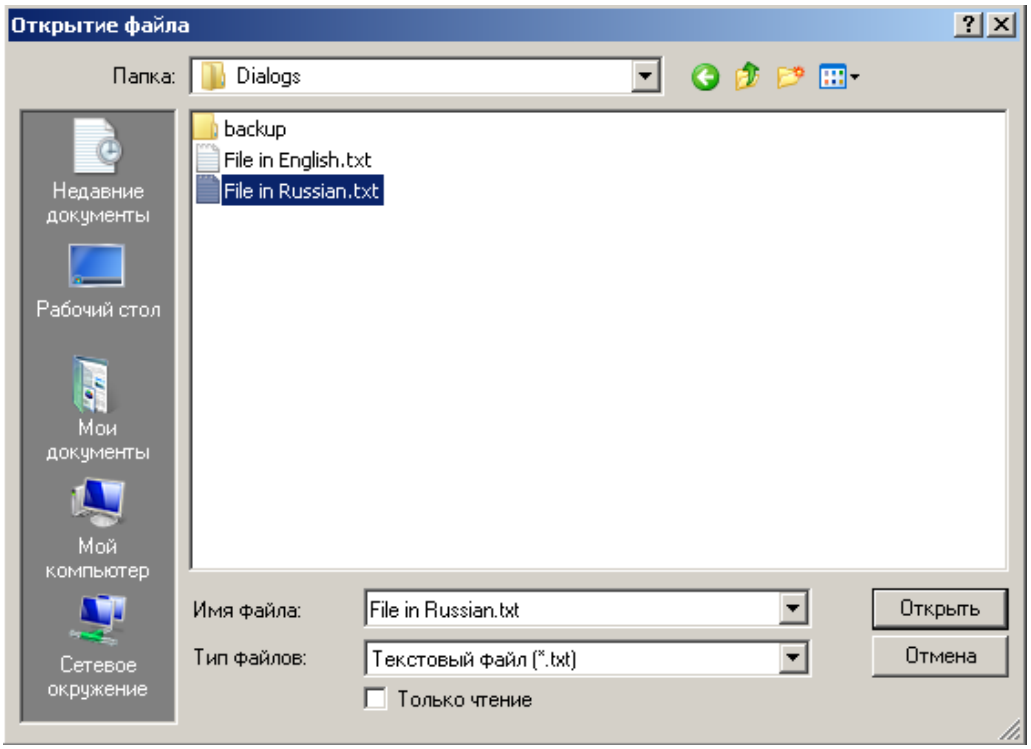
- InitialDir – визначає початковий каталог, який буде відкрито у момент початку роботи користувача з діалогом. Якщо значення цієї властивості не задано, відкривається поточний каталог або той, який був відкритий при останньому зверненні користувача до відповідного діалогу в процесі виконання цієї програми.
- Title – задає заголовок діалогового вікна.

Для виклику діалогу відкриття та завантаження файлу в TMemo слід написати наступний обробник:

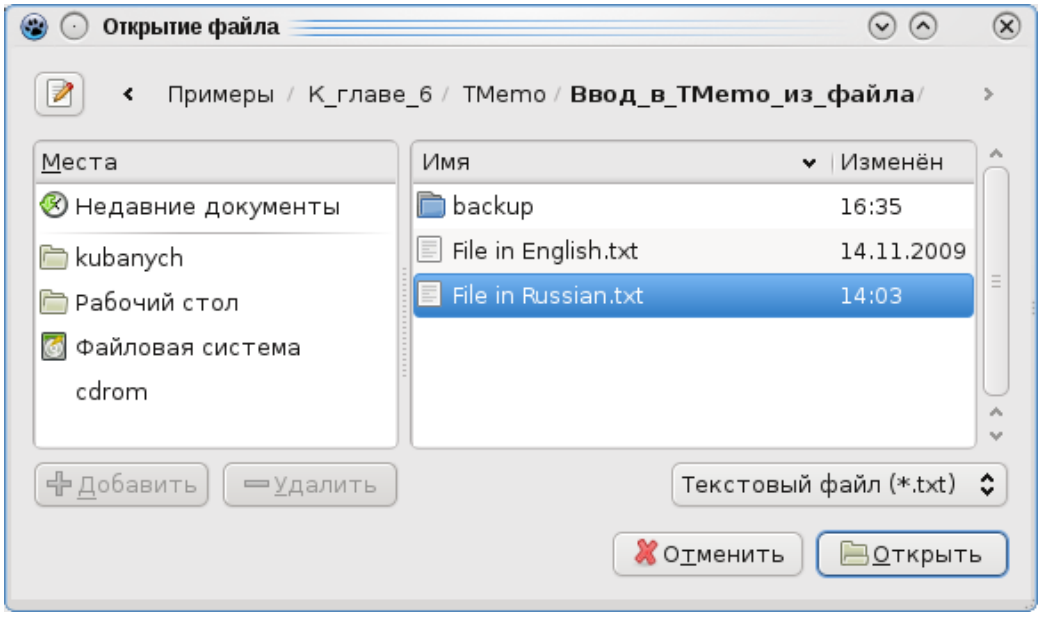
```
procedure TForm1.Button1Click(Sender: TObject);
var
  fname, str: string;
  tfile: TStringList;
begin
  tfile:= TStringList.Create; // Створення списку рядків
  if OpenDialog1.Execute
  then fname:= OpenDialog1.FileName;
  // Перетворення на системне
  кодування fname:=
  UTF8ToSys(fname);
  tfile.LoadFromFile(fname); str:
  = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  Memo1.Lines.Add(str);
  tfile.Free;
end;
```

Після запуску програми та натискання кнопки Button1 буде відкрито діа-

лог відкриття файлу, рис. 6.47, 6.48.



Мал. 6.47. Стандартний діалог для відкриття файлу в Windows.



Мал. 6.48. Стандартний діалог відкриття файлу в Linux.

Після вибору користувачем файлу, в властивості `OpenDialog1.FileName` буде знаходитись ім'я файлу разом з шляхом до не-

му. Зверніть увагу на оператора

```
fname:= UTF8ToSys(fname);
```

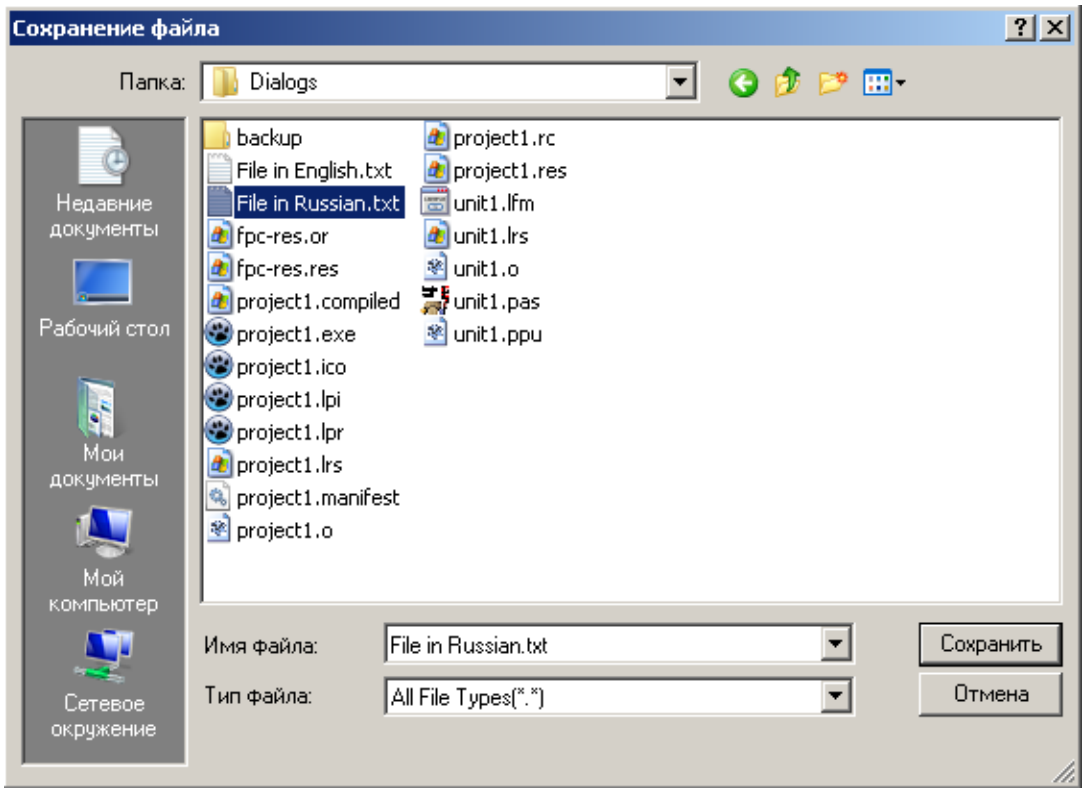
Якщо ім'я файлу, а також шлях містить кирилицю, необхідно рядок з ім'ям файлу і шляхом перетворення на системне кодування.

Для виклику діалогу збереження файлу потрібно написати наступний код:

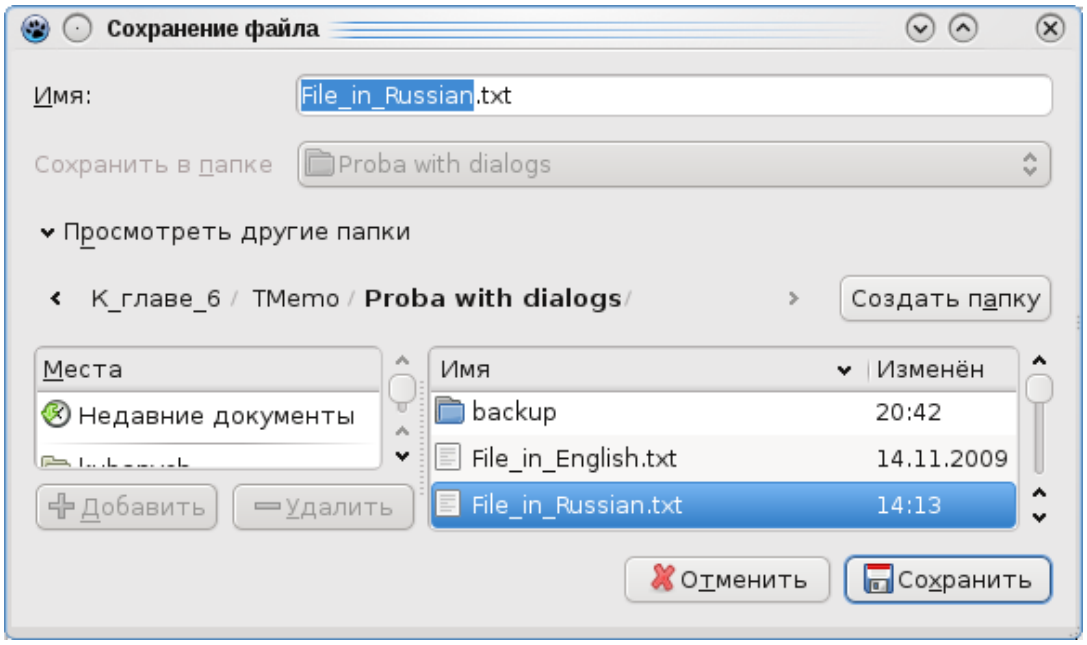
```
procedure TForm1.Button2Click(Sender: TObject);
var
  fname, str: string;
  tfile: TStringList;
begin
  tfile:= TStringList.Create; // Створення списку рядків
  str:=Memo1.Text;
  {$IFDEF WINDOWS}
    str:= UTF8ToSys(str);// перетворення на системне
    кодування
  {$ENDIF}
  tfile.Add(str);SaveDialogl.Fil
  eName:= fname; if
  SaveDialogl.Execute
  then fname:= SaveDialogl.FileName;
  // Перетворення на системне
  кодування fname:=
  UTF8ToSys(fname);
  tfile.SaveToFile(fname);
  tfile.Free;
end;
```

При натисканні кнопки Button2 буде відкрито стандартний діалог

збереження файлу, рис. 6.49, 6.50.



Мал. 6.49. Стандартний діалог збереження файлу у Windows.

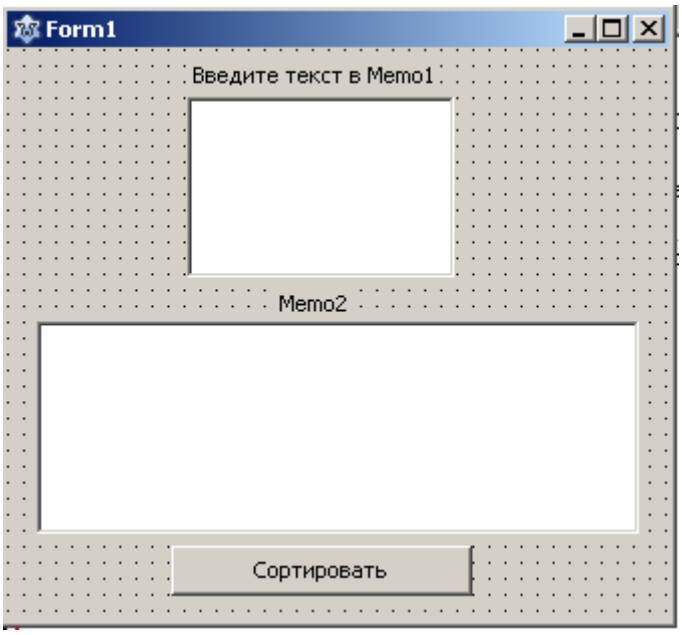


Мал. 6.50. Стандартний діалог збереження файлу у Linux.

Так, завантажувати файли в TMemo та зберігати ми навчилися. Тепер подивимося, які маніпуляції з рядками можна буде робити.

По-перше, можна сортувати. Якщо набір рядків містить лише символи латини, організувати сортування досить тривіально. Можна встановити просто властивість Sorted: true у TStringList і рядки будуть відсортовані за алфавітом автоматично. Можна застосувати метод Sort.

Створіть новий проект. Спроектуйте вікно майбутньої програми на формі так, як показано на рис. 6.51.



Мал. 6.51. Форма застосування

У обробнику натискання кнопки введіть код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  strList: TStringList;
begin
  strList:= TStringList.Create;
  strList.Duplicates:= dupAccept;
  strList.AddStrings(Memo1.Lines);
  strList.Sort;
```

```
  Memo2.Clear;
  Memo2.Lines.AddStrings(strList);
  strList.Free;
end;
```

У Memo1 введіть кілька рядків латиницею. При натисканні кнопки "Сортувати" в Memo2 відобразяться рядки, відсортовані в алфавітному порядку.

Однак цей код не працює для рядків, що містять кирилицю. Це відбувається, знову ж таки, через кодування UTF-8. За умовчанням TStringList використовує наступну функцію, яка працює з рядками ANSI:

```
function TStringList.DoCompareText(const s1,s2 : string)
: PtrInt;
begin
    if FCaseSensitive then
       result:= AnsiCompareStr(s1,s2)
    else
       result:= AnsiCompareText(s1,s2);
end;
```

До щастя, серед методів класу TStringList є метод CustomSort. Його оголошення в класі має вигляд:

```
type
TStringListSortCompare = function(List: TStringList;
Indexl, Index2: integer): integer;
procedure CustomSort(CompareFn: TStringListSortCompare);
```

Можна скористатися методом CustomSort передавши йому як параметр свою функцію порівняння. Щоб відсортувати список,

побудуєм

o

функцію порівняння двох елементів S1 та S2 з індексами indexl та index2 відповідно, яка має повертати:

< 0 - якщо, S1 < S2;

0 - якщо, S1 = S2;

> 0 - якщо, S1 > S2;

Функція має вигляд:

```
function ListSortRus(List: TStringList; Index1, Index2:
integer): integer;
begin
  Result:= WideCompareText(UTF8Decode(List[Index1]),
           UTF8Decode(List[Index2])));
end;
```

Тут використовується функція:

функція `WideCompareText(const S1:WideString; const S2:WideString): integer;`

Вона працює з рядками WideString. Тип WideString являє собою рядки, що динамічно розміщуються в пам'яті, довжина яких обмежена тільки обсягом вільної пам'яті. Кожен символ рядка типу WideString є символом Unicode. Функція повертає результат:

< 0 - якщо, S1 < S2;

0 - якщо, S1 = S2;

> 0 - якщо, S1 > S2;

Це саме те, що нам потрібне! У розділі implementation додайте опис функції

```
function ListSortRus(List: TStringList; Index1, Index2:
integer): integer;
begin
  Result:= WideCompareText(UTF8Decode(List[Index1]),
           UTF8Decode(List[Index2])));
end;
```

Функція

UTF8Decode(const S: UTF8String): WideString - перетворює рядок формату UTF-8 на рядок формату Unicode.

Перепишіть кнопки OnClick обробник:

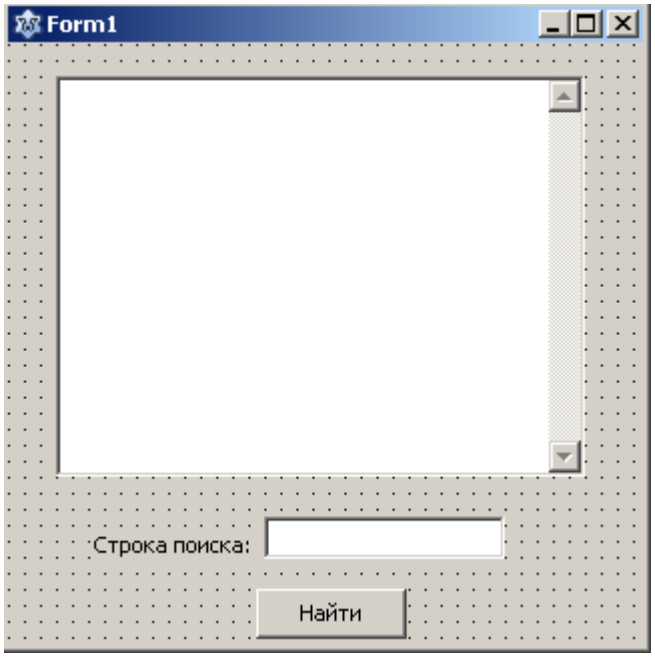
```
procedure TForm1.Button1Click(Sender: TObject);
var
  strList: TStringList;
begin
  strList:= TStringList.Create;
  strList.Duplicates:= dupAccept;
  strList.AddStrings(Memo1.Lines);
  strList.CustomSort(@ListSortRus);
  Memo2.Clear;
  Memo2.Lines.AddStrings(strList);
  strList.Free;
end;
```

В обробнику змінився лише один рядок. Замість методу Sort викликається CustomSort(@ListSortRus). Як бачите, щоб передати функцію як параметр, перед ім'ям функції поставити знак @.

Тепер сортування правильно працюватиме як для латиниці, так і для кирилиці.

По-друге, досить часто необхідно шукати. Причому не лише окремих рядків цілком, а й деякої частини рядка (кажуть ще підрядки). Якщо говорити про текст, необхідно знайти слово, фразу чи речення, причому речення може займати кілька рядків.

Для пошуку зручно використовувати властивість Text компонента TMemo в якому весь набір рядків подається у вигляді одного довгого рядка та функції Pos, яку ми вивчали в 3.3. Складемо програму пошуку слів у текстовому файлі. Створіть новий проект. Помістіть на форму компоненти, як показано на рис. 6.52.



Мал. 6.52. Форма застосування

Введіть наступний код:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  Memo1.Lines.LoadFromFile('File in English.txt');
  Edit1.SetFocus;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  k: integer;
begin
  Edit1.SetFocus;
  Edit1.SelectAll;
  if Edit1.Text= '' then
  begin
    ShowMessage('Введіть рядок для
    пошуку'); exit;
  end;
  k:= Pos(Edit1.Text, Memo1.Text);
  if k > 0 then
  begin
    Memo1.SetFocus;Memo1.Sel
    Start: = k - 1;
    Memo1.SelLength:= Length(Edit1.Text);
  end
  else
    ShowMessage('Рядок не
знайдено'); end;
```

Під час створення вікна програми Memo1 завантажується текстовий файл. У обробнику натискання кнопки здійснюється пошук фрагмента тексту, введеного користувачем у полі Edit1. Для виділення знайденого фрагмента використовуються властивості Memo1.SelStart та Memo1.SelLength. Щоб знайдений текст був виділений, необхідно, щоб фокус було встановлено в Memo1. У цьому прикладі ми здійснювали пошук у текстовому файлі, в якому

міститься лише англійські літери. Для тексту, що містить кирилицю, наша програма працювати коректно не буде.

Тому нам доведеться змінити програму для пошуку в текстових файлах, що містять кирилицю. У обробнику OnCreate форми завантажуємо кириличний текст Memo1, попередньо перетворивши його в UTF-8 (для Windows). У обробнику OnClick ми використовуємо вже знайому нам функцію UTF8Decode, а також функції UTF8Pos та UTF8Length. Не забудьте увімкнути модуль LCLProc в оголошення uses.

```
procedure TForm1.FormShow(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('File in Ukrainian.txt');
  str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  Memo1.Lines.Add(str);
  tfile.Free;
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  k: integer;
  text_UTF8, str_UTF8: string;
begin
  Edit1.SetFocus;
```

```
  if Edit1.Text= '' then
  begin
    ShowMessage('Введіть рядок для
    пошуку'); exit;
  end;
  text_UTF8:= UTF8Decode(Memo1.Text);
  str_UTF8:= UTF8Decode(Edit1.Text);
  k:= UTF8Pos(str_UTF8, text_UTF8);
  if k > 0 then
  begin
    Memo1.SetFocus;Memo1.Sel
    Start: = k - 1;
    Memo1.SelLength:= UTF8Length(str_UTF8);
  end
  else
    ShowMessage('Рядок не
знайдено'); end;
```

Дозволимо користувачеві закінчувати введення Edit1 натисканням клавіші Enter. Щоб уникнути дублювання коду, створимо процедуру CyrillicSearch і в обробниках викликатимемо цю процедуру:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls,
  LconvEncoding, LCLProc;
```

```
type
  {TForm1}

  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure CyrillicSearch;
    procedure FormShow(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{TForm1}
procedure TForm1.CyrillicSearch;
var
  k: integer;
  text_UTF8, str_UTF8: string;
begin
  Edit1.SetFocus;
  if Edit1.Text= '' then
```

```
  begin
    ShowMessage('Введіть рядок для
    пошуку'); exit;
  end;
  text_UTF8:= UTF8Decode(Memo1.Text);
  str_UTF8:= UTF8Decode(Edit1.Text);
  k:= UTF8Pos(str_UTF8, text_UTF8);
  if k > 0 then
  begin
    Memo1.SetFocus;Memo1.Sel
    Start: = k - 1;
    Memo1.SelLength:= UTF8Length(str_UTF8);
   end
  else
    ShowMessage('Рядок не
знайдено'); end;

procedure TForm1.FormShow(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('File in Ukrainian.txt');
  str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  Memo1.Lines.Add(str);
```

```
  tfile.Free;
  Edit1.SetFocus;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  CyrillicSearch;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key = #13 then
  begin
    Key:= #0;
    CyrillicSearch;
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

За такої реалізації наша програма знаходить лише перше входження шуканої підрядки. Щоб знайти всі входження будемо "годувати" функції Pos рядок щоразу без останнього знайденого підрядка. Для цього використовуємо властивості SelStart та SelLength Memo1 та функцію UTF8Copy таким чином:

```
text_UTF8:=UTF8Copy(UTF8Decode(Memo1.Text),
```

```
            Memo1.SelStart + 1 + Memo1.SelLength,
            Length(UTF8Decode(Memo1.Text)) -
            Memo1.SelStart - Memo1.SelLength);
```

Функцію UTF8Copy ми розглядали у 3.3.1.4. Щоб виділити в Memo1 наступний знайдений підряд оператор

```
Memo1.SelStart: = k - 1;
```

треба замінити на

```
Memo1.SelStart:=Memo1.SelStart +
                Memo1.SelLength + k – 1;
```

Тепер програма знаходить всі входження заданого підрядка Memo1. Для знаходження наступного входження потрібно натискати кнопку "Знайти". Натискати клавішу Enter не можна, оскільки фокус знаходиться в Memo1. Щоб продовжити пошук натисканням Enter, треба клавішею Tab або мишею встановити фокус на Edit1. З'являється деяка незручність у роботі. Якщо ж ми в програмі відразу після знаходження підрядка та його виділення передамо фокус Edit1, то із знайденого фрагмента виділення буде знято. Що теж недобре.

Можна додати обробник Memo1KeyPress, щоб тільки при натисканні Enter передати фокус Edit1. Отже, код покращеної програми (попередньо встановіть у Memo1 властивість WantReturns=false):

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
```

```
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, StdCtrls, LconvEncoding,
  LCLProc;
type
  {TForm1}

  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure CyrillicSearch;
    procedure FormShow(Sender: TObject);
    procedure Memo1KeyPress(Sender: TObject;
                            var Key: char);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{TForm1}

procedure TForm1.CyrillicSearch;
```

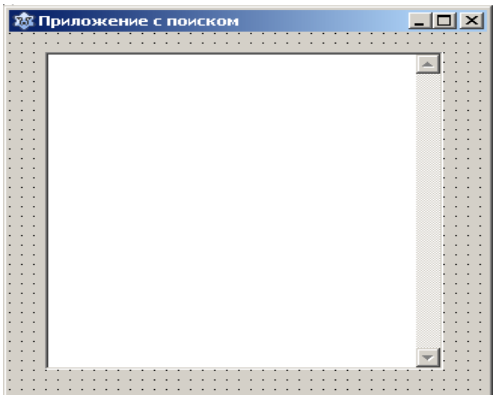
```
var
  k: integer;
  text_UTF8, str_UTF8: string;
begin
  Edit1.SetFocus;
  if Edit1.Text= '' then
  begin
    ShowMessage('Введіть рядок для
    пошуку'); exit;
  end;
  text_UTF8:= UTF8Copy(UTF8Decode(Memo1.Text),
              Memo1.SelStart + 1 + Memo1.SelLength,
              UTF8Length(UTF8Decode(Memo1.Text)) -
              Memo1.SelStart - Memo1.SelLength);
  str_UTF8:= UTF8Decode(Edit1.Text);
  k:= UTF8Pos(str_UTF8, text_UTF8);
  if k > 0 then
  begin
    Memo1.SetFocus;
    Memo1.SelStart:= Memo1.SelStart +
                     Memo1.SelLength + k – 1;
    Memo1.SelLength:= UTF8Length(str_UTF8);
   end
  else
  begin
    ShowMessage('Рядок не знайдено');
    Memo1.SelStart: = 0;
    Memo1.SelLength: = 0;
  end;
```

```
end;

procedure TForm1.FormShow(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('File in Ukrainian.txt');
  str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  Memo1.Lines.Add(str);Memo1
  .SelStart: = 0;
  Memo1.SelLength: = 0;
  tfile.Free;
  Edit1.SetFocus;
end;

procedure TForm1.Memo1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key = #13 then
     Edit1.SetFocus;
     CyrillicSearch;
end;

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

```
begin
  CyrillicSearch;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key = #13 then
  begin
    Key:= #0;
    CyrillicSearch;
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

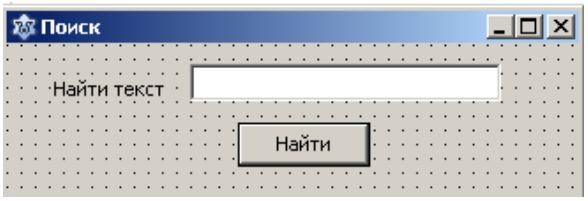
Практично у всіх текстових редакторах функцію пошуку можна запустити, використовуючи клавіші Ctrl+F. А продовження пошуку натисканням кнопки F3. Візьміть, наприклад, Word, Блокнот або Редактор вихідного коду Lazarus.

Давайте і ми реалізуємо такий же виклик пошуку у TMemo. Заодно навчимося використовувати у програмі кілька форм. Створіть новий проект. Покладіть на форму компонент TMemo, рис. 6.53.



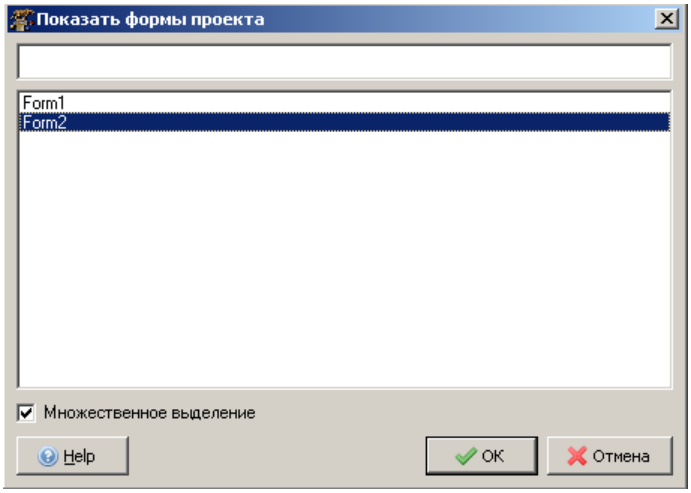
Мал. 6.53. Головна форма програми

У меню "Файл" виберіть "Створити форму". У вікні редактора вихідного коду з'явиться нова порожня форма та відповідний модуль з іменем Unit2. Помістіть на другу форму напис, TEdit та кнопку, рис. 6.54.



Мал. 6.54. Друга форма програми

Якщо у вас у додатку кілька форм, то для відкриття потрібної натисніть Shift+F12 і у вікні виберіть потрібну вам форму, рис. 6.55.



Мал. 6.55. Вікно вибору форми

У `Unit1` після ключового слова `implementation` помістіть

```
uses Unit2;
```

А в Unit2 після ключового слова implementation помістіть

```
uses Unit1;
```

Раніше, для розпізнавання натиснутого символу ми використовували обробник події OnKeyPress, однак при цьому не можна розпізнати натискання функціональних і службових клавіш. Для їхнього розпізнавання підходить подія OnKeyDown.

Заголовок обробника події OnKeyDown має такий вигляд:

```
procedure TForm1.Memo1KeyDown(Sender: TObject;
          var Key: Word; Shift: TShiftState);
```

Параметр Key визначає натиснуту під час події клавішу. Так само як і для події OnKeyPress визначено як var, тобто. може змінюватися в обробнику події. Але зверніть увагу, що це ціле число, а не символ. Коди клавіш можна вказувати у десятковому або шістнадцятковому вигляді. Для деяких клавіш введено також іменовані константи, які полегшують написання програми, оскільки не вимагають пам'ятати чисельні коди клавіш. Наприклад, замість

```
if (Key = 13) then ...;
```

можна записати

```
if (Key = VK_RETURN) then ...;
```

Для решти клавіш зручніше використовувати функцію ord, яка по-

обертає символ коду.

Параметр Shift – це безліч елементів, що відображають натиснуті в цей час службові клавіші. Елементи цієї множини ssShift – натиснута клавіша Shift, ssAlt – натиснута клавіша Alt та ssCtrl – натиснута клавіша Ctrl. Оскільки Shift є безліччю, перевіряти наявність у ньому тих чи інших елементів треба операцією in. Таким чином, для визначення натискання клавіш Ctrl+F можна написати наступний код:

```
if((Key = ord('F')) and (ssCtrl in Shift)) then...;
```

При натисканні Ctrl+F відкриватимемо вікно діалогу пошуку. Вікно відкриватимемо як модальне методом ShowModal. Модальним вікном називається таке вікно, в якому користувач не може перемикатися на інші вікна, доки не закриє поточне вікно. Код програми пошуку у TMemo складатиметься з двох модулів.

Модуль Unit1:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, StdCtrls, LconvEncoding,
  LCLProc;
type
  {TForm1}
  TForm1 = class(TForm)
    Memo1: TMemo;
```

```
    procedure FormCreate(Sender: TObject);
    procedure Memo1KeyDown(Sender: TObject;
                           var Key: Word;
                           Shift: TShiftState);
    procedure CyrillicSearch;

  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
uses Unit2;
{TForm1}

procedure TForm1.CyrillicSearch;
var
  k: integer;
  text_Area: string;
begin
  text_Area:= UTF8Copy(UTF8Decode(Memo1.Text),
              Memo1.SelStart + 1 + Memo1.SelLength,
              UTF8Length(UTF8Decode(Memo1.Text)) -
              Memo1.SelStart - Memo1.SelLength);
  k:= UTF8Pos(str_Search, text_Area);
  if k > 0 then
  begin
```

```
    Memo1.SetFocus;
    Memo1.SelStart:= Memo1.SelStart +
                     Memo1.SelLength + k – 1;
    Memo1.SelLength:= UTF8Length(str_Search);
  end
  else
  begin
    ShowMessage('Рядок не знайдено');
    Memo1.SelStart: = 0;
    Memo1.SelLength: = 0;
  end;
end;
procedure TForm1.Memo1KeyDown(Sender: TObject;
                              var Key: Word;
                              Shift: TShiftState);
begin
  if((Key = ord('F')) and (ssCtrl in Shift)) then
  begin
    Memo1.SelStart: = 0;
    Memo1.SelLength: = 0;
    Form2.ShowModal;
   end
  else
  якщо Key = 114 then // десятковий код клавіші F3
     CyrillicSearch;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  tfile: TStringList;
```

```
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('File in Ukrainian.txt');
  str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  Memo1.Lines.Add(str);
  tfile.Free;
end;
initialization
  {$I unit1.lrs}
end.
```

Модуль Unit2:

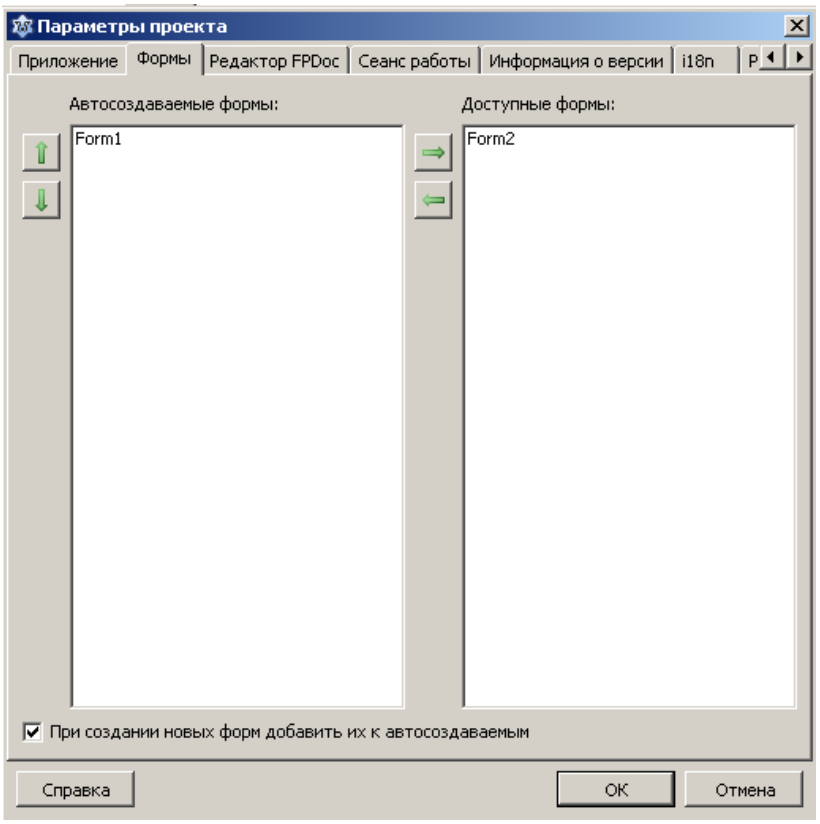
```
unit Unit2;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls;

type

  {TForm2}

  TForm2 = class(TForm)
```

```
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure GetSearchString;
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form2: TForm2;
  str_Search: string;

implementationus
es Unit1;
{TForm2}
procedure TForm2.GetSearchString;
begin
  if Edit1.Text='' then
  begin
    ShowMessage('Введіть рядок для
    пошуку'); exit;
  end;
  Edit1.AutoSelect:= true;
  if str_Search <> UTF8Decode(Edit1.Text) then
  begin
```

```
    str_Search:= UTF8Decode(Edit1.Text);
  end;
  Form2.Close; // Закриття форми
  Form1.CyrillicSearch;
end;
procedure TForm2.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key = #13 then
  begin
    Key:= #0;
    GetSearchString;
  end;
end;

procedure TForm2.Button1Click(Sender: TObject);
begin
  GetSearchString;
end;
initialization
  {$I unit2.lrs}
end.
```

У нашому додатку передбачається, що користувач обов'язково скористається функцією пошуку. Але ж це відбувається не завжди! Згадайте, чи часто ви користуєтесь пошуком? А за замовчуванням усі форми створюються в момент запуску програми. Значить, якщо користувач не скористався пошуком, форма Form2 буде створена "даремно"! Вона займатиме зайву пам'ять. У хороших програмах такого не повинно бути, особливо якщо є багато

форм. Вихід полягає в тому, щоб створювати форму тоді, коли це необхідно. Для цього у властивостях проекту (меню Проект->Параметри проекту->Форми) приберіть Form2 зі списку автостворюваних форм до списку доступних, рис. 6.56.



Мал. 6.56. Вікно списку автостворюваних форм

Будемо створювати Form2 тільки тоді, коли користувач натисне комбінацію клавіш Ctrl+F. В обробнику події Memo1KeyDown перед оператором

```
Form2.ShowModal;
```

запишіть оператор

```
Application.CreateForm(TForm2, Form2);
```

Після того, як нам форма вже не потрібна, ми її закриваємо оператором

```
Form2.Close;
```

Але закриття форми ще означає, що вона видалена з пам'яті. Вона ще "жива зараза" і ми можемо у будь-який момент її знову показати методом ShowModal на випадок, якщо користувач повторно запустить функцію пошуку. Форму буде "остаточно" знищено після закриття головного вікна програми.

Отже, перед створенням форми ми маємо перевірити, можливо, вона вже була створена. Тепер уже перед оператором

```
Application.CreateForm(TForm2, Form2);
```

вставте оператор

```
if not Assigned(Form2) then
```

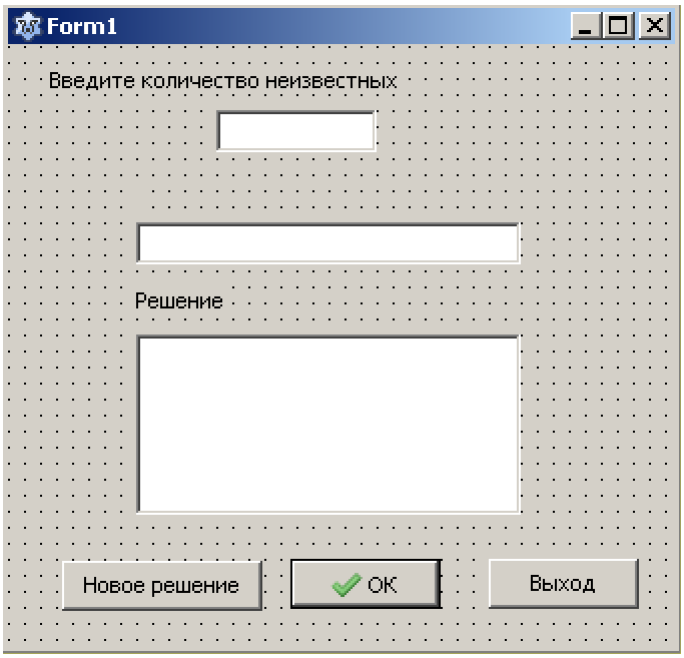
Наведу остаточну редакцію обробника Memo1KeyDown: procedure

```
TForm1.Memo1KeyDown(Sender: TObject;
                                var Key: Word;
                                Shift: TShiftState);
begin
  if((Key = ord('F')) and (ssCtrl in Shift)) then
  begin
    Memo1.SelStart: = 0;
    Memo1.SelLength: = 0;
    if not Assigned(Form2) then
      Application.CreateForm(TForm2, Form2);
      Form2.ShowModal;
```

```
   end
  else
  якщо Key = 114 then // десятковий код клавіші F3
     CyrillicSearch;
end;
```

На закінчення, напишемо програму розв'язання системи лінійних рівнянь алгебри методом Гауса з використанням TMemo. Консольний варіант програми ми розглядали у 1.3.4. та 3.4.2.

Спочатку наведу програму, написану одним із моїх студентів. Потім ми разом розберемо недоліки програми, а потім спробуємо збільшити цю програму. Форма програми має вигляд, рис. 6.57.



Мал. 6.57. Форма застосування

Код програми наступний:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls,
  Graphics, Dialogs, StdCtrls, Buttons;
```

```
type

  {TForm1}

  TForm1 = class(TForm)
    BitBtn1: TBitBtn;
    Button1: TButton;
    Button2: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Memo1: TMemo;
    procedure BitBtn1Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure Edit2Change(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Calculate;
  private
    { private declarations }
  public
    A, b, x: TStrings;
    v, v2: integer;
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{TForm1}

procedure Tform1.Calculate;
var i,j,n,k,p,h,r:integer;
    t,m,s:extended;
begin
  Memo1.Clear;
  n:=StrToInt(Edit1.Text);
```

```
for h:=0 to n-1 do
begin
  x.Add('0');
end;
for k:=1 n-1 do
begin
  for i:=k+1 n do begin
    if a.Strings[(k-1)*n+(k-1)]='0' then
    begin
      {Початок блоку перестановки рівнянь}

      p:=k; {У блоковій схемі
             використовується буква l,
             але вона схожа на цифру 1,
             тому використовуємо ідентифікатор p}

      for r:=i до n do
      if abs(StrToFloat(a.Strings[(r-1)*n+(k-1)]))>
      abs(StrToFloat (a.Strings[(p-1)*n+(k-1)]))
      then p:=r;
      if p<>k then
      begin
        for j:=k n do;
        begin
          t:=StrToFloat(a.Strings[(k-1)*n+(j-1)]);
          a.Strings[(k-1)*n+(j-1)]:=
          a.Strings [(p-1) * n + (j-1)];
          a.Strings[(p-1)*n+(j-1)]:=FloatToStr(t);
        end;
          t:=(StrToFloat (b.Strings[k-1]));
          b.Strings[k-1]:=b.Strings[p-
          1];b.Strings[p-
          1]:=FloatToStr(t);
      end;
    end; // Кінець блоку перестановок рівнянь
    m:=StrToFloat (a.Strings[(i-1)*n+
    (k-1)])/StrToFloat (a.Strings[(k-1)*n+(k-1)]);
    a.Strings[(i-1)*n+(k-
    1)]:='0';for j:=k+1 to n do
    begin
      a.Strings[(i-1)*n+(j-1)]:=
      FloatToStr(StrToFloat(a.Strings[(i-
      1))
      *n+(j-1)])-(m*StrToFloat(a.Strings[(k-1)*
```

```
    n+(j-1)])))));
```

```
      end;
      b.Strings[i-1]:=FloatToStr(StrToFloat
      (b.Strings[i-1])-
      (m*StrToFloat(b.Strings[k-1])));
    end;
  end;
  {Перевірка існування рішення}
  if StrToFloat (a.Strings[(n-1)*n+(n-1)])<>0 then
  {Рішення існує і єдино} begin
    x.Strings[n-1]:=(FloatToStr(StrToFloat
    (b.Strings[n-1]))/StrToFloat(a.Strings[(n-
    1)*)
     n+(n-1)]));
    for i:=n-1 downto 1 do
    begin
      s:=0;
      for j:=i+1 до n до
      початку
        s:=s-StrToFloat (a.Strings[(i-1)*n+(j-1)])*
             StrToFloat (x.Strings[j-1]);
      end;
      x.Strings[i-1]:=(FloatToStr((StrToFloat
      (b.Strings[i-1])+s)/
      StrToFloat (a.Strings[(i-1)*n+(i-1)])));
    end;
    Memo1.Lines.Add('Рішення:');f
    or j:=0 to x.Count-1 do
       Memo1.Lines.Add('X'+IntToStr(j+1)+
                       '='+x.Strings[j]);
  end
  else
  if StrToFloat (b.Strings[n-1])=0 then
     Memo1.Lines.Add('Система рівнянь'+
                     'не має рішення.')
  else
     Memo1.Lines.Add('Система рівнянь'+
                  'має безліч рішень.');
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  A:=TStringList.Create;
  b:=TStringList.Create;
  x:=TStringList.Create;
```

```
  v:=1;
  v2: = 1;
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
  if Edit2.Text = '-' then exit;
  StrToFloat (Edit2.Text);
  BitBtn1.Enabled:=true;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
var n:integer;
begin
  if (Edit2.Text='-') or (Edit2.Text='') then exit;
  begin
    n:=StrToInt(Edit1.Text);
    Edit2.SetFocus;
    Edit2.SelectAll;
    if v2=1 then
      Edit1.Enabled:=false;
    if v2<=n+1 then
    begin
      if v<=n+1 then
        v:=v+1
      else
        v:=1;
      if v<=n+1 then
      begin
        if v<=n then
        Label3.Caption:='Введіть a' +
                      IntToStr(v2) +
                        IntToStr(v);
        якщо в>n then
          Label3.Caption:='Введіть b'+IntToStr(v2);
        A.Add(Edit2.Text);
      end;
      якщо v>n+1
      then begin
        b.Add(Edit2.Text);
        v2:=v2+1;
        v:=1;
        if v2<=n then
```

```
          Label3.Caption:='Введіть a' +
                        IntToStr(v2)+
                        IntToStr(v)
        else
        begin
          Label3.Caption:='';
          calculate;
          BitBtn1.Enabled:=false;
          Button1.SetFocus;
        end;
      end;
    end;
  end
  else
  Edit2.SetFocus;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  b.Clear;
  a.Clear;
  x.Clear;
  Memo1.Clear;
  Edit1.Enabled:=true;
  Edit1.Clear;
  Edit2.Clear;
  Edit1.SetFocus;
  v:=1;
  v2: = 1;
  Label3.Caption:='';
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.Edit1Change(Sender: TObject);
begin
  if strtoint(Edit1.Text)>1 then
  begin
    BitBtn1.Enabled:=true;
    Edit2.Enabled:=true;
    Edit2.Clear;
```

```
      Label3.Caption:='Введіть a11';
      Edit2.SetFocus;
    end
    else
    begin
      BitBtn1.Enabled:=false;
      Edit2.Enabled:=false;
    end;
  end;

  procedure TForm1.Edit1KeyPress(Sender: TObject;
                                 var Key: char);
  begin
    if Key = #13 then Key := #0;
  end;
  initialization
    {$I Unit1.lrs}
  end.
```

Проаналізуємо програму з двох точок зору – з погляду користувача та з погляду програміста.

З точки зору користувача ми бачимо, що програма може вирішувати систему рівнянь не більше ніж із дев'ятьма невідомими. Спробуйте ввести двозначне число. У вас не вийде!

По-друге, хоча введення коефіцієнтів і організоване більш-менш задовільно, на жаль, у користувача немає можливості оглянути раніше введені коефіцієнти. При великій кількості рівнянь тут не дивно припуститися помилок, переплутати коефіцієнти. Хоча підказка, який коефіцієнт треба вводити в даний момент, є.

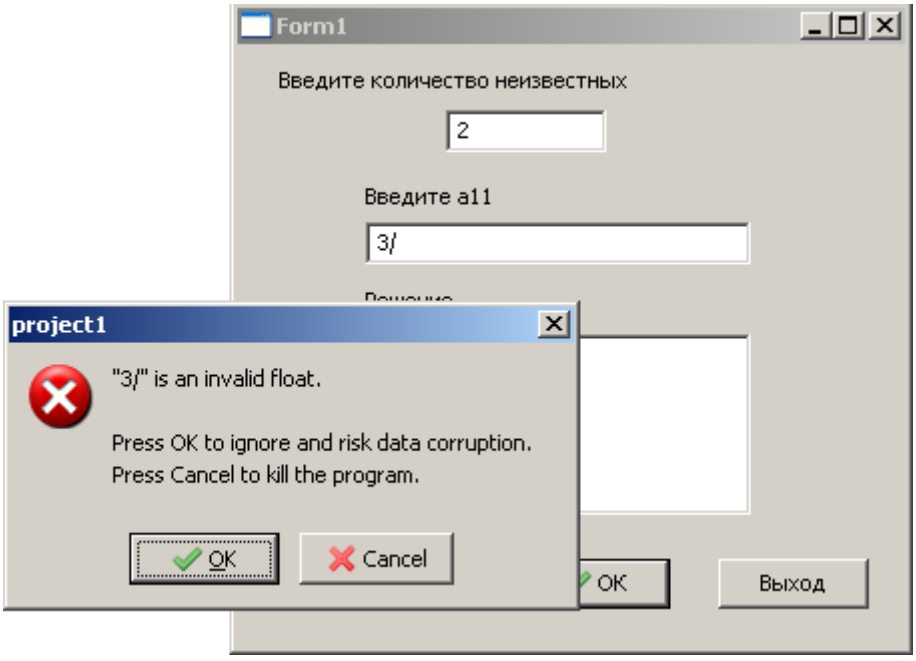
Звернемося тепер до коду програми. У програмі використається клас TStringList. Але, взагалі кажучи, клас TStringList не призначений для обчислень! Він призначений обробки рядків, тобто. для обробки символьної інформації. Подивіться скільки тут проводиться перетворень із символьної вистави в числове і назад.

У програмах такого роду, де здебільшого виробляються обчислення,

зазвичай перетворення виконуються лише двічі! На початку роботи програми (при введенні вихідних даних) рядки символів перетворюються на числа, виконуються всі необхідні обчислення і в кінці роботи програми одержані результати (числові) перетворюються на їх рядкове подання для виведення на екран або принтер. За всіх інших рівних умов, ця програма працюватиме на порядок (якщо не більше) повільніше, ніж програма, в якій спочатку всі дані перетворені на числа.

Тепер подивіться на код, яким реалізовано введення коефіцієнтів. Там "саме чорт ногу зломить"! Використовується дуже багато умовних операторів.

Крім того, у програмі геть-чисто відсутній контроль даних, що вводяться. Хоча стандартна реакція системи допоможе уникнути аварійного завершення програми, рис. 6.58.



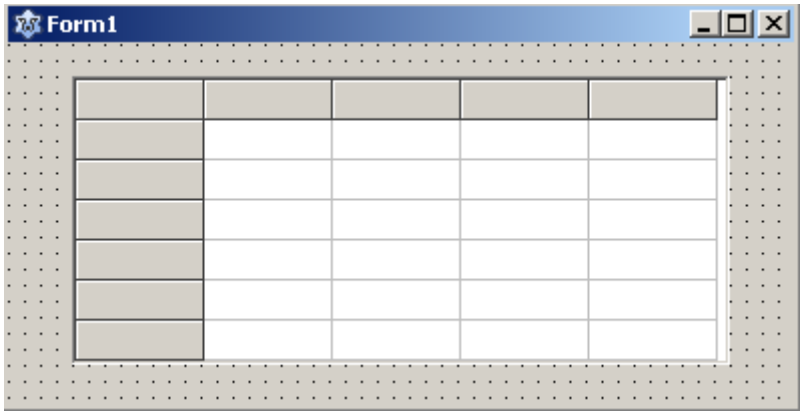
Мал. 6.58. Приклад виникнення виключення

Натиснувши кнопку OK, користувач може ввести правильне число.

Давайте виправимо програму з урахуванням зроблених вище зауважень. Для того, щоб користувач міг бачити всі введені коефіцієнти, а також редагувати їх зручніше застосувати компонент TStringGrid. Тому спочатку познайомимося з цим компонентом.

6.3.10.2. Компонент TStringGrid

TStringGrid знаходиться у вкладці Additional і має вигляд, рис. 6.59.



Мал. 6.59. Вигляд компонента TStringGrid

Компонент є таблицею, що складається з рядків Rows і стовпців Cols. Своєю чергою таблиця це двомірний масив, значеннями якого є рядки символів і, отже, має тип string. Доступ до даних здійснюється через властивість Cells. Осередку таблиці, що знаходиться на перетині стовпця з номером Col і рядка з номером Row, відповідає елемент масиву Cells[Col,Row]. Зверніть увагу, спочатку вказується стовпець, а потім рядок. Нумерація стовпців та рядків починається з нуля. Основні властивості компонента TStringGrid такі:

- ColCount – кількість стовпців таблиці;
- RowCount – кількість рядків таблиці;
- FixedCols – кількість фіксованих шпальт таблиці. Зазвичай фіксується один, найлівіший стовпець і використовується для постійної інформації, наприклад, заголовка стовпця. Але можна зафіксувати і більше шпальт. При цьому зафіксовані стовпці виділяються кольором і при горизонтальному прокручуванні таблиці залишаються на місці;
- FixedRows – кількість фіксованих рядків таблиці. Так

самo,

фіксується зазвичай один рядок для заголовка, але можна зафіксувати і більше рядків. Рядки виділяються кольором і при вертикальному прокручуванні таблиці залишаються на місці;

- FixedColor – колір фіксованих рядків та стовпців.
- VisibleColCount - кількість видимих (прокручуються) стовпців, так само ColCount - FixedCols;
- VisibleRowCount – кількість видимих (прокручуваних) рядків, що дорівнює

RowCount - FixedRows;

- ScrollBars – визначає наявність у таблиці смуг прокручування. Якщо вказати значення ssAutoBoth, смуги прокручування будуть з'являтися і зникати автоматично залежно від того, чи таблиця поміщається у вікно компонента чи ні.

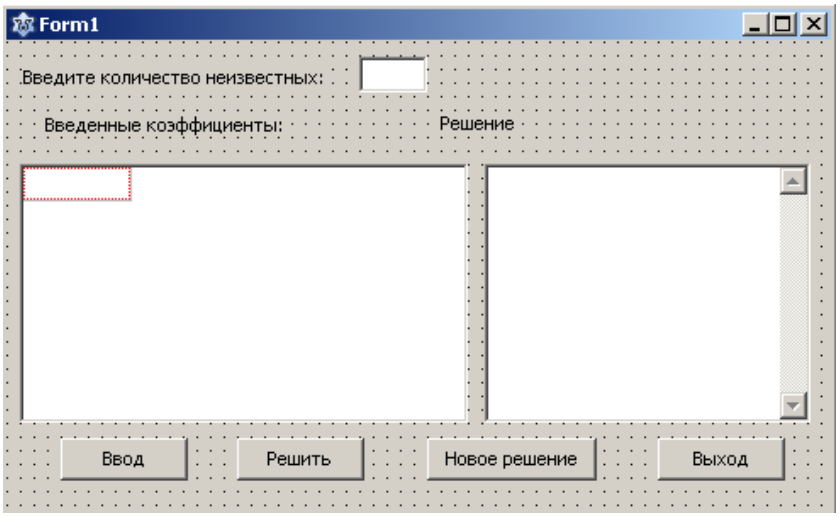
У вкладці Options властивостей TStringGrid визначено ряд властивостей, найважливішими з яких є:

- goEditing – дозволяє або забороняє редагування вмісту осередків таблиці. true – редагування дозволено, false – заборонено;
- goTab – дозволяє (true) або забороняє (false) використання клавіші <Tab> для переміщення курсору до наступного осередку таблиці;
- goAlwaysShowEditor – ознака знаходження компонента в режимі редагування. Якщо значення властивості false, то для того, щоб у комірці з'явився курсор, треба почати набирати текст, натиснути клавішу <F2> або натиснути мишею.

Отже, повернемося до реалізації методу Гауса вирішення системи лінійних рівнянь алгебри. Для введення коефіцієнтів розширеної матриці системи скористаємося компонентом TStringGrid. Створіть новий проект і спроектуйте вид програми так, як показано на малюнку 6.60.

Встановіть такі властивості компонента TStringGrid: ColCount = 1; RowCount = 1; FixedCols = 0; FixedRows = 0;

```
goEditing = true; goTab = true;
goAlwaysShowEditor = true;
```



Мал. 6.60. Форма застосування

Код програми:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   LResources,   Forms,
             Controls,Graphics, Dialogs, StdCtrls,
  Buttons, Grids, LCLType, LCLProc;
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label4: TLabel;
    Memo1: TMemo;
    StringGrid1: TStringGrid;
```

```
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure FormShow(Sender: TObject);
    procedure Gauss(var vector: array of extended;
                    var b: array of extended;
                    var x: array of extended;
                    var n: integer);
    procedure StringGrid1KeyDown(Sender: TObject;
                                 var Key: Word;
      Shift: TShiftState);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
  n: integer;
  a: array of array of extended; {матриця коефіцієнтів
системи, двовимірний динамічний масив}
  vector: array of extended; {перетворений одновимірний
 динамічний масив}
  b: array of extended;
  x: array of extended;
implementation
{TForm1}
procedure TForm1.Gauss(var vector: array of extended;
  var b: array of extended; var x: array of extended;
  var n: integer);
var
  a: array of array of extended; {матриця коефіцієнтів
системи, двовимірний динамічний масив}
  i, j, k, p, r: integer;
  m, s, t: real;
begin
  try
    SetLength(a, n, n); {установка фактичного
    розміру масиву Перетворення одновимірного
    масиву на двовимірний}
```

```
k:= 1;
for i:= 0 to n - 1 do
for j:= 0 to n - 1 do
begin
  a [i, j]: = vector
  [k]; k: = k + 1;
end;
for k:= 0 to n - 2 do
begin
  for i:= k + 1 to n - 1 do
  begin
    if (a [k, k] = 0)
    then begin
      {перестановка
      рівнянь} p: = k;
      for r:= i n - 1 do
      begin
        if abs(a[r, k]) > abs(a[p, k]) then p:= r;
      end;
      if p <> k then
      begin
        for j:= k n - 1 do begin
          t: = a [k, j];
          a [k, j]: = a [p,
          j]; a [p, j]: = t;
        end;
        t: = b [k];
        b[k]:= b[p];
        b [p]: = t;
      end;
    end; // кінець блоку перестановки рівнянь
    m: = a [i, k] / a [k,
    k]; a [i, k]: = 0;
    for j:= k + 1 to n - 1 do
    begin
      a [i, j]: = a [i, j] - m * a [k,
    j]; end;
    b [i]: = b [i] - m * b
  [k]; end;
end;
{Перевірка існування рішення}
if a[n - 1, n - 1] <> 0 then
```

```
  begin
    x [n - 1]: = b [n - 1] / a [n - 1, n -
    1]; for i:= n - 2 downto 0 do
    begin
      s: = 0;
      for j:= i + 1 to n - 1 do
      begin
        s: = s - a [i, j] * x [j];
      end;
      x[i]:=(b[i] + s)/a[i, i];
    end;
    Memo1.Lines.Add('Рішення:');
    for i:= 0 to n - 1 do
      Memo1.Lines.Add('x' + IntToStr(i + 1) +
                      '=' + FloatToStr(x[i]));
    end
    else
    if b[n - 1] = 0 then
       Memo1.Lines.Add('Система рівнянь' +
                       'не має рішення.')
    else
       Memo1.Lines.Add('Система рівнянь' +
                'має безліч рішень.');
except
  on EInvalidOP do
  Memo1.Lines.Add('Неправильні дані. Система рівнянь' +
                  'не має рішення.');
  on EMathError do
  Memo1.Lines.Add('Неправильні дані. Система рівнянь'
                  + 'не має рішення.');
  on EZeroDivide do
  Memo1.Lines.Add('Неправильні дані. Система рівнянь'
                  + 'не має рішення.');
  on EOverflow do
  Memo1.Lines.Add('Неправильні дані. Система рівнянь'
                  + 'не має рішення.');
  on EUnderflow do
  Memo1.Lines.Add('Неправильні дані. Система рівнянь'
                  + 'не має рішення.');
  on EAccessViolation do
  Memo1.Lines.Add('Неправильні дані. Система рівнянь'
                  + 'не має рішення.');
```

```
  end;
    a:= nil; // звільнення пам'яті
end;
procedure TForm1.StringGrid1KeyDown(Sender: TObject;
                  var Key: Word; Shift:TShiftState);
begin
  if (Key = VK_Return) then
  begin
    if (StringGrid1.Col >= n)   then
    begin
      StringGrid1.Row:= StringGrid1.Row + 1;
      StringGrid1.Col:= 0;
      Key:= 0;
    end;
   end;
end;
procedure TForm1.FormShow(Sender: TObject);
begin
  StringGrid1.ColCount:= 1;
  StringGrid1.RowCount:= 1;
  Edit1.SetFocus;
  Button3.Visible:= true;
  Button4.Visible:= false;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Clear;
  StringGrid1.Clean;StringGr
  id1.ColCount:= 1;
  StringGrid1.RowCount:= 1;
  Edit1.Clear;
  Edit1.SetFocus;
  Button3.Visible:= true;
  Button4.Visible:= false;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  {звільнення пам'яті,
 розподіленої для динамічних масивів a:= nil;
  vector:= nil;
  x:= nil;
  b:= nil;
```

```
  Close();
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Edit1.SetFocus;
  if Length(Edit1.Text) = 0 // якщо порожній рядок
  then exit;
  n:= StrToInt(Edit1.Text);
  StringGrid1.ColCount: = n + 1;
  StringGrid1.RowCount:= n;
  Stringgrid1.SetFocus;
  Button3.Visible:= false;
  Button4.Visible:= true;
end;
procedure TForm1.Button4Click(Sender: TObject);
begin
  if (StringGrid1.Col >= n)   then
  begin
    StringGrid1.Row:= StringGrid1.Row + 1;
    StringGrid1.Col:= 0;
  end
  else
    StringGrid1.Col:= StringGrid1.Col+1;
end;
procedure TForm1.Button5Click(Sender: TObject);
var i, j, k, code:integer;
begin
  {Установка реальних розмірів динамічних
  масивів} SetLength(a, n, n);
  SetLength(vector, n * n);
  SetLength(b, n);
  SetLength(x, n);
  for j:= 0 to n - 1 do
  for i:= 0 to n - 1 do
  begin
    val(StringGrid1.Cells[i, j], a[i, j], code);
    if code <> 0 then
    begin
      ShowMessage('Помилка при введенні коефіцієнтів
      матриці'); StringGrid1.SetFocus;
      exit;
    end;
  end;
```

```
  for j:= 0 to n - 1 do
  begin
    val(StringGrid1.Cells[n, j], b[j], code);
    if code <> 0 then
    begin
      ShowMessage('Помилка при введенні вільних членів');
      StringGrid1.SetFocus;
      exit;
    end;
  end;
  code:=0;
  {Перетворення двовимірного масиву на
  одновимірний} k:= 1;
  for j:= 0 to n - 1 do
  for i:= 0 to n - 1 do
  begin
    vector[k]:= a[i, j];
    k: = k + 1;
  end;
  {Виклик процедури розв'язання системи
  лінійних рівнянь алгебри методом Гауса}
  Gauss(vector, b, x, n);
end;
procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
if Key = #13 then
  begin
    if Length(Edit1.Text) = 0 // якщо порожній рядок
    then exit;
    n:=StrToInt(Edit1.Text);
    StringGrid1.ColCount: = n + 1;
    StringGrid1.RowCount:= n;
    Stringgrid1.SetFocus;
    exit;
  end;
  {дозволяємо лише цифри, знак мінус та кл.
  BackSpace} if not (Key in ['0'.. '9', #
  8])
  then
  begin
    Key:= #0;
    exit;
  end;
```

```
end;
initialization
  {$I Unit1.lrs}
end.
```

У програмі користувач може вводити коефіцієнти і редагувати їх у TStringGrid. Після натискання кнопки "Рішити" дані з TStringGrid перетворюються на числове подання і записуються в динамічні масиви a – матриця коефіцієнтів системи, b – вектор вільних членів. Далі здійснюється виклик процедури Gauss() вирішення системи лінійних рівнянь алгебри методом Гаусса, яка практично не відрізняється від консольного варіанту. Якщо рішення системи існує, одержаний вектор x[x1, x2, . . . , xn] перетворюється на рядок і виводиться на екран.

Для контролю за введенням даних у Edit1 ми використовували метод, застосований нами в 6.3.7.1. Для контролю введення StringGrid1 функцію Val(), а при обчисленнях застосували механізм винятків.

6.3.10.3. Компоненти вибору

У цих компонентах можна організувати вибір елементів зі списку. Весь перелік міститься у властивості Items і має тип TString. Елементами списку є рядки. Рядки нумеруються, починаючи з нуля. Доступ до рядка здійснюється за допомогою вказівки індексу елемента властивості Items, наприклад Items[k] або властивості Items.Strings, наприклад Items.Strings[k].

Компонент TListBox

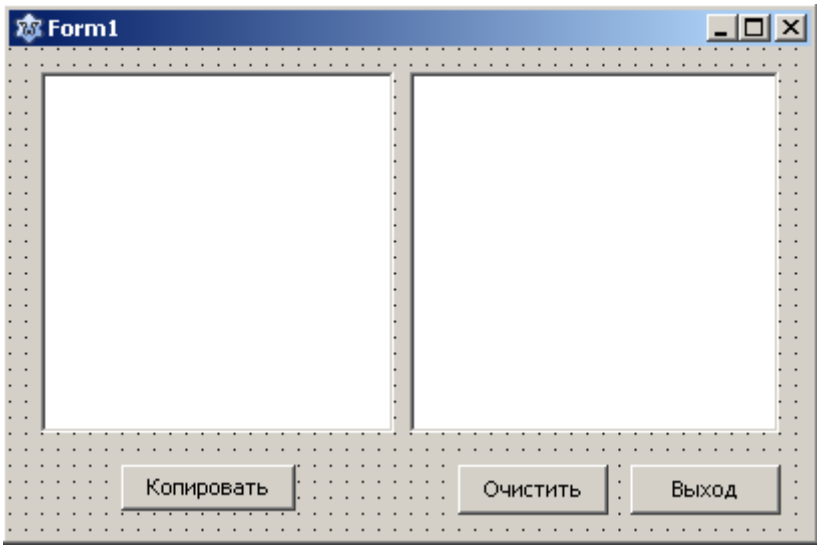
Розглянемо компонент `TListBox`. Він розташований на сторінці `Standard`. `Основні властивості компонента:`

- `MultiSelect – ознака множинного вибору. Якщо MultiSelect = true, дозволяється вибір одночасно кількох елементів.`

- ExtendedSelect – якщо ExtendedSelect= true і MultiSelect= true, вибір кількох елементів можна проводити стандартним способом, тобто. при натиснутій клавіші Shift можна вибрати кілька елементів, розташованих поспіль, а при натиснутій клавіші Ctrl вибрати елементи у довільному порядку.
- Count – загальна кількість елементів у компоненті.
- ItemIndex – індекс обраного елемента (при MultiSelect = false). Якщо вибрано кілька елементів (MultiSelect= true), то містить індекс елемента, на якому встановлено фокус.
- Selected[i] – якщо вибрано елемент з індексом i, то значення цього властивості дорівнює true.
- Sorted – якщо true, то елементи компонента автоматично сортуються.

Розглянемо кілька прикладів, щоб освоїти найпростішу техніку роботи з цим компонентом. Помістіть на форму два компоненти TListBox і три кнопки, оскільки показано на рис. 6.61.

Заповнимо програмно ListBox1 вмістом текстового файлу, в якому містяться прізвища, допустимо студентів. Потім просто поміщатимемо вибрані елементи в ListBox2. Спочатку реалізуємо вибір одиночного елемента.



Мал. 6.61. Форма застосування

Ось код цієї програми:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls;
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    ListBox1: TListBox;
    ListBox2: TListBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
```

```
procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{TForm1}
procedure TForm1.Button2Click(Sender: TObject);
begin
  with ListBox1 do
  begin
    if ItemIndex >= 0 then
      ListBox2.Items.Add(Items[ItemIndex]);
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox2.Clear;
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  tfile: TStringList;
```

```
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('List.txt');
  str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
  tfile.Free;
 end;
initialization
  {$I unit1.lrs}
end.
```

Якщо встановити ListBox2 властивість Sorted = true, то елементи будуть виведені у відсортованому вигляді. При додаванні нового елемента він буде поміщений у потрібне місце автоматично.

У цій реалізації, якщо при тому самому вибраному елементі натиснути кнопку " Копіювати " кілька разів, цей елемент потрапить у ListBox2 також кілька разів, тобто. рядки в ListBox2 виявляться не унікальними.

Щоб елементи в ListBox2 не повторювалися, напишемо функцію, яка перевіряє, чи не міститься вже цей елемент у ListBox2. Якщо такий елемент, функція повертає true, якщо ні, то false.

```
unit Unit1;
```

```
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls;
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    ListBox1: TListBox;
    ListBox2: TListBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject); функція
    Search(str: string): boolean;
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{TForm1}
function TForm1.Search(str: string): boolean;
var
```

```
  i: integer;
begin
  Result: = false;
  for i:= 0 to ListBox2.Count - 1 do
  if ListBox2.Items[i] = str then
  begin
    Result: = true;
    exit;
  end
  else Result:= false;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  with ListBox1 do
  begin
    if (ItemIndex >= 0) and (не Search(GetSelectedText))
    then ListBox2.Items.Add(Items[ItemIndex]);
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox2.Clear;
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
```

```
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('List.txt');
  str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
  tfile.Free;
end;
initialization
  {$I unit1.lrs}
end.
```

Реалізуємо тепер множинний вибір. Обробник події Button2Click дещо зміниться, т.к. при множинному виборі для того, щоб визначити, чи вибрано цей елемент, необхідно використовувати властивість Selected.

```
  unit Unit1;
  {$mode objfpc}{$H+}
  interface
  uses
    Classes, SysUtils, FileUtil, LResources, Forms,
    Controls, Graphics, Dialogs, StdCtrls;
```

```
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    ListBox1: TListBox;
    ListBox2: TListBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    функція Search(str: string): boolean;
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{TForm1}
function TForm1.Search(str: string): boolean;
var
  i: integer;
begin
  Result: = false;
  for i:= 0 to ListBox2.Count - 1 do
  if ListBox2.Items[i] = str then
```

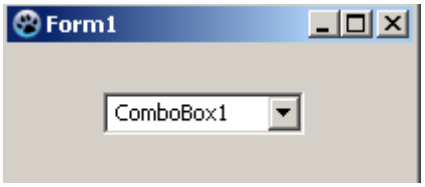
```
  begin
    Result: = true;
    exit;
  end
  else Result:= false;
end;
procedure TForm1.Button2Click(Sender: TObject);
var
  i: integer;
begin
  with ListBox1 do
  for i:= 0 to Items.Count - 1 do
  begin
    if (Selected[i]) and (не Search(Items[i]))
    then
      ListBox2.Items.Add(Items[i]);
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox2.Clear;
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  tfile: TStringList;
```

```
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('List.txt')
  ; str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str);// перетворення на кодування
    UTF-8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
  tfile.Free;ListBox1.MultiSelect
  := true; ListBox2.Sorted:=
  true;
end;
initialization
  {$I unit1.lrs}
end.
```

У компоненті можна відображати не лише рядки, а й зображення, наприклад, піктограми. Однак ми в цій книзі ці питання не розглядатимемо.

Компонент TComboBox

Цей компонент знаходиться на сторінці Standard. Зовнішній вигляд компонента показано на рис. 6.62.



Мал. 6.62. Компонент TComboBox

При натисканні на кнопку з трикутником з'явиться список, що розкривається (кажуть ще випадаючий). Методи роботи з таким списком практично не відрізняються від TListBox. Порівняно з TListBox він має перевагу, що полягає в тому, що можна редагувати елементи списку, а також додавати до списку нові елементи. Крім того, компонент дозволяє заощадити простір, коли на формі розташовано багато компонентів. Власне для цього він і використовується. До недоліків можна віднести те, що в ньому не можна вибрати одночасно кілька елементів. Основні властивості компонента:

- Items містить елементи списку. Доступ до окремих елементів можливий за індексом, наприклад, Items[k] містить елемент з номером (індексом) k (нумерація починається з нуля).
- Style – стиль відображення даних у компоненті, має такі значення:

1. csDropDown – список, що розкривається, з вікном редагування, що дозволяє користувачеві редагувати або додавати нові рядки до списку;
2. csDropDownList – список, що розкривається. У цьому режимі можна вибирати лише існуючі елементи списку. Редагувати або додавати рядки в цьому режимі не можна;
3. csSimple – список завжди розкритий, по суті збігається з ListBox, але є можливість редагування та додавання нових рядків;
4. csOwnerDrawFixed – список, що розкривається, з рядками однакової висоти, в яких можуть відображатися зображення та текст;
5. csOwnerDrawVariable – список, що розкривається, з

рядками різного

висоти, в яких можуть відображатись зображення та текст.

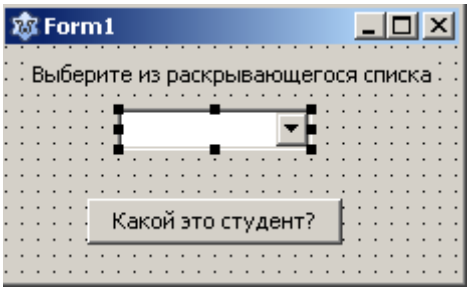
• Text – містить вибраний елемент списку або знову введений рядок;• ItemIndex – містить індекс вибраного елемента. За замовчуванням у момент проектування ItemIndex = -1. Якщо ви запустите програму з таким значенням ItemIndex, то у вікні ComboBox нічого не буде відображено. Особливо якщо ви використовуєте властивість Style = csSimple, та ще й забудете збільшити висоту Height компонента. У цьому випадку користувачеві буде виведено маленьке порожнє віконце без кнопки відкриття! Тому бажано або під час проектування, або програмно, наприклад, при створенні форми (OnCreate) встановлюватизначення ItemIndex. Найчастіше встановлюють ItemIndex = 0, у цьому випадку у вікно ComboBox буде виведено перший елемент списку. Але можна призначити ItemIndex та інше значення (природно всередині допустимого діапазону індексів елементів списку). Наприклад, елемента, який потрібно вибрати за замовчуванням. Під час виконання програми значення ItemIndex може набувати значення -1. Це відбувається у разі, якщо у вікні компонента проводилося редагування поточного елемента, тобто. за значенням ItemIndex = -1 можна дізнатися, що редагування проводилося і вжити відповідних дій, див. приклад нижче;

• DropDownCount – задає кількість елементів списку, що виводяться без смуги прокручування.

• DroppedDown – вказує стан компонента. При DroppedDown = true, перелік розкритий.

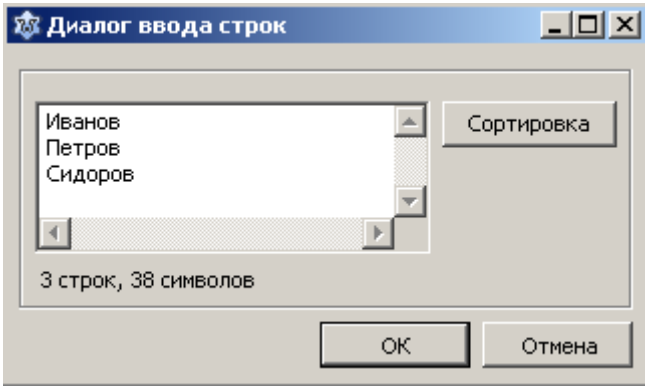
Розглянемо кілька прикладів.

Створіть новий проект, помістіть форму компоненти TComboBox, TLabel і TButton, як показано на рис. 6.63.



Мал. 6.63. Форма програми

Встановіть властивість Style компонента ComboBox1 рівним csDropDownList. У властивості Items введіть будь-які три прізвища, наприклад, як на малюнку 6.64. Нехай це будуть прізвища деяких студентів.



Мал. 6.64. Діалог введення рядків у компонент

В обробник події OnClick введіть наступний код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with ComboBox1 do
  case ItemIndex of
  0: ShowMessage('Це відмінник');
  1: ShowMessage('Це двієчник');
  2: ShowMessage('А цей
  середняк'); else
```

```
    ShowMessage('Ніхто не
  обраний'); end;
end;
```

Під час запуску програми у вікні ComboBox1 не буде відображено прізвище студента. Користувач має розкрити список. Як ми вже казали, це зовсім не зручно. Зробимо так, щоб під час запуску програми відображалося перше прізвище. З іншого боку, нехай повідомлення виводиться відразу після вибору потрібного елемента у списку ComboBox1, тобто. обійдемося без кнопки. Для цього використовуємо подію OnSelect.

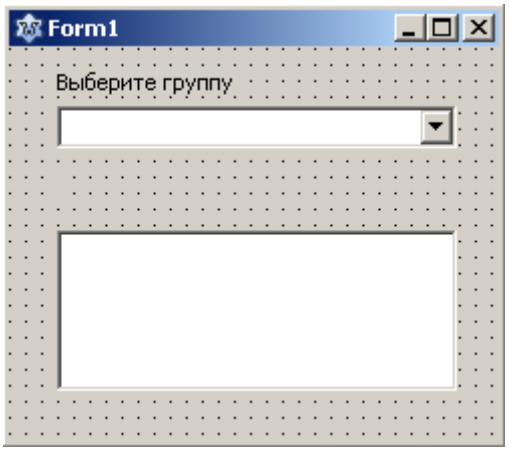
І ще, у багатьох випадках у вікно ComboBox1 виводиться значення, яке має бути прийняте як значення за промовчанням. Якщо користувача це значення влаштовує, то зазвичай йому досить просто натиснути клавішу Enter. Реалізуємо таку функціональність нашої програми. Отже, код програми (не забудьте видалити з форми кнопку):

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls,
  LCLType;
type
  {TForm1}
  TForm1 = class(TForm)
    ComboBox1: TComboBox;
    Label1: TLabel;
    procedure ComboBox1KeyPress(Sender: TObject;
                                var Key: char);
    procedure ComboBox1Select(Sender: TObject);
```

```
procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{TForm1}
procedure TForm1.ComboBox1Select(Sender: TObject);
begin
  with ComboBox1 do
  case ItemIndex of
  0: ShowMessage('Це відмінник');
  1: ShowMessage('Це двієчник'); 2:
  ShowMessage('А цей середняк');
  else
    ShowMessage('Ніхто не обраний');
  end;
end;
procedure TForm1.ComboBox1KeyPress(Sender: TObject;
                                   var Key: char);
begin
  if Key = #13 then ComboBox1Select(Sender);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  ComboBox1.ItemIndex:= 0;
```

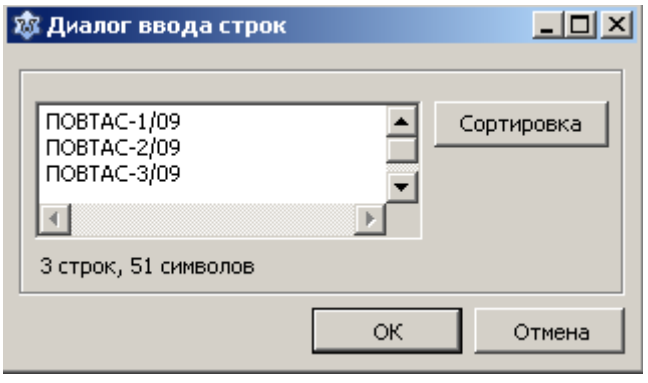
```
end;
initialization
  {$I unit1.lrs}
end.
```

Зробимо більш корисну програму. Припустимо, у деяких текстових файлах містяться списки кількох груп студентів. Нехай для певності груп буде три. Залежно від вибору користувача необхідно виводити список студентів вибраної групи. Список групи виводитимемо в TListBox. У новому проекті сформуйте вигляд вікна вашої програми, рис. 6.65.



Мал. 6.65. Форма застосування

У властивості Items введіть будь-які три назви груп, наприклад, як на малюнку 6.66.



Мал. 6.66. Діалог введення рядків у компонент

Створіть три текстові файли з іменами List1.txt, List2.txt та List3.txt з прізвищами студентів. Заповнювати TListBox вмістом файлу ми вже вміємо, для простоти припустимо, що файли знаходяться в тій же папці, що й наша програма. Код програми:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls,
  LCLType;
type
  {TForm1}
  TForm1 = class(TForm)
    ComboBox1: TComboBox;
    Label1: TLabel;
    Label2: TLabel;
    ListBox1: TListBox;
    procedure ComboBox1KeyPress(Sender: TObject; var Key:
char);
    procedure ComboBox1Select(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure LoadListGroup(namefile: string);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
```

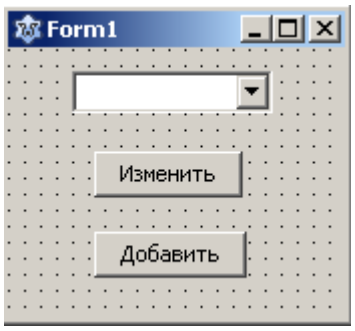
```
  Form1: TForm1;
implementation
{TForm1}

procedure TForm1.LoadListGroup(namefile: string);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile(namefile);
  str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
  tfile.Free;ListBox1.Sorted
  := true;
end;
procedure TForm1.ComboBox1Select(Sender: TObject);
begin
  with ComboBox1 do
  begin
    Label2.Caption:='Список групи' + Text;
    case ItemIndex of
    0: LoadListGroup('List1.txt');
```

```
    1: LoadListGroup('List2.txt');
    2: LoadListGroup('List3.txt');
    else
      ShowMessage('Група не
    обрана'); end;
  end;
end;
procedure TForm1.ComboBox1KeyPress(Sender: TObject;
                                   var Key: char);
begin
  if Key = #13 then
  begin
    ComboBox1Select(Sender);
    Key:=#0;
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  ComboBox1.ItemIndex:= 0;
  ComboBox1.Style:= csDropDownList;
end;
initialization
  {$I unit1.lrs}
end.
```

Завантаження в TListBox вмісту текстового файлу ми оформили у вигляді процедури, в яку як параметр передається ім'я файлу, що відповідає вибраній групі.

На закінчення розглянемо приклад, як у TComboBox реалізуються корекції.

тирування та додавання елементів до списку. Створіть форму, рис. 6.67.



Мал. 6.67. Форма застосування

Тепер уже встановіть властивість Style компонента ComboBox1 рівним csDropDown. У властивості Items введіть будь-які дані. У обробнику OnSelect у змінній oldItemIndex ми запам'ятовуємо індекс обраного елемента. Якщо натиснуто кнопку "Змінити", то оператор

```
ComboBox1.Items[oldItemIndex]:= ComboBox1.Text;
```

змінить вибраний елемент новим вмістом. Якщо користувач натисне кнопку "Додати", оператор

```
ComboBox1.Items.Add(ComboBox1.Text);
```

додасть новий запис до списку.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, StdCtrls;
```

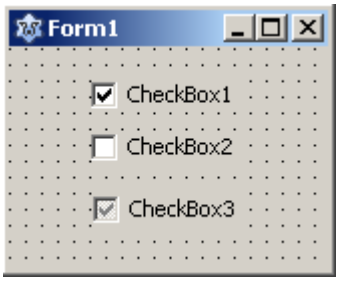
```
type
  {TForm1}
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    ComboBox1: TComboBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure ComboBox1Select(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
  oldItemIndex: integer;
implementation
{TForm1}
procedure TForm1.ComboBox1Select(Sender: TObject);
begin
  oldItemIndex:= ComboBox1.ItemIndex;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  ComboBox1.ItemIndex:= 0;
  ComboBox1.Style:= csDropDown;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  якщо ComboBox1.ItemIndex = - 1 then
  ComboBox1.Items[oldItemIndex]:= ComboBox1.Text;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  якщо ComboBox1.ItemIndex = - 1 then
  ComboBox1.Items.Add(ComboBox1.Text);
end;
initialization
  {$I unit1.lrs}
end.
```

Компоненти вибору – перемикачі

Ця група компонентів реалізує такі елементи графічного інтерфейсу, як прапорці та перемикачі. Як було сказано у 6.1. за допомогою таких елементів можна здійснювати вибір та встановлення будь-яких опцій, режимів, станів тощо. Таким чином, у цих компонентах, на відміну від таких компонентів як TListBox та TComboBox, проводиться вибір не даних як таких, а якихось властивостей та характеристик об'єктів, що використовуються у додатку. Вибір користувача в цих компонентах передбачає відповідь користувача типу "Так", "Ні". Розглянемо компонент TCheckBox.

Цей компонент реалізує елемент графічного інтерфейсу прапорець. Також застосовується термін індикатор. Багато програмістів говорять також просто "чек бокс". Зазвичай у діалоговому вікні є кілька прапорців. Стан кожного прапорця залежить від інших прапорців, тому часто використовується термін група незалежних перемикачів, рис. 6.68.



Мал. 6.68. Вигляд компонента TCheckBox

Основні властивості компонента:

- State – компонент TCheckBox може бути в загальному випадку в трьох станах:

1. Прапорець обраний, у віконці компонента стоїть галочка. Цьому стану відповідає значення State=cbChecked;
2. Прапорець не вибраний, у віконці компонента галочка відсутня. Цьому зі-стоянню відповідає значення State=cbUnchecked;
3. Прапорець обраний, у віконці компонента стоїть галочка. Але саме віконце та галочка сірого кольору. Цьому стану відповідає значення State=cbGrayed; Це так зване проміжний стан. Найчастіше застосовується у тих випадках, коли той чи інший режим, та чи інша опція має бути обов'язково обрана (задіяна), але так, щоб користувач був проінформований про це. Найчастіше цей стан використовується у комбінації з властивістю Enabled=false, тобто. ця опція автоматично встановлена, користувач про це знає, але міняти цю опцію не можна.

- AllowGrayed – дозволяє або не дозволяє застосування стану State = cbGrayed. Якщо AllowGrayed= false (значення за промовчанням), можливі лише два стани, прапорець обраний і прапорець не выбран;

- Checked – якщо Checked = true, прапорець вибраний. Значення State = cbChecked. Якщо ж Checked = false, то State дорівнює або cbUn-checked, або cbGrayed;

- `Caption - задає текст, пов'язаний з прапорцем;`

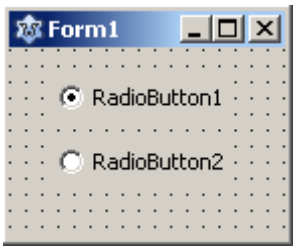
Щоб визначити обраний прапорець чи ні достатньо записати оператор

```
if CheckBox1.Checked then <дії, якщо прапорець
обрано> else <дії, якщо прапорець не обраний>;
```

або

```
case CheckBox1.State of
  cbChecked: <відповідні дії>; cbUnchecked:
  <відповідні дії>; cbGrayed: <відповідні
  дії>;
end;
```

Розглянемо компонент, що реалізує перемикач TRadioButton, рис. 6.69. Говорять також радіокнопка або залежний перемикач.



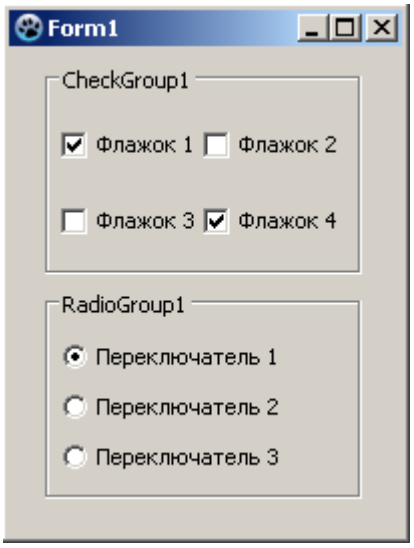
Мал. 6.69. Вигляд компонента TRadioButton

Слід зауважити, що використання одного перемикача немає сенсу, оскільки механізм перемикачів призначений для організації вибору користувачем лише одного з кількох можливих режимів (опцій, властивостей тощо). Тому необхідне використання щонайменше двох компонентів TRadioButton. Властивості компонента:

- Checked – визначає увімкнений або не увімкнений перемикач. Якщо перемикач включений, він відзначається точкою всередині кружечка, а Checked= true;
- `Caption – задає текст напису поряд із кнопкою.`

Зазвичай компоненти TRadioButton розміщуються у спеціальному компоненті-контейнері (про це трохи пізніше), але можна покласти їх і на форму. Як тільки ви розмістите кілька компонентів TRadioButton на форму, вони починають працювати як єдина група перемикачів. Тобто. навіть на етапі проектування ви не зможете встановити Checked = true одночасно кільком радіокнопкам, а можете встановити цю властивість рівним true тільки в одного перемикача!

Для зручної роботи з групами компонентів TCheckBox та TRadioButton існують спеціальні контейнери TCheckGroup та TRadioGroup. Контейнер це такий компонент, де можна розміщувати інші компоненти. Інакше їх ще називають панелями, рис. 6.70.



Мал. 6.70. Компоненти-контейнери TCheckGroup та TRadioGroup

Загальні властивості контейнерів TCheckGroup та TRadioGroup:

- `Caption – текст заголовка панелі;`

• Columns - задає кількість стовпців. На рис. 6.70. панель CheckGroup1

містить два стовпці, а панель RadioGroup1 складається з одного стовпця;

• ColumnLayout – задає спосіб розміщення дочірніх компонентів у стовпцях. Має значення: clHorizontalThenVertical – компоненти розміщуються горизонталлю, clVerticalThenHorizontal – компоненти розміщуються по вертикалі. На малюнку прапорці розміщені по горизонталі;

• Items – містить список рядків типу TStrings із текстами заголовків відповідних компонентів. Рядки можна формувати як при проектуванні, так і програмно. При проектуванні необхідно розкрити редактор рядків Items та ввести необхідні рядки. Скільки було введено рядків, стільки і буде створено перемикачі або прапорці. Доступ до тексту прапорця або перемикача здійснюється за індексом Items[k], індекси зазвичай починаються з нуля.

Визначення того, який елемент вибрав у компонентах TCheckGroup і TRadioGroup різняться. Так, у TCheckGroup використовується властивість Checked[k], де k індекс прапорця. Наприклад:

```
if CheckGroup1.Checked[k]
then ShowMessage('Вибраний прапорець' + IntToStr(k + 1));
```

У TRadioGroup для цих цілей можна використовувати властивість ItemIndex, яка вказує індекс обраного перемикача, наприклад:

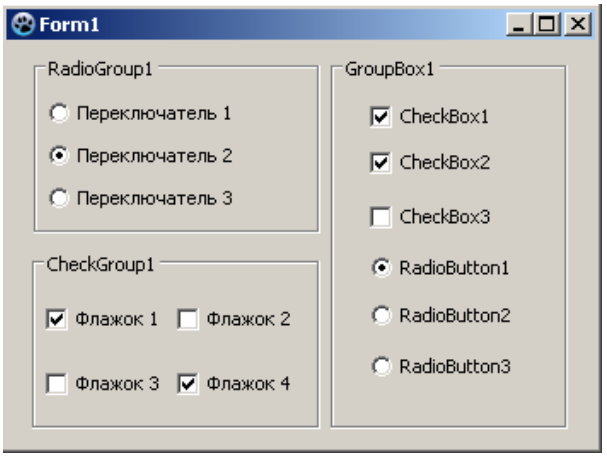
```
with RadioGroup1 do
begin
  k:= ItemIndex + 1;
  case ItemIndex of
  0: ShowMessage('Вибраний перемикач' + IntToStr(k));
  1: ShowMessage('Вибраний перемикач' + IntToStr(k));
```

```
  2: ShowMessage('Вибраний перемикач' + IntToStr(k));
  else
    ShowMessage('Не вибраний жоден перемикач');
  end;
end;
```

За замовчуванням ItemIndex=-1. Це означає, що жодного перемикача не вибрано. Щоб програмно або при проектуванні вказати вибраний перемикач, достатньо ItemIndex присвоїти значення з допустимого діапазону.

У розглянутих нами контейнерах TCheckGroup, TRadioGroup розташування перемикачів та прапорців формується автоматично самим контейнером. Іноді буває зручніше застосувати інший контейнер, а саме TGroupBox, де можна раціональніше використовувати площу контейнера. Крім того, у TGroupBox можна розміщувати будь-які компоненти. Різниця від розглянутих раніше спеціальних контейнерів TCheckGroup, TRadioGroup у тому, що необхідно використовувати компоненти TCheckBox і TRadioButton.

Зовнішній вигляд розглянутих компонентів наведено на рис. 6.71.



Мал. 6.71. Компонент-контейнер TGroupBox

6.3.10.4. Компоненти відображення структурованих даних

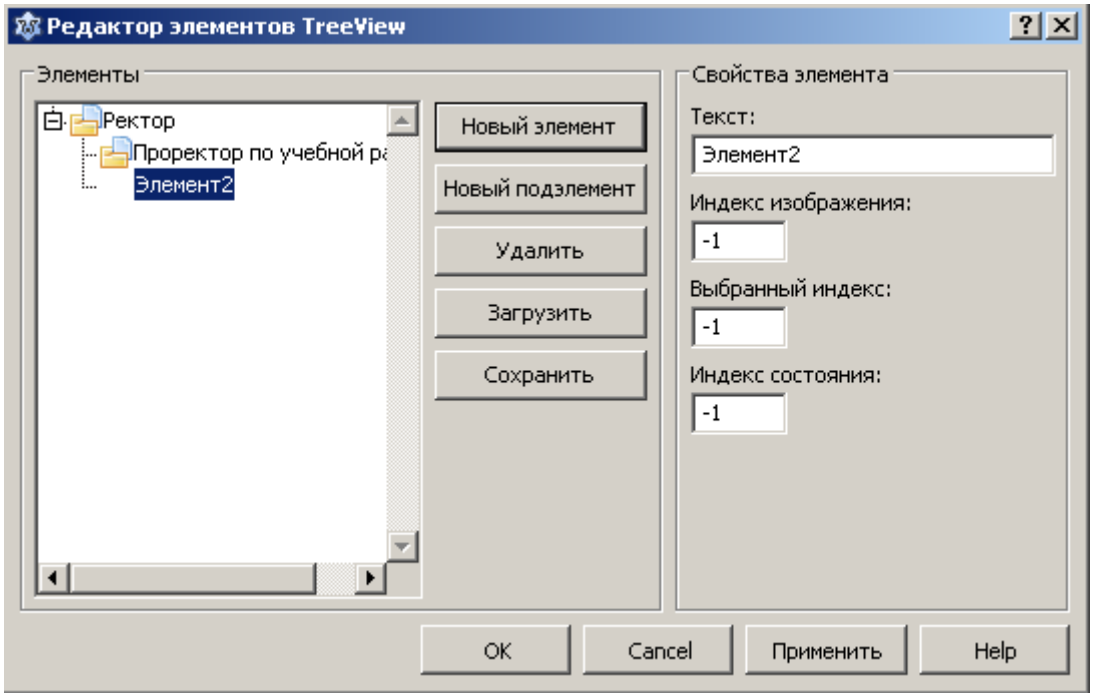
Компонент TTreeView

Цей компонент (вкладка Common Controls) застосовується для відображення даних, що мають ієрархічну структуру або, інакше кажучи, даних, які можна подати у вигляді кількох рівнів. До таких даних, наприклад, належать структура якогось підприємства, генеалогічне дерево людини, файлова структура диска тощо.

У компоненті TTreeView інформація відображається у вигляді зв'язаних вузлів. Кожен вузол складається з деякого тексту, що власне складають дані. Може також містити деяку піктограму і, крім того, з кожним вузлом може бути пов'язаний певний об'єкт. Вузол може мати подузли, тобто. вузли, що лежать на наступному, нижчому рівні. Також будь-який вузол, крім вузла верхнього рівня, може входити в інший вузол як підвузл. Інформація про вузли дерева міститься у властивості Items. Інформацію до компонента можна заносити як вручну, так і програмно.

Для ручного заповнення існує спеціальний редактор елементів TReeView, відкрити який можна вибравши пункт Items в інспекторі об'єктів і натиснувши кнопку з трьома крапками або просто двічі клацнувши по самому компоненту на формі.

Щоб ввести новий елемент, натисніть кнопку "Новий елемент". Натисніть кнопку "Новий поделемент", щоб ввести для нововведеного вузла його підвузли. Щоб ввести вузол, розташований одному рівні з якимось вузлом, його необхідно попередньо виділити. Після цього натисніть кнопку "Новий елемент", рис. 6.72.



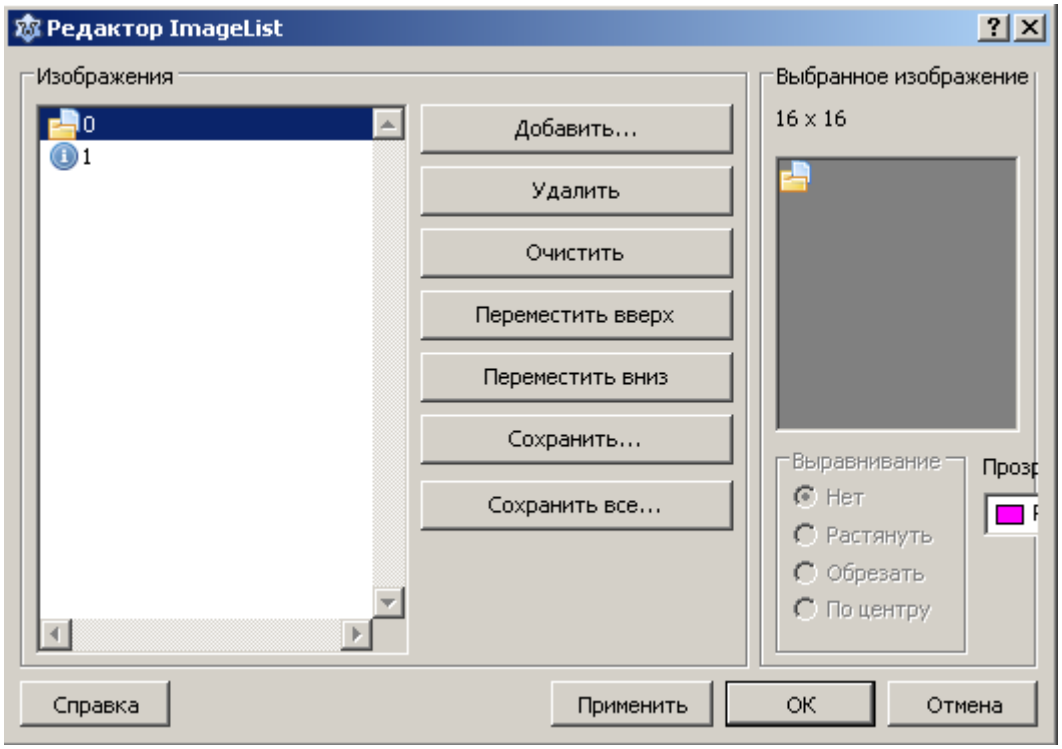
Мал. 6.72. Редактор елементів TTreeView

У полі "Текст" введіть змістовну інформацію для цього сайту.

Кнопка "Завантажити" дозволяє завантажити дані з текстового файлу. У файлі мають бути тексти вузлів. Нижчі рівні (підвузли) відзначаються відступом. Так, щойно введену інформацію, ми могли б завантажити з файлу, записи якого мали б вигляд:

```
Ректор
  Проректор з навчальної роботи
```

Тепер про індекси. Як видно з малюнка, поруч із текстом вузла знаходиться малюнок (піктограма). Малюнки розміщуються у спеціальному компоненті TImageList, розташованому на тій же вкладці Common Controls. Помістіть компонент TImageList у будь-якому місці форми, це невидимий компонент (природно під час виконання). У ньому також є спеціальний редактор для занесення зображень. Щоб відкрити його найпростіше двічі клацнути по самому компоненту, рис. 6.73.



Мал. 6.73. Редактор компоненту TImageList

Натисніть кнопку "Додати…". Відкриється стандартне діалогове вікно відкриття файлу зображення. Я вибрав два малюнки, що сподобалися мені, з папки, де встановлений Lazarus. Кожне зображення має свій індекс, за яким здійснюється доступ до нього. Індекси починаються з нуля. Основні властивості компонента TImageList – висота Height та ширина Width. Ці властивості дозволяють встановити розмір зображення. За замовчуванням розмір зображення 16x16 пікселів.

Щоб зв'язати компонент TImageList з TTreeView, виділіть компонент TTreeView, виберіть в інспекторі об'єктів властивість Images і в списку вкажіть потрібний компонент, наприклад ImageList1. Або програмно запишіть оператор

```
TreeView1.Images:= ImageList1;
```

Властивість `ImageIndex` дозволяє вказати індекс зображення в

TImageList, яке з'являтиметься поруч із поточним вузлом. Властивість SelectedIndex дозволяє вказати індекс зображення в TImageList, яке з'являтиметься поруч із поточним вузлом при його виборі (виділенні). І, нарешті, властивість StateIndex дозволяє додати додаткову піктограму, яка з'являтиметься поруч із поточним вузлом. Зображення, що відповідають StateIndex, зберігаються в іншому компоненті TImageList, а зв'язок здійснюється через властивість StateImages. Отже, якщо ви збираєтеся використовувати додаткові піктограми, ви повинні помістити на форму другий компонент TImageList та заповнити його відповідними зображеннями. За промовчанням значення всіх індексів рівні

-1, Мал. 6.72. Це означає, що піктограм відсутні.

Розглянемо деякі інші властивості компонента TTreeView:

- Images: TImageList – містить набір зображень, які будуть використовуватись при промальовуванні вузлів;
- Indent: integer – визначає відступ у пікселях від лівого кута вузла всім його подузлов;
- Items: TTreeNodes – містить список усіх вузлів. Доступ до будь-якого вузла здійснюється за індексом. Індексація починається із нуля;
- ReadOnly: boolean – забороняє/дозволяє редагування тексту написів вузлів;
- RightClickSelect: boolean - дозволяє вибір вузлів правою кнопкою миші;
- Selected: TTreeNode – містить список усіх вибраних вузлів. Якщо нічого не вибрано, то одно nil;
- SelectionCount: Cardinal – кількість вибраних вузлів;

- `Selections[Index: integer]: TTreeNode` – доступ до вибраних вузлів за індексом;
- `RowSelect: boolean` – дозволяє виділення кольором ліній вибраних

вузлів. Ігнорується, якщо ShowLines дорівнює false;

- ShowButtons: boolean – дозволяє/забороняє показ стандартних кнопок розкриття підвузлів. Типово true. Якщо false, вузол розкривається подвійним клацанням миші;
- ExpandSignType – визначає вид піктограми розкриття вузла. Може приймати значення tvestPlusMinus - значок у вигляді плюса/мінуса, tvestArrow - значок у вигляді стрілки, tvestTheme - вид значка визначається системними налаштуваннями.
- ShowLines: boolean – дозволяє/забороняє показ ліній;
- ShowRoot: boolean – дозволяє/забороняє показ ліній, що йдуть від найвищого рівня ієрархії. Ігнорується, якщо ShowLines = false;
- SortType: TSortType – вказує спосіб сортування вузлів: stNone – немає сортування, stData – сортування за пов'язаними з вузлами даними об'єктів, stText – сортування за текстом написів, stBoth – сортування за текстом та за даними;
- Topitem: TTreeNode – визначає кореневий вузол.

Методи компонента:

- procedure FullCollapse – згортає всі вузли, крім вузлів найвищого рівня;
- procedure FullExpand - розкриває всі вузли;
- function IsEditing: boolean – повертає true, якщо користувач редагує будь-який вузол;
- procedure LoadFromFile(const FileName: string) – завантажує дані компонент з файла;
- procedure SaveToFile(const FileName: string) – зберігає у файлі вміст компонента;

Розглянемо події, пов'язані з TTreeView:

- OnChange – виникає при зміні стану вибору вузла, наприклад, при виділенні одного з вузлів або при скасуванні виділення;
- OnExpanding – з'являється перед розкриттям вузла. В обробник події OnExpanding передається параметр var AllowExpansion: boolean, який можна встановити рівним false, якщо ви хочете заборонити розкриття цього вузла;
- OnExpanded – виникає після розкриття вузла;
- OnCollapsing – виникає перед згортанням вузла. Обробник OnCollapsing передає параметр var AllowCollapse: boolean, що дозволяє або забороняє згортання вузла;
- OnCollapsed – виникає після згортання вузла;
- OnCompare - виникає при порівнянні двох вузлів Node1 та Node2. У параметрі Compare обробник повинен повернути негативне число, якщо Node1<Node2, нуль, якщо Node1=Node2 і позитивне число, якщо Node1>Node2;
- OnDeletion - виникає при видаленні вузла;
- OnEdited – виникає після завершення редагування тексту написи у вузлі. У обробник передаються вузол Node та S – новий напис.

Для роботи з вузлами Items передбачено спеціальний клас TTreeNodes. Розглянемо деякі властивості та методи цього класу.

- Count – кількість вузлів у Items;
- Item [Index: integer] - вузол з індексом Index; Методи класу TTreeNodes:
- function Add(Node: TTreeNode; const S: String): TTree-Node – додає вузол у кінець того списку, у якому зареєстровано вузол Node. Якщо Node = nil додається кореневий вузол для всього компонента;

- `function AddChild(Node: TTreeNode; const S: string):`

TTreeNode – додає вузол до кінця списку item дочірніх вузлів вузла Node;• function AddChildFirst(Node: TTreeNode;

const S: String): TTreeNode – додає вузол на початок списку Item дочірніх вузлів вузла Node;

• function AddFirst(Node: TTreeNode;

const S: String): TTreeNode – додає вузол на початок списку, в якому зареєстрований вузол Node;

• procedure BeginUpdate – блокує оновлення екрана доти, доки не буде виконано метод EndUpdate. Використовується при великій кількості операцій з видалення та вставки вузлів, оскільки при цьому може виникати мерехтіння екрана;

• procedure Clear – очищає список усіх вузлів та підвузлів компонента;

• procedure Delete(Node: TTreeNode) – видаляє вузол Node;

• procedure EndUpdate – скасовує блокування оновлення екрану методом BeginUpdate;

• function GetFirstNode: TTreeNode – повертає перший вузол у списку Items[0];

• function GetNode(Itemid: HTreeItem): TTreeNode – повертає вузол за його ідентифікатором Itemid;

• function Insert(Node: TTreeNode;

const S: string): TTreeNode – вставляє вузол безпосередньо перед вузлом Node;

Для роботи безпосередньо з вузлом у класі TTreeNode має свій набір методів, властивостей та подій:

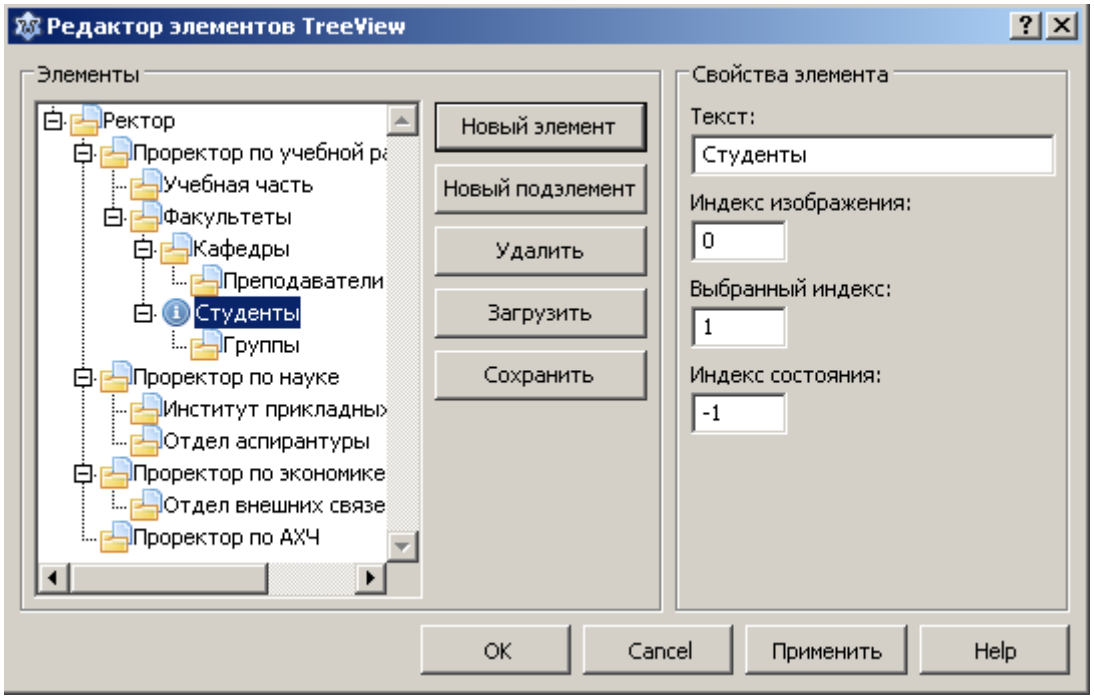
- `AbsoluteIndex: integer-` повертає абсолютний індекс вузла(з урахуванням усіх підвузлів);
- `Count: integer – містить кількість підвузлів у списку Item;`

- Cut: boolean - позначає вузол статусом "вирізаний";
- Expanded: boolean - дорівнює true, якщо вузол розкритий;
- Focused: boolean – дорівнює true, якщо вузлі встановлено фокус;
- HasChildren: boolean - true, якщо вузол має дочірні вузли;
- ImageIndex: TImageIndex – дорівнює індексу пов'язаної з вузлом піктограми;
- Index: Longint – містить індекс вузла у списку дочірніх вузлів його батьківського вузла;
- IsVisible: boolean – дорівнює true, якщо вузол видно;
- Level: integer – містить ієрархічний рівень вузла;
- Owner: TTreeNodes – містить посилання на власника цього сайту;
- Parent: TTreeNode – містить посилання батьківський вузол;
- Selected: boolean - дорівнює true, якщо вузол виділено;
- SelectedIndex: integer – номер піктограми виділеного вузла;
- Text: Strings – містить текст вузла;
- function AlphaSort: boolean - сортує вузли за алфавітом властивостей Text і повертає true у разі успіху;
- procedure Collapse(Recurse: boolean)-закриває всі вузли (Recurse = true) або лише розкриті (Recurse = false);
- procedure Delete – видаляє поточний вузол;
- procedure DeleteChildren – видаляє дочірні вузли;
- function EditText: boolean – переводить текст вузла в режим редагування;
- procedure EndEdit(Cancel: boolean) – закінчує редагування тексту та зберігає зміни, якщо Cancel= false;

- `procedure Expand(Recurse: boolean)` – відкриває вузол (і всі подузли, якщо `Recurse= true`);

- `function GetFirstChild: TTreeNode` – повертає посилання на перший подузол або nil, якщо немає підвузлів;
- `function GetLastChild: TTreeNode` - повертає посилання на останній подузол або nil, якщо немає підвузлів;
- `function GetNext: TTreeNode` – повертає посилання черговий подузел;
- `function HasAsParent(Value: TTreeNode): boolean` – повертає true, якщо Value – батьківський вузол;
- `function IndexOf(Value: TTreeNode): integer` – повертає індекс вузла Value.

Давайте тепер завершимо процес введення в TReeView1 структуру типового університету, рис. 6.74.



Мал. 6.74. Приклад введення елементів за допомогою редактора TTreeView

Покладіть на форму компонент TStatusBar і напишіть наступні обробники:

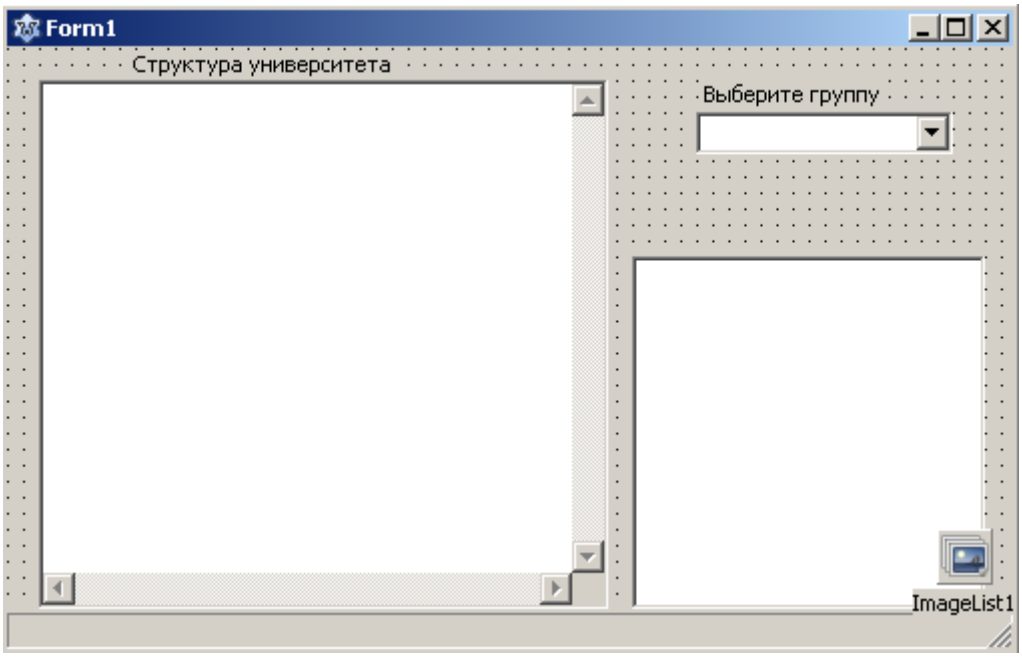
```
procedure TForm1.TreeView1Change(Sender: TObject;
                                 Node: TTreeNode);
begin
  if Node.Selected then
    StatusBar1.SimpleText:= 'Вибраний вузол:' +
Node.Text; end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  TreeView1.Images:= ImageList1;
  TreeView1.ExpandSignType:= tvestPlusMinus;
end;
```

У обробнику OnCreate форми ми зв'язали зображення, що містяться в ImageList1 з TreeView1, а також вказали на вигляд кнопки розкриття вузла. Обробник OnChange передається вузол Node, стан якого змінилося (наприклад, при клацанні користувача на вузол, він буде виділений). Дізнатися, чи виділено (вибрано) вузол чи ні, можна за властивістю Selected. Текст цього вузла міститься як Text.

Тепер створимо це дерево програмним шляхом. Крім виведення в TStatusBar інформації про обраний користувачем вузлі, виконаємо якісь дії. Наприклад, виведемо список груп студентів при натисканні вузла "Групи". Створіть новий проект, помістіть на форму компоненти TTreeView, TComboBox, TListBox, TStatusBar, TImageList та три компоненти TLabel, оскільки показано на малюнку 6.75.

Спочатку компоненти TComboBox, TListBox та два TLabel будуть невидимі. Тільки при виборі вузла "Групи" ми зробимо їх видимими, а при виборі іншого вузла знову зробимо їх невидимими.

Підготуйте три текстові файли зі списками студентів або візьміть файли з прикладу, де ми розглядали компонент TComboBox.



Мал. 6.75. Форма застосування

```
unit Unit1;
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, ComCtrls,
  StdCtrls;
type
  {TForm1}
  TForm1 = class(TForm)
    ComboBox1: TComboBox;
    ImageList1: TImageList;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    ListBox1: TListBox;
    StatusBar1: TStatusBar;
    TreeView1: TTreeView;
```

```
    procedure      ComboBox1KeyPress(Sender:      TObject;
                                     var Key: char);
    procedure ComboBox1Select(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure TreeView1Change(Sender: TObject;
                              Node: TTreeNode);
    procedure LoadListGroup(namefile: string);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{TForm1}
procedure TForm1.FormCreate(Sender: TObject);
var
  i: integer;
begin
  with TreeView1 do
  begin
    Images:= ImageList1;
    ExpandSignType:= tvestPlusMinus;
    Items.Clear;
    Items.Add(nil, 'Ректор');
    Items.AddChild(Items.Item[0],
                   'Проректор з навчальної роботи');
    Items.AddChild(Items.Item[1], 'Навчальна частина');
```

```
    Items.AddChild(Items.Item[1], 'Факультети');
    Items.AddChild(Items.Item[3], 'Кафедри');
    Items.AddChild(Items.Item[4], 'Викладачі');
    Items.AddChild(Items.Item[3], 'Студенти');
    Items.AddChild(Items.Item[6], 'Групи');
    Items.AddChild(Items.Item[0], 'Проректор з науки');
    Items.AddChild(Items.Item[8],
                  'Інститут прикладних досліджень');
    Items.AddChild(Items.Item[8], 'Відділ аспірантури');
    Items.AddChild(Items.Item[0], 'Проректор з економіки');
    Items.AddChild(Items.Item[11], 'Відділ зовнішніх
    зв'язків'); Items.AddChild(Items.Item[0], 'Проректор з
    АХЧ');
    for i:= 0 to Items.Count - 1 do
    begin
      Items.Item[i].ImageIndex:= 0;
      Items.Item[i].SelectedIndex:= 1;
    end;
    FullExpand;
  end;
  Label1.Caption:= 'Виберіть групу';
  Label2.Caption:='';
  Label3.Caption:= 'Структура
  університету'; Label1.Visible:=
  false; Label2.Visible:= false;
  ComboBox1.Visible:= false;
  ListBox1.Visible:= false;
end;
procedure TForm1.TreeView1Change(Sender: TObject; Node:
```

```
TTreeNode);
begin
  StatusBar1.SimpleText:= 'Вибраний:' +
  Node.Text; якщо Node.AbsoluteIndex = 7 then
  begin
    Label1.Visible: = true;
    ComboBox1.Visible:= true;
    ComboBox1.SetFocus;
    ComboBox1.ItemIndex:= 0;
  end
  else
  begin
    Label1.Visible:= false;
    Label2.Visible:= false;
    ComboBox1.Visible:= false;
    ListBox1.Visible:= false;
  end;
end;
procedure TForm1.ComboBox1Select(Sender: TObject);
begin
  with ComboBox1 do
  begin
    Label2.Visible: = true;
    Label2.Caption:='Список групи' + Text;
    ListBox1.Visible:= true;
    case ItemIndex of
    0: LoadListGroup('List1.txt');
    1: LoadListGroup('List2.txt');
    2: LoadListGroup('List3.txt');
```

```
    else
      ShowMessage('Група не
    обрана'); end;
  end;
end;
procedure TForm1.LoadListGroup(namefile: string);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile(namefile);
  str: = tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // Перетворення на кодування UTF-
    8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
  tfile.Free;ListBox1.Sorted
  := true;
end;
procedure TForm1.ComboBox1KeyPress(Sender: TObject; var
Key: char);
begin
  if Key = #13 then
  begin
    ComboBox1Select(Sender);
```

```
    Key:=#0;
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

Зверніть увагу вузли, що знаходяться на першому рівні, мають однакові індекси. Наприклад, у всіх проректорів Items.Item[0]. Оскільки наше дерево фіксоване, додавання нових вузлів та підвузлів у програмі не передбачено і, отже, індекс вузла "Групи" нам відомий, для визначення вибору вузла "Групи" ми скористалися властивістю AbsoluteIndex:

```
якщо Node.AbsoluteIndex = 7 then
```

Але можна було вчинити і по-іншому:

```
if Node.Text = 'Групи' then
```

Компонент TTreeView можна застосувати для відображення дерева папок дисків, наприклад, у лівому вікні Провідника Windows. Спробуємо і ми реалізувати це завдання. Скажу відразу, на панелі компонентів у вкладці "Misc" є спеціалізований компонент TShellTreeView, призначений саме для цього. Але ми спробуємо зробити це самі, так би мовити "ручками", адже "не боги горщики обпікають"!

Спочатку нам необхідно познайомитися з деякими функціями, які нам будуть потрібні для роботи.

Функції пошуку файлів

Для пошуку файлів зазвичай використовуються такі функції:

- FindFirst - визначається так:

```
function FindFirst (Const Path: String; Attr: Longint;
                    out Rslt : TSearchRec): Longint;
```

Параметр Path – задає шлях до каталогу, і навіть маску пошуку файлів. Маска може містити символи - шаблони:

? – будь-який одиночний символ;

* - скільки завгодно символів або навіть відсутність символів.

Наприклад, маска *.* означає будь-які файли з будь-яким розширенням, а маска *.pas означає будь-які файли з розширенням pas.

Параметр Attr – атрибути файлу. Визначає, які типи файлів потрібно шукати. Може приймати такі значення:

- faReadOnly – файли лише для читання;
- faHidden – приховані файли;
- faSysFile – системні файли;- faDirectory - папки та каталоги;- faArchive – архівні файли;
- faAnyFile – будь-які файли.

Можна застосовувати будь-які комбінації, наприклад, для пошуку прихованих і системних файлів можна задати faHidden + faSysFile.

Змінна Rslt представляє собою тип даних запис TSearchRec визначається таким чином (наводжу тільки ті поля, які представляють для нас інтерес):

```
Type
  TSearchRec = record
    Time: Longint;
    Size: Int64;
    Attr: Longint;
    Name: TFileName;
  end;
```

Тут Attr – атрибути файлу (див. вище), Time – час і дата створення або останнього оновлення файлу в системному форматі, Size – довжина файлу в байтах, Name – ім'я та розширення файлу.

Директива out ідентифікує параметр лише для виведення. Це еквівалентно var за винятком того, що значення не може бути змінено підпрограмою.

Функція FindFirst шукає файли, які відповідають параметрам Path та Attr, повертаючи 0, якщо перша відповідність знайдена. Результат пошуку при цьому зберігається у Rslt. Якщо відповідність не знайдена, то функція повертає негативне число і в Rslt нічого не записується.

- `FindNext - функція шукає наступний відповідний файл, як задано у параметрах пошуку функції FindFirst.`

Визначається так:

```
function FindNext(var Rslt: TSearchRec): Longint;
```

Повертає 0, якщо знайдено відповідність, результат пошуку зберігається в `Rslt.`

- `FindClose` – звільняє ресурси, використовуваніфункціями `FindFirst та FindNext при` пошуку файлів. Визначено в модулі

SysUtils наступним чином:

```
   procedure FindClose (var F: TSearchRec);
```

Цю процедуру потрібно викликати завжди після закінчення пошуку.

Напишемо програму, яка шукає в поточному каталозі файл з розширенням *.pas і, якщо знаходить, виводить його в TMemo. Крім того, в TStatusBar виводить ім'я файлу, дату його створення та розмір у байтах. Нехай це відбувається у момент показу головної форми програми, тобто. в обробнику OnShow.

```
procedure TForm1.FormShow(Sender: TObject);
var
 srRec: TSearchRec;
 tfile: TStringList;
 str: string;
begin
  if FindFirst('*.pas', faAnyFile, srRec) <> 0 then exit;
  tfile:= TStringList.Create;
  tfile.LoadFromFile(srRec.Name);
  str: = tfile.Text;
  Memo1.Lines.Add(str);
  tfile.Free;
  StatusBar1.SimpleText:='Файл: '+srRec.Name+', створений '
  + DateToStr(FileDateToDateTime(srRec.Time)) +
  ', розмір: '+ IntToStr(srRec.Size) +
  'байтів'; FindClose(srRec);
end;
```

Тут для перетворення системного формату дати та часу на звичайний

формат дата-час ми скористалися функцією FileDateToDateTime(), а для перетворення на рядок функцією DateToStr().

Зверніть увагу, при завантаженні файлу в TMemo ми не стали перетворювати його на кодування UTF-8, як це ми робили при завантаженні текстового файлу. Справа в тому, що файл з вихідним кодом модуля з розширенням *.pas створюється Lazarus і він уже в кодуванні UTF-8!

Також нам знадобиться функція

```
function SetCurrentDir(const NewDir: string): boolean;
Встановлює поточну директорію NewDir, повертаючи true у
разі успіху.
```

Наступні дві функції потрібні лише тим, хто використовує Windows. Затятим "лінуксоїдам", які "на дух" не переносять Windows, опис цих функцій можуть пропустити.

Функція GetLogicalDriveStrings повертає список усіх дисків, зареєстрованих у Windows. Опис:

```
function GetLogicalDriveStrings(nBufferLength: DWORD;
                                 lpBuffer: LPSTR): DWORD;
```

Параметр nBufferLength визначає максимальний розмір масиву (буфера) символів, вказаних у параметрі lpBuffer.

Параметр lpBuffer – покажчик типу PChar на масив, який містить рядки з нульовим символом (#0) наприкінці.

У разі успішного завершення роботи функції, значення, що повертається, є загальною довжиною (у символах) всіх рядків скопійованих в буфер. У масиві lpBuffer містяться імена всіх дисків (включаючи логічні диски, CD-ROM, знімні носії, такі як флешки, дискети і т.д.),

ні один від одного завершальним нулем #0.

Якщо функція зазнає невдачі, значення, що повертається, є нулем. Функція GetDriveType - визначає та повертає тип носія. Опис-

ня:

```
function GetDriveType(lpRootPathName: LPCSTR):UINT;
```

GetDriveType використовує лише один параметр – покажчик на пристрій. Функція повертає одне з наступних значень:

- 0 – мнемонічне позначення Drive_Unknown: диск не визначено чи не існує;
- 1 - Drive_No_Root_Dir: неправильний шлях;
- 2 – Drive_Removable: знімний пристрій (дискета, флешка тощо);
- 3 – Drive_Fixed: тип пристрою – фіксований (жорсткий диск);
- 4 - Drive_Remote: віддалений (мережевий) диск;
- 5 – Drive_CDROM: це пристрій CD-ROM;
- 6 - Drive_RAMDisk: віртуальний диск, створений в оперативній пам'яті.

Напишемо функцію, яка визначає імена всіх розділів жорсткого диска (або дисків, якщо на комп'ютері є кілька). Для простоти розглядатимемо лише жорсткі диски.

```
function Find_Logical_Disks(): boolean;
const
  Hard_Disk = 3; // розглядаємо лише жорсткі диски
var
  size: LongWord;
  Drives: array[0..128] of char;
  pDrive: PChar;
```

```
begin
  size:= GetLogicalDriveStrings(SizeOf(Drives), Drives);
  if size = 0 then
  begin
    Result: =
    false; exit;
  end;
  if size > SizeOf(Drives) then
  raiseException.Create(SysErrorMessage(ERROR_OUTOFMEMORY));
  pDrive: = Drives; // встановлюємо покажчик на
  Drives while pDrive^ <> #0 do
  begin
     // якщо тип пристрою жорсткий диск
    if GetDriveType(pDrive) = Hard_Disk the
    begin
      s: = pDrive;
      s:= Copy(s, 1, 2); // беремо тільки ім'я розділу та
      двокрапку
      Memo1.Lines.Add(s); // додаємо ім'я розділу Memo1
    end;inc(pDrive,
    4);
  end;
end;
```

Тут питання може викликати оператор

```
inc(pDrive, 4);
```

Справа в тому, що функція GetLogicalDriveStrings() записує в масив (буфер) Drives імена пристроїв у вигляді 'Однолітерний символ імені пристрою:\#0', наприклад 'C:\#0', тобто. кожен рядок масиву складається з че-

три символи.

Знайдені імена логічних дисків ми зберігаємо в Memo1, причому ми вирізали два останні символи (зворотний сліш та #0). Можна, звичайно, використовувати масив рядків. Але оскільки кількість встановлених на тому чи іншому комп'ютері жорстких дисків і створених в них розділів нам невідома, довелося б використовувати динамічний масив. До того ж довелося б цей масив передавати в інші функції та процедури. Тому простіше використовувати Memo1, адже, по суті, це і є динамічний масив рядків! Просто в програмі ми його не показуватимемо, встановивши властивість Visible у false.

За допомогою наступної процедури ми записуємо в TTreeView вміст Memo1.

```
procedure TForm1.SetAllDirectories;
var
  i: integer;
  Node: TTreeNode;
begin
  TreeView1.BeginUpdate;
  for i:= 0 to Memo1.Lines.Count - 1 do
  begin
    Node = TreeView1.Items.AddChild (nil, Memo1.Lines [i]);
    Node.ImageIndex:=0;
    Node.SelectedIndex:= 0;
    Node.HasChildren:= true;
  end;
  TreeView1.EndUpdate;
end;
```

Зверніть увагу, якщо ви заносите в TTreeView кілька елементів (вузлів), то може виникнути неприємне мерехтіння екрана. Тому ми використовували TreeView1.BeginUpdate перед початком запису вузлів та TreeView1.EndUpdate після завершення запису.

Далі ми вказали, що вузол має дочірні підвузли оператором

```
Node.HasChildren:= true;
```

У цьому випадку поруч вузла з'явиться значок розкриття вузла. Ну, що ж, тепер ми можемо реалізувати завдання відображення дерева папок та каталогів у TTreeView. Створіть новий проект. Помістіть на форму компоненти TTreeView, TMemo та TImageList. Код програми:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, ComCtrls, StdCtrls
  {$IFDEF WINDOWS}
   Windows
  {$ENDIF};
type
  {TForm1}
  TForm1 = class(TForm)
    ImageList1: TImageList;
    Memo1: TMemo;
    TreeView1: TTreeView;
    procedure FormCreate(Sender: TObject);
```

```
    procedure TreeView1Change(Sender: TObject;
                              Node: TTreeNode);
    procedure TreeView1Expanding(Sender: TObject;
       Node: TTreeNode; var AllowExpansion: Boolean);
    function Real_Directory(name: string): boolean;
    procedure Show_Only_Dir(ParentNode: TTreeNode);
    {$IFDEF WINDOWS}
    function Find_Logical_Disks(): boolean;
    procedure SetAllDirectories;
    {$ENDIF}
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{TForm1}

function TForm1.Real_Directory(sname: string): boolean;
begin
  result:= (sname <> '.') and (sname <> '..');
end;

{$IFDEF WINDOWS}
// Ці функції потрібні лише для Windows
function TForm1.Find_Logical_Disks(): boolean;
const
```

```
  Hard_Disk = 3; // розглядаємо лише жорсткі диски
var
  size: LongWord;
  Drives: array[0..128] of char;
  pDrive: PChar;
  s: string;
begin
  size:= GetLogicalDriveStrings(SizeOf(Drives), Drives);
  if size = 0 then
  begin
    Result: =
    false; exit;
  end;
  if size > SizeOf(Drives) then
    raiseException.Create(SysErrorMessage(ERROR_OUTOFMEMORY));
  pDrive: = Drives; // встановлюємо покажчик на
  Drives while pDrive^ <> #0 do
  begin
     // якщо тип пристрою жорсткий диск
    if GetDriveType(pDrive) = Hard_Disk the
    begin
      s: = pDrive;
      s:= Copy(s, 1, 2); // беремо тільки ім'я розділу та
      двокрапку
      Memo1.Lines.Add(s); // додаємо ім'я розділу Memo1
    end;inc(pDrive,
    4);
  end;
end;
```

```
procedure TForm1.SetAllDirectories;
var
  i: integer;
  Node: TTreeNode;
begin
  TreeView1.BeginUpdate;
  for i:= 0 to Memo1.Lines.Count - 1 do
  begin
    Node:= TreeView1.Items.AddChild(nil, Memo1.Lines[i]);
    Node.ImageIndex:=0;
    Node.SelectedIndex:= 0;
    Node.HasChildren:= true;
  end;
  TreeView1.EndUpdate;
end;
{$ENDIF}
procedure TForm1.FormCreate(Sender: TObject);
var
  Node: TTreeNode;
  srNode, srChild: TSearchRec;
  searchMask: string;
  SetDirWin: boolean = false;
begin
  Memo1.Clear;Memo1.Visible:
  = false;
  TreeView1.Images:= ImageList1;
  TreeView1.ExpandSignType:= tvestPlusMinus;
  TreeView1.BeginUpdate;
    {$IFDEF WINDOWS}
```

```
    // Визначення логічних
  дисків if
  Find_Logical_Disks() then
  begin
    SetAllDirectories;Se
    tDirWin: = true;
  end
  else
  {Якщо сталася помилка у функції Find_Logical_Disks(),
    то вибираємо кореневий каталог поточного диска SetCurrentDir('\');
{$ELSE}
  SetCurrentDir('/'); // кореневий каталог у Linux
{$ENDIF}
if not SetDirWin then
begin
  path:= GetCurrentDir;
  if FindFirst(path + '*', faDirectory, srNode) = 0
  then
  begin
    repeat
      // Показуємо лише каталоги
      if (srNode.Attr and faDirectory <> 0)
      and (Real_Directory(srNode.Name)) then
      begin
        Node:= TreeView1.Items.AddChild(nil,
              SysToUTF8(srNode.Name));
        Node.ImageIndex:=0;
        Node.SelectedIndex:= 0;
        {$IFDEF WINDOWS}
```

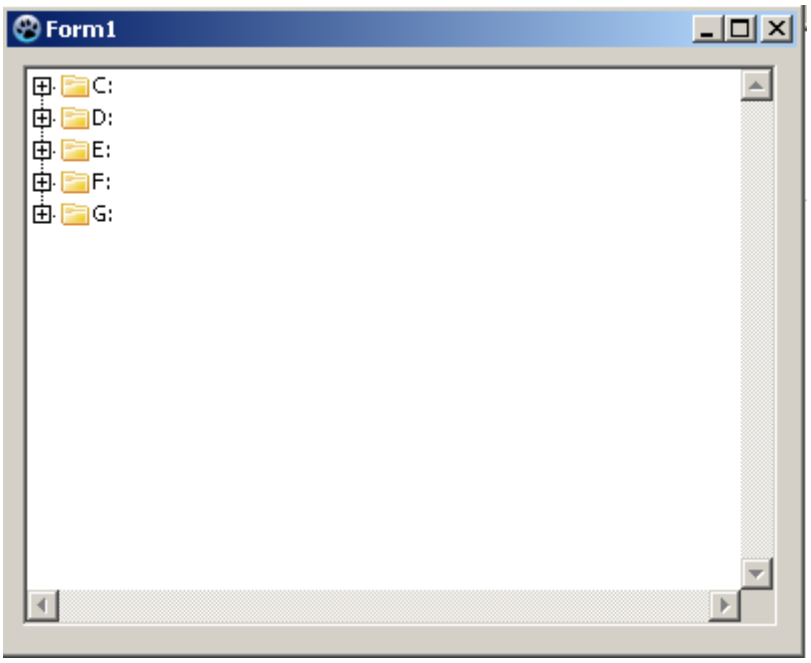
```
            searchMask:= path + srNode.Name + '\*';
            {$ELSE}
              searchMask:= path + srNode.Name + '/*';
            {$ENDIF}
            if FindFirst(searchMask, faDirectory,
               srChild) = 0 then
            repeat
              if (srChild.Attr and faDirectory <> 0) and
                  Real_Directory(srChild.Name)
              then Node.HasChildren:= true;
            until (FindNext(srChild) <> 0) or
                   Node.HasChildren;
            // Визволення зайнятих ресурсів
            SysUtils.FindClose(srChild);
          end;
        until FindNext(srNode) <> 0;
        // Визволення зайнятих ресурсів
        SysUtils.FindClose(srNode);
      end;
    end;
  TreeView1.EndUpdate;
end;
procedure TForm1.Show_Only_Dir(ParentNode: TTreeNode);
var
  srNode, srChild: TSearchRec;
  Node: TTreeNode;
  path: string;
  searchMask: string;
```

```
begin
  Node:= ParentNode;
  path:='';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node=nil;
  if FindFirst(path + '*', faDirectory, srNode) = 0 then
  repeat
  if (srNode.Attr and faDirectory <> 0) and
      Real_Directory(srNode.Name) then
  begin
    Node:= TreeView1.Items.AddChild(ParentNode,
           SysToUTF8(srNode.Name));
    Node.ImageIndex:=0;
    Node.SelectedIndex:= 0;
    {$IFDEF WINDOWS}
      searchMask:= path + srNode.Name + '\*';
    {$ELSE}
      searchMask:= path + srNode.Name + '/*';
    {$ENDIF}
    if FindFirst(searchMask, faDirectory, srChild) = 0
then
    repeat
      if (srChild.Attr and faDirectory <> 0) and
```

```
          Real_Directory(srChild.Name)the
      n Node.HasChildren:= true;
    until (FindNext(srChild) <> 0) або Node.HasChildren;
    SysUtils.FindClose(srChild);
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.TreeView1Change(Sender: TObject; Node:
TTreeNode);
begin
  if Node = nil then exit;
  TreeView1.BeginUpdate;
  Node.DeleteChildren;
  Show_Only_Dir(Node);
  Node.Expanded:= true;
  TreeView1.EndUpdate;
end;
procedure TForm1.TreeView1Expanding(Sender: TObject;
Node: TTreeNode;
  var AllowExpansion: Boolean);
begin
  TreeView1.BeginUpdate;
  Node.DeleteChildren;
  Show_Only_Dir(Node);
  TreeView1.EndUpdate;
end;
initialization
```

```
  {$I unit1.lrs}
end.
```

Розглянемо процедуру FormCreate. У ній, якщо це Windows, за допомогою функції Find_Logical_Disks() та процедур SetAllDirectories визначаються імена розділів жорстких дисків і виводяться в TreeView1. Вікно програми для Windows матиме вигляд, рис. 6.76:

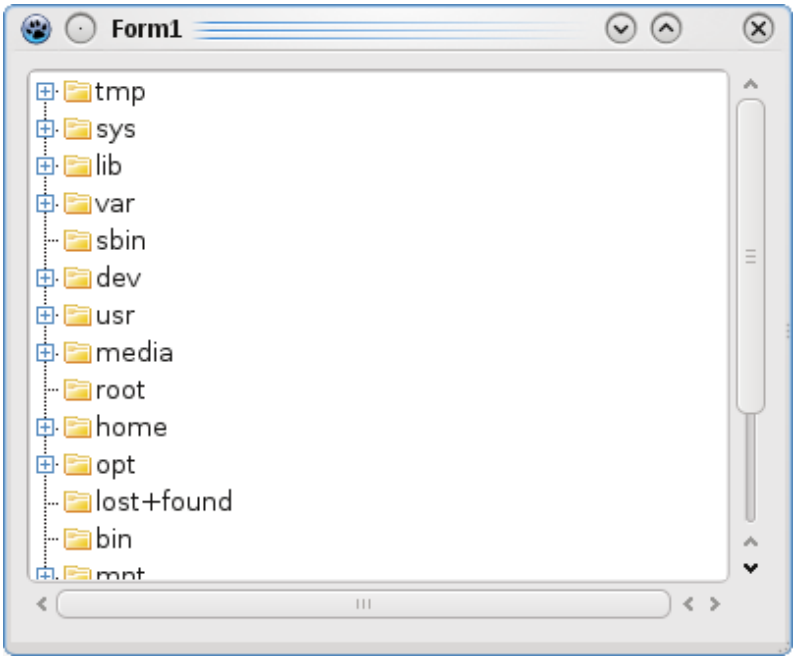


Мал. 6.76. Вигляд вікна програми у Windows

У Linux файлова система організована зовсім інакше. Тому ми оператором

```
SetCurrentDir('/');
```

встановлюємо кореневий каталог і програма сканує каталоги тільки верхнього рівня, рис. 6.77.



Мал. 6.77. Вигляд вікна програми в Linux

Таким чином, наша програма не сканує всі диски та каталоги. Програма спочатку лише визначає літери дисків та розділів дисків (якщо це Windows) або сканує лише каталоги найвищого рівня, розташовані у кореневому каталозі (якщо це Linux).

Лише при натисканні користувачем на будь-який вузол або кнопку розкриття цього вузла, програма почне сканувати каталоги, розташовані на один рівень нижче і тільки для поточного каталогу. Поясню прикладом. Нехай ми знаходимося в каталозі C: lazarus components lazreport або `/usr/lib/lazarus/components/lazreport`.

При натисканні на вузол з ім'ям lazreport програма просканує цей каталог і знайде п'ять підкаталогів doc, images, samples, source, tools. При цьому ці підкаталоги мають у свою чергу ще вкладені підкаталоги, але всіх їх програма зараз сканувати не буде! Для того щоб визначити, що, наприклад, каталог samples має вкладені підкаталоги, програмі достатньо знайти тільки один підкаталог (перший, що попався!). У нашому випадку це буде каталог barcode.

Це дозволяє значно прискорити роботу програми.

Розглянемо процедуру Show_Only_Dir, яка й реалізує викладений алгоритм. Процедура викликається у обробниках OnChange та OnExpanding. Спочатку процедура визначає повний шлях до поточного каталогу, рухаючись знизу вгору від дочірнього вузла до батьківського до кореневого. Потім запускається цикл пошуку FindFirst – FindNext за допомогою якого знаходяться всі підкаталоги поточного каталогу. Вкладений цикл FindFirst – FindNext дозволяє визначити, чи є у знайдених підкаталогів вкладені підкаталоги. Цей внутрішній цикл завершується, як тільки вкладений каталог знайдено, а його батько позначається ознакою Node.HasChildren: = true.

Зазначимо, що перед викликом процедури Show_Only_Dir ми оператором

```
Node.DeleteChildren
```

видаляємо всі нащадки поточного вузла, оскільки процедура наново формує їх. Точно за таким же алгоритмом сканується кореневий каталог в обробнику `OnCreate`.

Ось, власне, і все! Залишилося пояснити, навіщо призначена функція `Real_Directory()`.

Як відомо, у кожному каталозі є два спеціальні записи. Одна з них позначається просто точкою і є посиланням на поточний каталог, а другий запис, що позначається двома точками (за стандартом і читається точка-точка) є посиланням на батьківський каталог. Функції FindFirst–FindNext їх знаходять та ідентифікують як каталоги. Але зазвичай у дереві каталогів їх не прийнято показувати. Тому функція Real_Directory() перевіряє, чи не є ім'я знайденого каталогу точкою або точкою-точкою та повертає

`true, якщо це не так.`

Компонент TListView

На відміну від TListBox, компонент TListView дозволяє відображати дані в різних стилях або режимах. Так, дані можуть відображатися у вигляді списку, великих або дрібних піктограм і, нарешті, таблиці. Стиль відображення визначається властивістю ViewStyle, табл. 6.5.

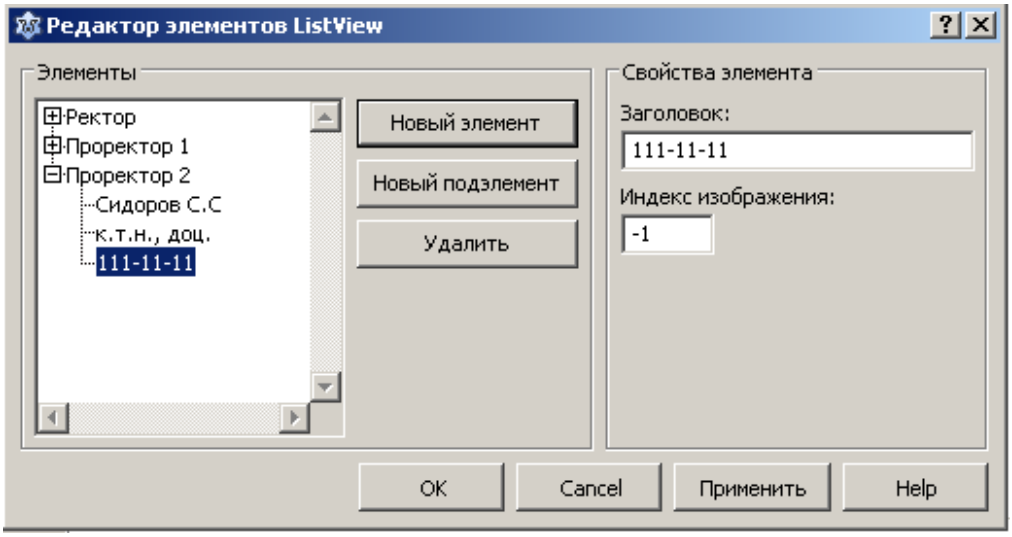
Елементи списку містяться у властивості Items об'єкта TListItem. Компонент можна заповнювати як вручну, під час проектування, так і програмно. Для ручного заповнення є редактор, схожий редактор TTreeView. У ньому так само задаються "Новий елемент" і "Новий поделемент". На відміну від TTreeView в TListView допускається трохи більше рівня вкладеності поделементов.

Таблиця 6.5

Значення	Стиль відображення даних
`vsIcon`	Елементи списку з'являються у вигляді великих значків із написом під ними. Картинки для великих значків зберігаються в компоненті TImageList, вказаному у властивості LargeImages. Можливе їхнє перетягування.
`vsSmallIcon`	Елементи списку з'являються як маленькі значки з написом праворуч. Зображення для маленьких значків зберігаються у компоненті TImageList, зазначеному у властивості SmallImages. Можливе їхнє перетягування
`vsList`	Елементи списку з'являються в колонці один під одним із написом праворуч. Перетягування невозможна

vsReport	Елементи списку з'являються у кількох колонках один під одним. У першій міститься маленький значок і напис, у решті — певна програмістом інформація. Якщо властивість Show-ColumnHeaders встановлено на значення true, колонки забезпечуються заголовками

Як і TTreeView зображення необхідно попередньо занести в TImageList, потім вказувати індекс потрібного зображення у властивості ImageIndex, рис. 6.78.

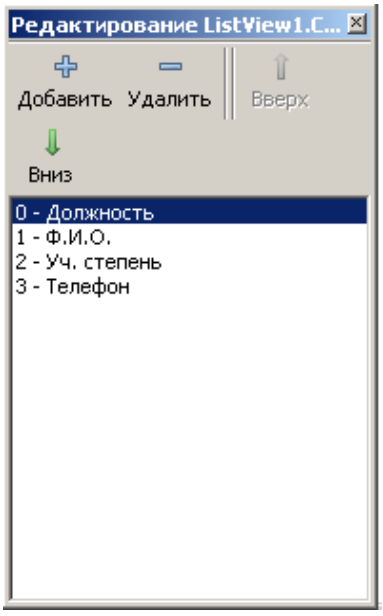


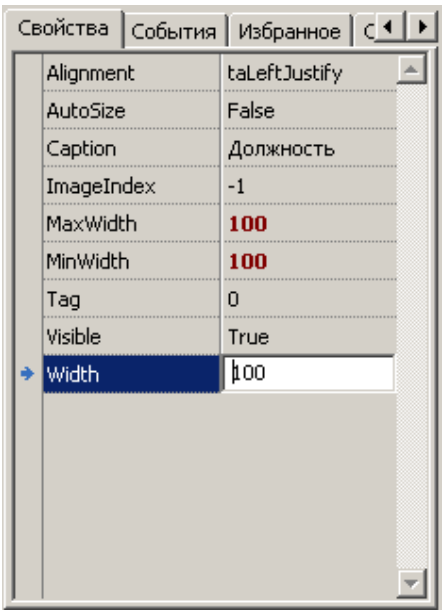
Мал. 6.78. Редактор елементів TListView

Але поделементи будуть видно тільки в режимівsReport. Для того, щоб інформація, введена на рис. 6.78. було видно, необхідно створити чотири колонки (стовпця). Для цього в Інспекторі об'єктів натисніть кнопку з трьома крапками навпроти властивості Columns, з'явиться редактор колонок рис. 6.79. Щоб додати колонку, натисніть кнопку "Додати". В Інспекторі об'єктів з'являться властивості цієї колонки, рис. 6.80. Свойст-

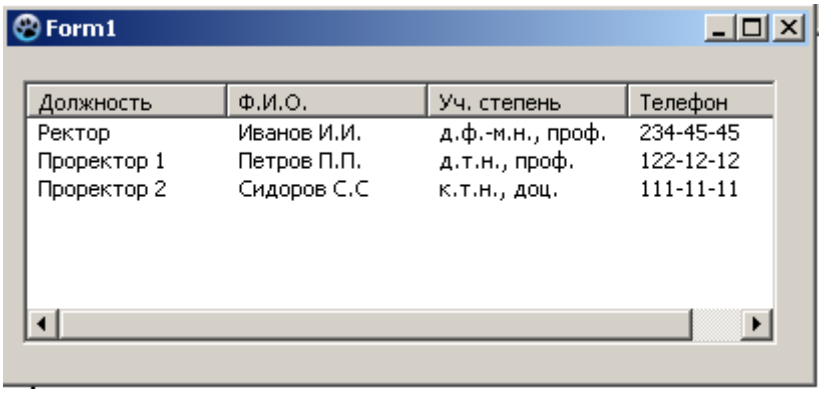
в Caption задає заголовок колонки, якість Width ширину колонки. Ус-

тановіть необхідні властивості кожного з чотирьох колонок і запустіть програму, рис. 6.81.





Мал. 6.79. Редактор колонок Мал. 6.80. Властивості колонки



Мал. 6.81. Вікно програми

Розглянемо основні властивості компонента:

- `Columns – містить об'єкти – колонки (стовпці). Колонки видно лише для стилю відображення ViewStyle = vsReport. При цьому має бути певна хоча б одна колонка. Інакше елементи не буде видно;`
- Column[Index: integer] – відкриває доступ до колонок елементів за їх індексами;
- `ColumnClick: boolean – дозволяє чи забороняє генерацію події`

OnColumnClick;

- GridLines: boolean – дозволяє або забороняє показ ліній у режимі

ViewStyle = vsReport;

- HideSelection: boolean – забороняє або дозволяє зберігати вибір елементів у разі втрати компонентом фокуса;
- Items: TListItems – містить перелік всіх елементів;
- LargeImages: TImageList – вказує джерело великих піктограм;
- MultiSelect: boolean - дозволяє/забороняє множинний вибір;
- ReadOnly: boolean - забороняє/дозволяє редагування написів;
- ShowColumnHeaders: boolean - дозволяє/забороняє показ заголовків колонок як ViewStyle = vsReport;
- SmallImages – вказує на джерело дрібних піктограм;
- SortType – вказує спосіб сортування елементів, можливі значення

stNone, stData, stText, stBoth;

- StateImages – вказує джерело піктограм для обраних елементів;

Властивості та методи класу TListItems:

- Item[Index: integer]: TListItem – відкриває індексний доступ до елементів списку;
- Count – містить кількість елементів у Item;
- function Add: TListItem – додає ще один елемент до списку;
- procedure Clear – очищує список;
- procedure BeginUpdate – блокує оновлення екрана, доки не буде виконано метод EndUpdate. Використовується для одночасної вставки кількох елементів списку для запобігання мерехтіння екрана;

- procedure Delete(Index: Integers) – видаляє елемент списку з

індексом Index;

- procedure EndUpdate - скасовує блокування оновлення екрану методом BeginUpdate;
- function IndexOf(Value: TListItem) – повертає індекс елемента Value;
- function Insert(Index: integer): TListItem – вставляє новий елемент після елемента, заданого Index;

Дуже важливе властивість SubItems: TStrings.

За допомогою цієї властивості з кожним елементом списку може бути пов'язаний цілий набір рядків та об'єктів. Спочатку слід створити необхідну кількість колонок. При цьому слід врахувати, що перша буде відведена під сам елемент списку. Наступні колонки будуть відображати тексти рядків з властивості Items.SubItems.

Заповнимо програмно TListView даними з рис. 6.78. У новому проекті помістіть компонент TListView і в обробнику OnShow форми введіть наступний код:

```
procedure TForm1.FormShow(Sender: TObject);
var
  ListItem: TListItem;
begin
  with ListView1 do
  begin
    ViewStyle:= vsReport;
    // Створюємо колонку
    Columns.Add;
    // заголовок колонки
    Columns.Items[0].Caption:='Посада';
```

```
// ширина колонки
Columns.Items[0].Width:= 100;
// Створюємо колонку
Columns.Add;
// заголовок колонки
Columns.Items[1].Caption:='П.І.Б.';
// ширина колонки
Columns.Items[1].Width:= 100;
// Створюємо колонку
Columns.Add;
// заголовок колонки
Columns.Items[2].Caption:='Уч. ступінь';
// ширина колонки
Columns.Items[2].Width:= 100;
// Створюємо колонку
Columns.Add;
// заголовок колонки
Columns.Items[3].Caption:='Телефон';
// ширина колонки
Columns.Items[3].Width:= 100;
// Додаємо перший елемент
ListItem:= Items.Add;
ListItem.Caption:= 'Ректор';
// додаємо поделементи першого
елемента
ListItem.SubItems.Add('Іванов І.І.');
ListItem.SubItems.Add('д.ф.-м.н.,
проф.'); ListItem.SubItems.Add('234-
45-45');
// додаємо другий елемент
```

```
    ListItem:= Items.Add;
    ListItem.Caption:= 'Проректор 1';
    // Додаємо поделементи другого
    елемента
    ListItem.SubItems.Add('Петров
    П.П.');
    ListItem.SubItems.Add('д.т.н.,
    проф.');
    ListItem.SubItems.Add('122-12-
    12');
    // Додаємо третій елемент
    ListItem: = Items.Add;
    ListItem.Caption:= 'Проректор
    2';
    // додаємо поделементи третього
    елемента
    ListItem.SubItems.Add('Сидоров
    С.С.');
    ListItem.SubItems.Add('к.т.н.,
    доц.'); ListItem.SubItems.Add('111-
    11-11');
  end;
end;
```

Продовжимо наші досліди з Провідником. Давайте тепер реалізуємо праве вікно Провідника. Для цієї мети компонент TListView підходить якнайкраще. Створіть новий проект. Помістіть форму TTreeView, а потім компонент TListView. Не забудьте помістити на форму також TMemo та TImageList.

Якщо ми подивимося, як працює Провідник, ми побачимо, що при натисканні кнопки розкриття вузла-каталога в лівому вікні, вузол просто

розкривається, а в правому вікні нічого не з'являється. Якщо ж ми натиснемо на сам вузол-каталог, він також розкривається і, водночас, вміст цього каталогу (підкаталоги та файли) з'являються в ListView1.

Для відображення вмісту вузла-каталогу можна використовувати ту ж саму процедуру Show_Only_Dir(), тільки знайдені підкаталоги тепер

будуть додаватися не тільки TTreeView, але й TListView. Але ця процедура використовується для виведення каталогів у лівому вікні, тому створимо процедуру з іншим ім'ям. Для виведення файлів, якщо в цьому підкаталозі вони є, також створимо процедуру. Алгоритм той самий, тільки функції FindFirst-FindNext будуть шукати вже не каталоги, а файли. Причому приховані файли ми не показуватимемо. Код програми:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,
             Forms,Controls, Graphics, Dialogs, ComCtrls,
  StdCtrls, ExtCtrls, LCLType,

  {$IFDEF UNIX}
    unix;
  {$ELSE}
    Windows;
  {$ENDIF}

type
  {TForm1}
  TForm1 = class(TForm)
    ImageList1: TImageList;
    ListView1: TListView;
    Memo1: TMemo;
    TreeView1: TTreeView;
    procedure FormCreate(Sender: TObject);
```

```
    procedure TreeView1Change(Sender: TObject;
                                      Node: TTreeNode);
    procedure TreeView1Expanding(Sender: TObject;
       Node: TTreeNode; var AllowExpansion: Boolean);
    function Real_Directory(name: string): boolean;
    procedure Show_Only_Dir(ParentNode: TTreeNode);
    procedureShow_in_ListView_Dir(ParentNode:
    TTreeNode);procedureShow_in_ListView_Files(ParentNode:
    TTreeNode);

    {$IFDEF WINDOWS}
    function Find_Logical_Disks(): boolean;
    procedure SetAllDirectories;

    {$ENDIF}
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;
  path: string;
implementation

{TForm1}

function TForm1.Real_Directory(sname: string): boolean;
begin
```

```
  result:= (sname <> '.') and (sname <> '..');
end;

{$IFDEF WINDOWS}
// Ці функції потрібні лише для Windows
function TForm1.Find_Logical_Disks(): boolean;
const
  Hard_Disk = 3; // розглядаємо лише жорсткі диски
var
  size: LongWord;
  Drives: array[0..128] of char;
  pDrive: PChar;
  s: string;
begin
  size:= GetLogicalDriveStrings(SizeOf(Drives), Drives);
  if size = 0 then
  begin
    Result: =
    false; exit;
  end;
  if size > SizeOf(Drives) then
    raiseException.Create(SysErrorMessage(ERROR_OUTOFMEMORY));
  pDrive: = Drives; // встановлюємо покажчик на
  Drives while pDrive^ <> #0 do
  begin
    // якщо тип пристрою жорсткий диск
    if GetDriveType(pDrive) = Hard_Disk the
    begin
      s: = pDrive;
```

```
      s:= Copy(s, 1, 2); // беремо тільки ім'я розділу та
      двокрапку
      Memo1.Lines.Add(s); // додаємо ім'я розділу Memo1
    end;inc(pDrive,
    4);
  end;
end;

procedure TForm1.SetAllDirectories;
var
  i: integer;
  Node: TTreeNode;
begin
  TreeView1.BeginUpdate;
  for i:= 0 to Memo1.Lines.Count - 1 do
  begin
    Node:= TreeView1.Items.AddChild(nil, Memo1.Lines[i]);
    Node.ImageIndex:=0;
    Node.SelectedIndex:= 0;
    Node.HasChildren:= true;
  end;
  TreeView1.EndUpdate;
end;
{$ENDIF}

procedure TForm1.FormCreate(Sender: TObject);
var
  Node: TTreeNode;
  srNode, srChild: TSearchRec;
  searchMask: string;
```

```
  SetDirWin: boolean = false;
begin
  Memo1.Clear;Memo1.Visible:
  = false;
  TreeView1.Images:= ImageList1;
  TreeView1.ExpandSignType:= tvestPlusMinus;
  TreeView1.BeginUpdate;
    {$IFDEF WINDOWS}
       // Визначення логічних
      дисків if
      Find_Logical_Disks() then
      begin
        SetAllDirectories;Se
        tDirWin: = true;
      end
      else
      {Якщо сталася помилка функції
       Find_Logical_Disks(), то вибираємо кореневий
       каталог поточного диска}
        SetCurrentDir('\');
    {$ELSE}
      SetCurrentDir('/'); // кореневий каталог у Linux
    {$ENDIF}
    if not SetDirWin then
    begin
      path:= GetCurrentDir;
      if FindFirst(path + '*', faDirectory, srNode) = 0
      then
      begin
        repeat
```

```
// Показуємо лише каталоги
```

```
        if (srNode.Attr and faDirectory <> 0)
        and (Real_Directory(srNode.Name)) then
        begin
        Node:= TreeView1.Items.AddChild(nil,
               SysToUTF8(srNode.Name));
        Node.ImageIndex:=0;
        Node.SelectedIndex:= 0;
        {$IFDEF WINDOWS}
          searchMask:= path + srNode.Name + '\*';
          {$ELSE}
            searchMask:= path + srNode.Name + '/*';
          {$ENDIF}
          if FindFirst(searchMask, faDirectory,
             srChild) = 0 then
          repeat
            if (srChild.Attr and faDirectory <> 0) and
                Real_Directory(srChild.Name)
            then Node.HasChildren:= true;
          until (FindNext(srChild) <> 0) or
                 Node.HasChildren;
          // Визволення зайнятих ресурсів
          SysUtils.FindClose(srChild);
        end;
      until FindNext(srNode) <> 0;
      // Визволення зайнятих ресурсів
      SysUtils.FindClose(srNode);
    end;
  end;
TreeView1.EndUpdate;
```

```
  // Встановлення властивостей ListView1
  with ListView1 do
  begin
    Align: = alClient;
    ViewStyle:= vsReport;
    Columns.Add;
    Columns.Items[0].Width:= 425;
    ScrollBars:= ssAutoBoth;
    SmallImages:= ImageList1;
  end;
end;

procedure TForm1.Show_Only_Dir(ParentNode: TTreeNode);
var
  srNode, srChild: TSearchRec;
  Node: TTreeNode;
  searchMask: string;
begin
  Node:= ParentNode;
  path:='';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node=nil;
  if FindFirst(path + '*', faDirectory, srNode) = 0 then
```

```
  repeat
  if (srNode.Attr and faDirectory <> 0) and
      Real_Directory(srNode.Name) then
  begin
    Node:= TreeView1.Items.AddChild(ParentNode,
           SysToUTF8(srNode.Name));
    Node.ImageIndex:=0;
    Node.SelectedIndex:= 0;
    {$IFDEF WINDOWS}
      searchMask:= path + srNode.Name + '\*';
    {$ELSE}
      searchMask:= path + srNode.Name + '/*';
    {$ENDIF}
    if FindFirst(searchMask, faDirectory, srChild) = 0
then
    repeat
      if (srChild.Attr and faDirectory <> 0) and
          Real_Directory(srChild.Name)
      then Node.HasChildren:= true;
    until (FindNext(srChild) <> 0) або Node.HasChildren;
    SysUtils.FindClose(srChild);
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.Show_in_ListView_Dir(ParentNode:
                                      TTreeNode);
var
```

```
  srNode, srChild: TSearchRec;
  Node: TTreeNode;
  ListItem: TListItem;
  searchMask: string;
begin
  Node:= ParentNode;
  path:='';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node=nil;
  if FindFirst(path + '*', faDirectory, srNode) = 0 then
  repeat
  if (srNode.Attr and faDirectory <> 0) and
      Real_Directory(srNode.Name) then
  begin
    Node:= TreeView1.Items.AddChild(ParentNode,
           SysToUTF8(srNode.Name));
    if srNode.Attr and faHidden = 0 then
    begin
      ListItem:= ListView1.Items.Add;
      ListItem.Caption:= Node.Text;
      ListItem.ImageIndex:= 0;
    end;
    {$IFDEF WINDOWS}
```

```
      searchMask:= path + srNode.Name + '\*';
    {$ELSE}
      searchMask:= path + srNode.Name + '/*';
    {$ENDIF}
    if FindFirst(searchMask, faDirectory, srChild) = 0
    then
    repeat
      if (srChild.Attr and faDirectory <> 0) and
          Real_Directory(srChild.Name)
      then Node.HasChildren:= true;
    until (FindNext(srChild) <> 0) або Node.HasChildren;
    SysUtils.FindClose(srChild);
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.Show_in_ListView_Files(ParentNode:
                                        TTreeNode);
var
  srNode: TSearchRec;
  Node: TTreeNode;
  ListItem: TListItem;
begin
  Node:= ParentNode;
  path:='';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
```

```
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node=nil;
  if FindFirst(path + '*', faAnyFile, srNode) = 0 then
  repeat
  if srNode.Attr and faDirectory = 0 then
  begin
    if srNode.Attr and faHidden = 0 then
    begin
      ListItem:= ListView1.Items.Add;
      ListItem.Caption:= SysToUTF8(srNode.Name);
      ListItem.ImageIndex:= 1;
    end;
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.TreeView1Change(Sender: TObject;
                                 Node: TTreeNode);
begin
  if Node = nil then exit;
  TreeView1.BeginUpdate;
  ListView1.BeginUpdate;
  ListView1.Clear;
  Node.DeleteChildren;
  Show_in_ListView_Dir(Node);
```

```
  Show_in_ListView_Files(Node);N
  ode.Expanded:= true;
  ListView1.EndUpdate;
  TreeView1.EndUpdate;
end;

procedure TForm1.TreeView1Expanding(Sender: TObject;
      Node: TTreeNode; var AllowExpansion: Boolean);
begin
  TreeView1.BeginUpdate;
  Node.DeleteChildren;
  Show_Only_Dir(Node);
  TreeView1.EndUpdate;
end;
initialization
  {$I unit1.lrs}
end.
```

Чому ми розділили висновок у TListView каталогів та файлів на дві процедури? Якщо їх виводити в одній процедурі, вони відображатимуться вперемішу. Адже прийнято, щоб спочатку відображалися каталоги, і лише потім файли. Тут сортування не допоможе.

Зверніть увагу, що для файлів ми вибрали іншу піктограму.

Тепер нам потрібно реалізувати розкриття папки в TListView за подвійним натисканням на його ім'я. Для цього напишемо дві процедури, одна з яких показує лише каталоги, а інша лише файли. Викликатимуться в обробнику події OnDblClick для TListView.

```
procedure TForm1.Show_ListView_Dir(DirName: string);
var
```

```
  srNode: TSearchRec;
  ListItem: TListItem;
begin
  oldpath:= path;
  {$IFDEF WINDOWS}
  path:= path + UTF8ToSys(DirName) + '\';
  {$ELSE}
  path:= '/' + path + DirName + '/';
  {$ENDIF}
  ListView1.Clear;
  if FindFirst(path + '*', faDirectory, srNode) = 0 then
  repeat
  if (srNode.Attr and faDirectory <> 0) and
      Real_Directory(srNode.Name) then
  begin
    ListItem:= ListView1.Items.Add;
    ListItem.Caption:= SysToUtf8(srNode.Name);
    ListItem.ImageIndex:= 0;
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.Show_ListView_Files(DirName: string);
var
  srNode: TSearchRec;
  ListItem: TListItem;
begin
  path:='';
```

```
  {$IFDEF WINDOWS}
  path:= oldpath + UTF8ToSys(DirName) + '\';
  {$ELSE}
  path:= '/' + oldpath + DirName + '/';
  {$ENDIF}
  if FindFirst(path + '*', faAnyFile, srNode) = 0 then
  repeat
  if srNode.Attr and faDirectory = 0 then
  begin
    if srNode.Attr and faHidden = 0 then
    begin
      ListItem:= ListView1.Items.Add;
      ListItem.Caption:= SysToUTF8(srNode.Name);;
      ListItem.ImageIndex:= 1;
    end;
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.ListView1DblClick(Sender: TObject);
var
  s: string;
  DirName: string;
  srNode: TSearchRec;
begin
  if ListView1.Selected = nil then exit;
  {$IFDEF WINDOWS}
  s:= path + UTF8ToSys(ListView1.Selected.Caption);
```

```
  {$ELSE}
  s:= path + ListView1.Selected.Caption;
  {$ENDIF}
  if FindFirst(s, faAnyFile, srNode) = 0 then
  if (srNode.Attr and faDirectory) <> 0 then
  begin
    DirName:= ListView1.Selected.Caption;
    Show_ListView_Dir(DirName);
    Show_ListView_Files(DirName);
    SysUtils.FindClose(srNode);
  end;
end;
```

Додамо в наш Провідник ще одну функціональність, а саме по подвійному клацанню на ім'я файлу будемо його запускати. Запуск виконуваного файлу (зовнішньої програми) у Windows найпростіше організувати за допомогою функції WinExec. Ця функція визначена так:

```
function WinExec(lpCmdLine:LPCSTR; uCmdShow:UINT):UINT;
```

Параметр lpCmdLine є вказівником на рядок з нульовим символом у кінці, що містить ім'я файлу. Можна увімкнути повний шлях до файлу і, якщо потрібно, параметри командного рядка. Якщо ім'я файлу вказано без шляху, то Windows шукатиме в каталогах виконуваний файл у наступній послідовності:

- Каталог, з якого завантажено додаток;•

Поточний каталог;

- Системний каталог Windows, повертається функцією `GetSystemDirectory`;
- Каталог Windows, що повертається функцією GetWindowsDirectory;

- Список каталогів із змінної оточення PATH.

Параметр uCmdShow визначає форму представлення вікна програми Windows, що запускається. Найчастіше використовується значення 1, при якому вікно програми, що запускається, активізується і відображається на екрані. Якщо це вікно в даний момент згорнуто або розгорнуто, воно відновлюється до своїх початкових розмірів і відображається в початковій позиції.

У Linux файл можна запустити за допомогою функції Shell.
Визначено в модулі unix:

```
function Shell(const Command: String): cint;
```

Параметр Command задає ім'я файлу шляхом або без.

Обробник події OnDblClick перепишіть так:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
var
  s: string;
  progr: pchar;
  srNode: TSearchRec;
  DirName: string;
begin
  if ListView1.Selected = nil then exit;
  {$IFDEF WINDOWS}
  s:= path + UTF8ToSys(ListView1.Selected.Caption);
  {$ELSE}
  s:= path + ListView1.Selected.Caption;
  {$ENDIF}
  if FindFirst(s, faAnyFile, srNode) = 0 then
```

```
  if (srNode.Attr and faDirectory) <> 0 then
  begin
    DirName:= ListView1.Selected.Caption;
    Show_ListView_Dir(DirName);
    Show_ListView_Files(DirName);
    exit;
  end;
  if (srNode.Attr and faHidden) = 0 then
  begin
    progr: = pchar (s);
    {$IFDEF UNIX}
      shell (progr);
    {$ELSE}
      WinExec(progr, 1);
    {$ENDIF}
  end;
end;
```

У Linux при такому способі запуску виконуваного файлу запустити консольна програма не вдасться, оскільки консольна програма повинна запускатися в консолі або терміналі. Крім того, існує більш ефективний спосіб запуску зовнішніх програм, що підходить як для Linux, так і для Windows. Це використання процесів та потоків (ниток). Не вдаючись до подробиць, зазначу, що під час запуску програми породжується процес. Якщо програма запускається інша програма, то породжується дочірній процес.

Створити та запустити дочірній процес можна наступним чином:

```
var
  AProcess: TProcess;
begin
```

```
    AProcess:= TProcess.Create(nil);
    AProcess.CommandLine:= 'Ім'я файлу, що
    виконується'; AProcess.Execute;
    AProcess.Free;
 end;
```

Перепишемо обробник OnDblClick:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
var
  s: string;
  AProcess: TProcess;
  srNode: TSearchRec;
  DirName: string;
begin
  if ListView1.Selected = nil then exit;
  {$IFDEF WINDOWS}
  s:= path + UTF8ToSys(ListView1.Selected.Caption);
  {$ELSE}
  s:= path + ListView1.Selected.Caption;
  {$ENDIF}
  if FindFirst(s, faAnyFile, srNode) = 0 then
  if (srNode.Attr and faDirectory) <> 0 then
  begin
    DirName:= ListView1.Selected.Caption;
    Show_ListView_Dir(DirName);
    Show_ListView_Files(DirName);
    exit;
  end;
```

```
  if (srNode.Attr and faHidden) = 0 then
  begin
    AProcess:= TProcess.Create(nil);
    {$IFDEF UNIX}
      if MessageDlg('Відкривати в терміналі?', mtCustom,
                    [mbYes, mbNO], 0) = mrYes then
      s:= 'xterm -T ''Lazarus Run Output''' +
          '-e $(TargetCmdLine)' + s;
    {$ENDIF}
    AProcess.CommandLine:= s;
    AProcess.Execute;
    AProcess.Free;
  end;
end;
```

Окинемо ще раз поглядом код усієї нашої програми. Ми, що у ній є п'ять схожих процедур. Звісно, вони не зовсім схожі! Кожна процедура вирішує своє завдання. Я маю на увазі, що в них є ділянки коду, що повторюються. Хотів би написати одну загальну процедуру. Зрозуміло, що ця "універсальна" процедура буде досить заплутаною через необхідність проводити численні перевірки. У таких випадках перед програмістом завжди постає дилема, що вибрати – простий і ясний код, але з дублюванням або код "з наворотами".

Все ж більшість професійних програмістів віддадуть перевагу простішому і прозорішому коду, незважаючи на те, що розмір файлу, що виконується, може виявитися більшим. Особливо, якщо планується програму надалі модифікувати. Адже через деякий час розбиратися навіть у своїй власній програмі досить важко.

Однак ми все ж таки спробуємо об'єднати ці п'ять процедур в одну. Назо-

введемо цю процедуру ShowExpandNode і введемо два додаткові параметри ShowMode: integer та IsinTListView: boolean.

Нехай ShowMode = 0, якщо потрібно вивести каталоги в TTreeView.

ShowMode = 1, якщо необхідно вивести тільки каталоги в TListView, але вузол береться з TTreeView.

ShowMode = 2, якщо потрібно вивести тільки файли в TListView

з каталогу вибраного у TTreeView.

IsinTListView = false, якщо ми перебуваємо в TTreeView.

IsinTListView = true, якщо ми перебуваємо в TListView.

Остаточно код програми виглядатиме так:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, ComCtrls, StdCtrls,
  ExtCtrls, LCLType, Process,

  {$IFDEF UNIX}
    unix;
  {$ELSE}
    Windows;
  {$ENDIF}
type

  {TForm1}
  TForm1 = class(TForm)
    ImageList1: TImageList;
```

```
    ListView1: TListView;
    Memo1: TMemo;
    TreeView1: TTreeView;
    procedure FormCreate(Sender: TObject);
    procedure ListView1DblClick(Sender: TObject);
    procedure TreeView1Change(Sender: TObject; Node:
                              TTreeNode);procedur
    e TreeView1Expanding(Sender: TObject;
          Node: TTreeNode; var AllowExpansion: Boolean);
    function Real_Directory(name: string): boolean;
    procedure ShowExpandNode(ParentNode: TTreeNode;
                    DirName: string; ShowMode: integer;
                    IsinTListView: boolean);
    function GetPath(ParentNode: TTreeNode): string;
    function GetPath_1(DirName: string;
                      p: string): string;
    {$IFDEF WINDOWS}
    function Find_Logical_Disks(): boolean;
    procedure SetAllDirectories;
    {$ENDIF}
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
  path, oldpath: string;
implementation
```

```
{TForm1}

function TForm1.Real_Directory(sname: string): boolean;
begin
  result:= (sname <> '.') and (sname <> '..');
end;

{$IFDEF WINDOWS}
// Ці функції потрібні лише для Windows
function TForm1.Find_Logical_Disks(): boolean;
const
  Hard_Disk = 3; // розглядаємо лише жорсткі диски
var
  size: LongWord;
  Drives: array[0..128] of char;
  pDrive: PChar;
  s: string;
begin
  size:= GetLogicalDriveStrings(SizeOf(Drives), Drives);
  if size = 0 then
  begin
    Result: =
    false; exit;
  end;
  if size > SizeOf(Drives) then
    raiseException.Create(SysErrorMessage(ERROR_OUTOFMEMORY));
  pDrive: = Drives; // встановлюємо покажчик на
  Drives while pDrive^ <> #0 do
```

```
  begin
    // якщо тип пристрою жорсткий диск
    if GetDriveType(pDrive) = Hard_Disk the
    begin
      s: = pDrive;
      s:= Copy(s, 1, 2); // беремо тільки ім'я розділу та
      двокрапку
      Memo1.Lines.Add(s); // додаємо ім'я розділу Memo1
    end;inc(pDrive,
    4);
  end;
end;

procedure TForm1.SetAllDirectories;
var
  i: integer;
  Node: TTreeNode;
begin
  TreeView1.BeginUpdate;
  for i:= 0 to Memo1.Lines.Count - 1 do
  begin
    Node:= TreeView1.Items.AddChild(nil, Memo1.Lines[i]);
    Node.ImageIndex:=0;
    Node.SelectedIndex:= 0;
    Node.HasChildren:= true;
  end;
  TreeView1.EndUpdate;
end;
{$ENDIF}
function TForm1.GetPath(ParentNode: TTreeNode): string;
```

```
var
  Node: TTreeNode;
begin
  Node:= ParentNode;
  path:='';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node=nil;
  Result:= path;
end;

function TForm1.GetPath_1(DirName: string;
                          p: string): string;
begin
  {$IFDEF WINDOWS}
  p:= p + UTF8ToSys(DirName) + '\';
  {$ELSE}
  p:= '/' + p + DirName + '/';
  {$ENDIF}
  Result:= p;
end;

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

```
  Node: TTreeNode;
  srNode, srChild: TSearchRec;
  searchMask: string;
  SetDirWin: boolean = false;
begin
  Memo1.Clear;Memo1.Visible:
  = false;
  TreeView1.Images:= ImageList1;
  TreeView1.ExpandSignType:= tvestPlusMinus;
  TreeView1.BeginUpdate;
    {$IFDEF WINDOWS}
       // Визначення логічних
      дисків if
      Find_Logical_Disks() then
      begin
        SetAllDirectories;Se
        tDirWin: = true;
      end
      else
      {Якщо сталася помилка функції
       Find_Logical_Disks(), то вибираємо кореневий
       каталог поточного диска}
        SetCurrentDir('\');
    {$ELSE}
      SetCurrentDir('/'); // кореневий каталог у Linux
    {$ENDIF}
    if not SetDirWin then
    begin
      path:= GetCurrentDir;
      if FindFirst(path + '*', faDirectory, srNode) = 0
```

```
    then
```

```
begin
  repeat
    // Показуємо лише каталоги
    if (srNode.Attr and faDirectory <> 0)
    and (Real_Directory(srNode.Name)) then
    begin
    Node:= TreeView1.Items.AddChild(nil,
           SysToUTF8(srNode.Name));
    Node.ImageIndex:=0;
    Node.SelectedIndex:= 0;
    {$IFDEF WINDOWS}
      searchMask:= path + srNode.Name + '\*';
      {$ELSE}
        searchMask:= path + srNode.Name + '/*';
      {$ENDIF}
      if FindFirst(searchMask, faDirectory,
         srChild) = 0 then
      repeat
        if (srChild.Attr and faDirectory <> 0) and
            Real_Directory(srChild.Name)
        then Node.HasChildren:= true;
      until (FindNext(srChild) <> 0) or
             Node.HasChildren;
      // Визволення зайнятих ресурсів
      SysUtils.FindClose(srChild);
    end;
  until FindNext(srNode) <> 0;
// Визволення зайнятих ресурсів
  SysUtils.FindClose(srNode);
```

```
      end;
    end;
  TreeView1.EndUpdate;
  // Встановлення властивостей ListView1
  with ListView1 do
  begin
    Align: = alClient;
    ViewStyle:= vsReport;
    Columns.Add;
    Columns.Items[0].Width:= 425;
    ScrollBars:= ssAutoBoth;
    SmallImages:= ImageList1;
  end;
end;

procedure TForm1.ShowExpandNode(ParentNode: TTreeNode;
                      DirName: string; ShowMode: integer;
                                 IsinTListView: boolean);
var
  srNode, srChild: TSearchRec;
  Node: TTreeNode;
  SearchMode: LongInt;
  ListItem: TListItem;
  searchMask: string;
begin
  if not IsinTListView then
    path:= GetPath(ParentNode)
  else
  begin
```

```
  if ShowMode = 1 then
  begin
    oldpath:= path;
    ListView1.Clear;
  end;
  path: = GetPath_1 (DirName, oldpath);
end;
if ShowMode = 2 then
  SearchMode:= faAnyFile
else
  SearchMode:= faDirectory;
if FindFirst(path + '*', SearchMode, srNode) = 0 then
repeat
if ((srNode.Attr and faDirectory <> 0) and
          Real_Directory(srNode.Name)) or
    ((srNode.Attr and faDirectory = 0) and
                        (ShowMode = 2)) then
begin
  if ShowMode <> 2 then
  if not IsinTListView then
    Node:= TreeView1.Items.AddChild(ParentNode,
           SysToUTF8(srNode.Name));
  case ShowMode of
  0: begin
       Node.ImageIndex:=0;
       Node.SelectedIndex:= 0;
     end;
  1: begin
       ListItem:= ListView1.Items.Add;
```

```
      ListItem.Caption:= SysToUTF8(srNode.Name);
      ListItem.ImageIndex:= 0
    end;
2: якщо srNode.Attr and faDirectory = 0 then
    begin
      if srNode.Attr and faHidden = 0 then
      begin
        ListItem:= ListView1.Items.Add;
        ListItem.Caption:= SysToUTF8(srNode.Name);;
        ListItem.ImageIndex:= 1;
      end;
    end;
end;
if not IsinTListView then
begin
  {$IFDEF WINDOWS}
    searchMask:= path + srNode.Name + '\*';
  {$ELSE}
    searchMask:= path + srNode.Name + '/*';
  {$ENDIF}
  if FindFirst(searchMask, faDirectory, srChild) = 0
  then
  repeat
    if (srChild.Attr and faDirectory <> 0) and
              Real_Directory(srChild.Name)
    then Node.HasChildren:= true;
  until (FindNext(srChild) <> 0) або Node.HasChildren;
  SysUtils.FindClose(srChild);
end;
```

```
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.ListView1DblClick(Sender: TObject);
var
  s: string;
  AProcess: TProcess;
  srNode: TSearchRec;
  DirName: string;
  IsinTListView: boolean;
begin
  if ListView1.Selected = nil then exit;
  {$IFDEF WINDOWS}
  s:= path + UTF8ToSys(ListView1.Selected.Caption);
  {$ELSE}
  s:= path + ListView1.Selected.Caption;
  {$ENDIF}
  if FindFirst(s, faAnyFile, srNode) = 0 then
  if (srNode.Attr and faDirectory) <> 0 then
  begin
    DirName:= ListView1.Selected.Caption;
    IsinTListView:= true;
    ShowExpandNode(nil, DirName, 1, IsinTListView);
    ShowExpandNode(nil, DirName, 2, IsinTListView);
    SysUtils.FindClose(srNode);
    exit;
  end;
```

```
  if (srNode.Attr and faHidden) = 0 then
  begin
    AProcess:= TProcess.Create(nil);
    {$IFDEF UNIX}
      if MessageDlg('Відкривати в терміналі?', mtCustom,
         [mbYes, mbNO], 0) = mrYes then
      s:= 'xterm -T ''Lazarus Run Output''' +
          '-e $(TargetCmdLine)' + s;
    {$ENDIF}
    AProcess.CommandLine:= s;
    AProcess.Execute;
    AProcess.Free;
  end;
end;

procedure TForm1.TreeView1Change(Sender: TObject;
                                 Node: TTreeNode);
var
  IsinTListView: boolean;
begin
  if Node = nil then exit;
  TreeView1.BeginUpdate;
  ListView1.BeginUpdate;
  ListView1.Clear;
  Node.DeleteChildren;
  IsinTListView:= false;
  ShowExpandNode(Node, '', 1, IsinTListView);
  ShowExpandNode(Node, '', 2, IsinTListView);
  Node.Expanded:= true;
```

```
  ListView1.EndUpdate;
  TreeView1.EndUpdate;
end;

procedure TForm1.TreeView1Expanding(Sender: TObject;
      Node: TTreeNode; var AllowExpansion: Boolean);
var
  IsinTListView: boolean;
begin
  TreeView1.Items.BeginUpdate;
  Node.DeleteChildren;IsinTListV
  iew:= false;
  ShowExpandNode(Node, '', 0, IsinTListView);
  TreeView1.Items.EndUpdate;
end;
initialization
  {$I unit1.lrs}
end.
```

Подальше наповнення функціональністю нашого Провідника предо-
тавляю вам, шановний читачу. Зокрема, спробуйте зробити наступне – при подвійному натисканні на який-небудь файл необхідно запустити пов'язану з цим файлом програму. Наприклад, при подвійному клацанні на файл *.doc повинен запускатися текстовий редактор Word і відкривати цей файл.

Ну і всі ті функції, якими повинен мати будь-який файловий менед-
жер, тобто. реалізувати операції копіювання, переміщення та видалення файлів та каталогів та ін.

Можете "замахнутися" створення файлового менеджера типу TotalCommander, Krusader чи Double Commander (який, до речі, розробляється на Lazarus). Справа не в тому, що ви будете "винаходити велосипед".

Це буде чудовим тренуванням для вас!

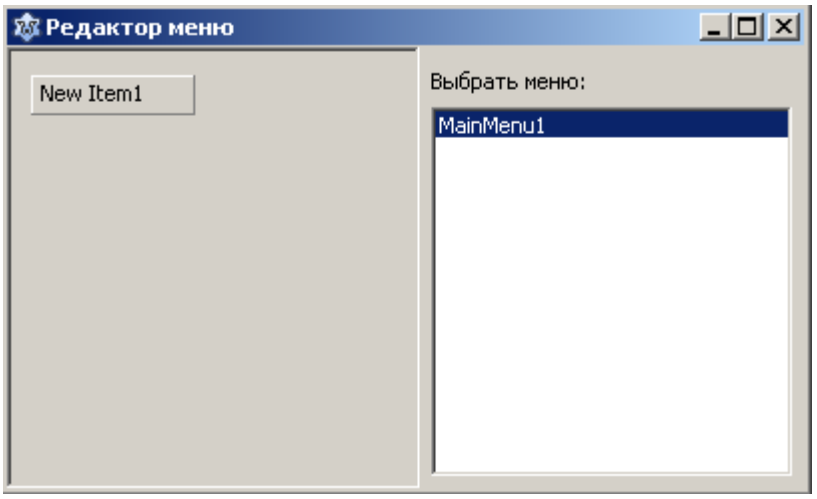
А я з вашого дозволу перейду до розгляду інших питань.

6.3.11 Організація меню. Механізм дій - Actions

6.3.11.1. Компонент TMainMenu

Розглянемо способи створення та організації меню. У меню користувач вибирає дані, а можливі дії. Таким чином, меню це спеціальний механізм, що дозволяє користувачеві всередині вашої програми вибрати якісь дії, наприклад відкрити файл, зберегти файл, встановити якісь параметри і т.д.

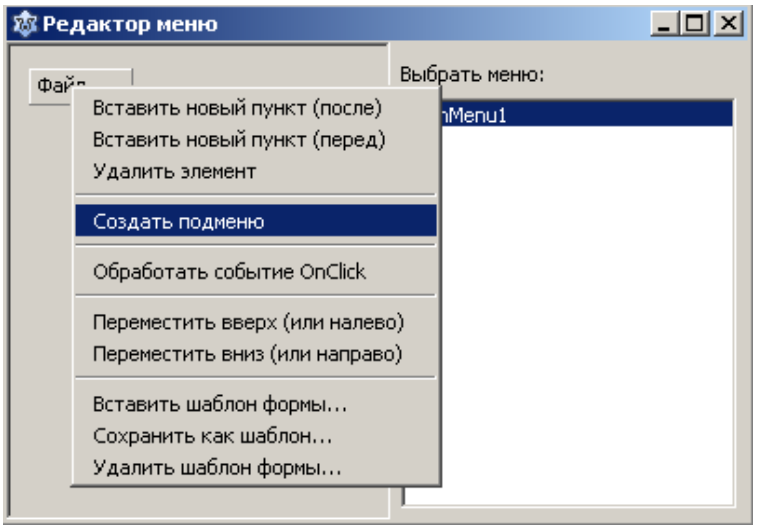
Для створення головного меню використовується компонент TMainMenu (вкладка Standard). Компонент не є візуальним. Пункти меню містяться у властивості Items. Щоб почати формування пунктів меню, достатньо двічі клацнути по компоненту на формі або натиснути на кнопку з трьома крапками у властивості Items компонента в інспекторі об'єктів. Відкриється спеціальний редактор меню, рис. 6.82.



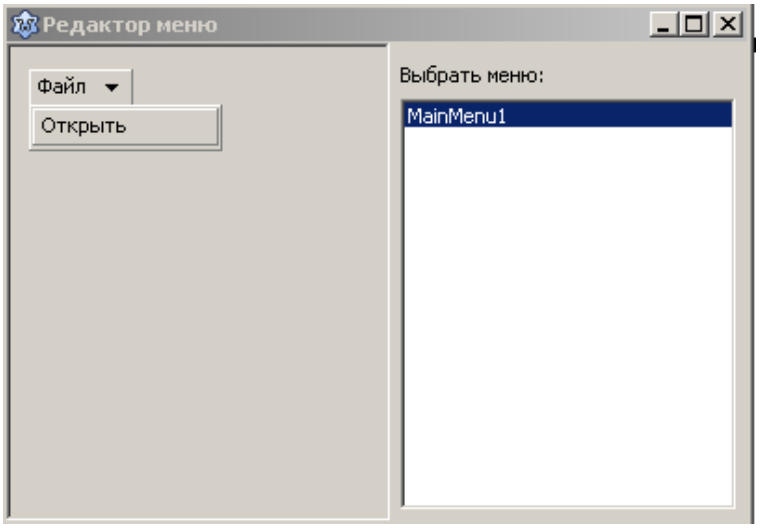
Мал. 6.82. Редактор меню

В інспекторі об'єктів у властивості Caption введіть ім'я пункту меню. Щоб створити наступний пункт меню або підменю, встановіть курсор на те-

кущий елемент меню та натисніть на праву клавішу миші, рис. 6.83. Виберіть потрібну дію, наприклад, для створення підменю натисніть "Створити підміню", мал. 6.84.

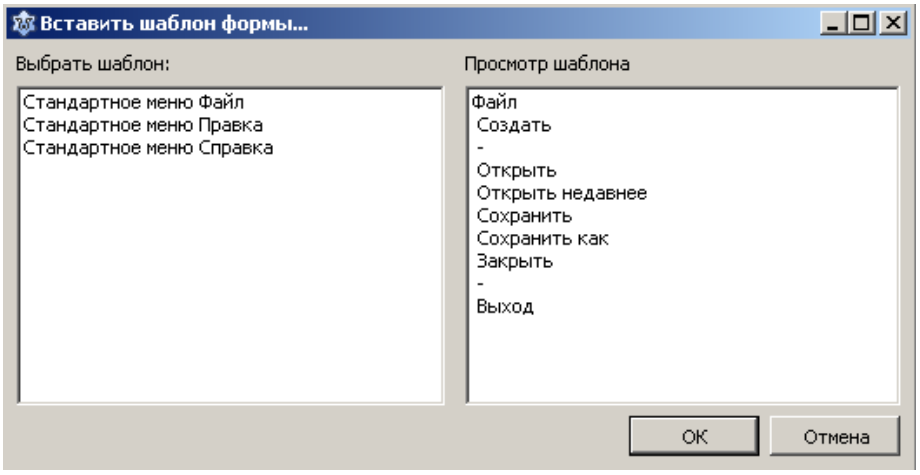


Мал. 6.83. Контекстне меню редактора



Мал. 6.84. Приклад створення підменю

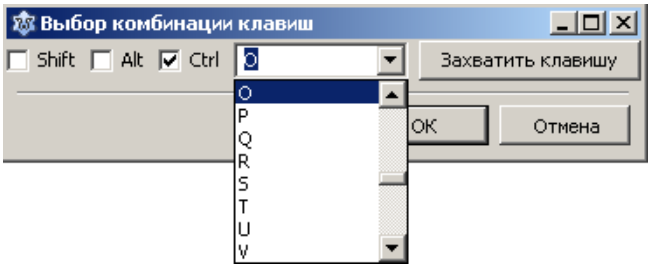
Lazarus надає можливість при створенні меню застосувати шаблони меню. Є три стандартні шаблони для меню "Файл", "Правка" та "Довідка", рис. 6.85.



Мал. 6.85. Шаблони меню

При створенні меню можна вставити поруч із текстом пункту меню піктограми. Для цього помістіть на форму компонент TImageList, заповніть його відповідними малюнками. Потім у TMainMenu у властивості Images вкажіть ім'я TImageList у програмі. А при створенні підпункту меню у властивості ImageIndex вкажіть індекс відповідного зображення.

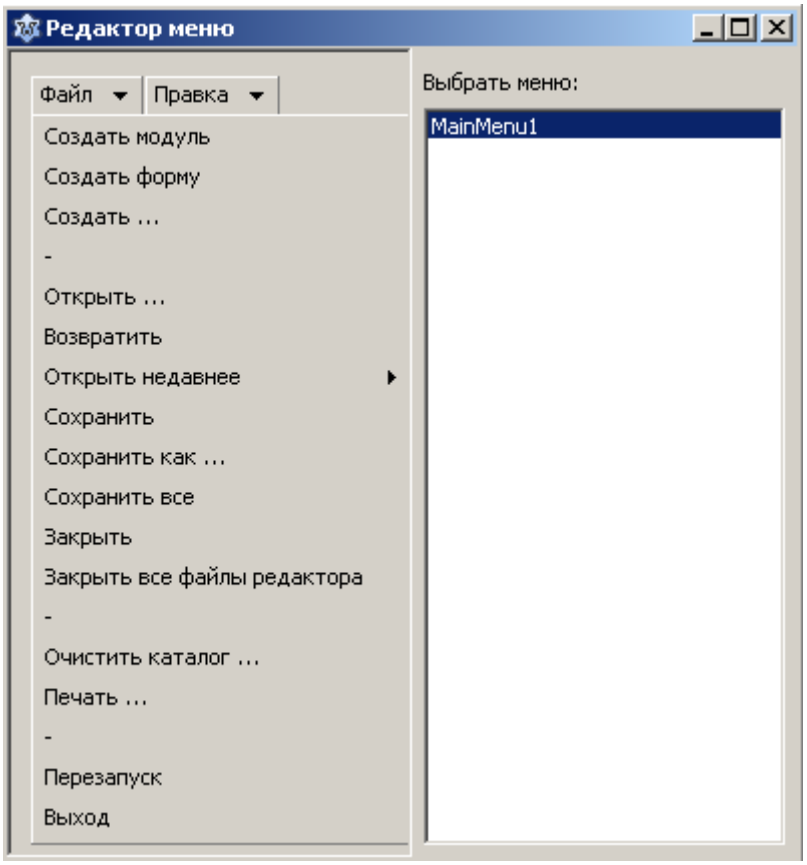
Також можна додати гарячі клавіші для цього пункту меню. Для цього є властивість Shortcut. Ви можете прямо ввести потрібне поєднання клавіш у поле введення цієї властивості або натиснувши на кнопку з трьома крапками, вибрати потрібне поєднання з редактора вибору сполучення клавіш, рис. 6.86.



Мал. 6.86. Встановлення клавіш.

Для створення розділової лінії між пунктами меню достатньо створити новий елемент та у властивість Caption ввести знак "-".

Спробуйте продати пункт меню "Файл" Lazarus. Малюнки можна знайти в папці установки Lazarus (підпапка Images), рис. 6.87.



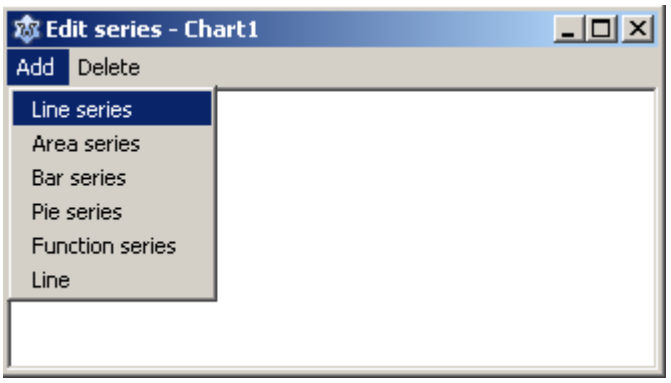
Мал. 6.87. Приклад створення меню

При цьому замість безликих імен пунктів меню (у програмі) типу MenuItem1, MenuItem2 і т.д. бажано надавати їм осмислені імена, наприклад MCreateModule, MCreateForm.

Для того, щоб реалізувати дію пункту меню, необхідно написати відповідний обробник події OnClick. Перебуваючи в редакторі меню, достатньо двічі клацнути по цьому пункту.

Ще з першого розділу за мною тягнеться один боржок. У 1.3.3. ми з вами розглядали метод найменших квадратів, а програму ми так і не написали. Але, як кажуть, усьому свій час. І ось цей час настав. Приступимо до цього методу.

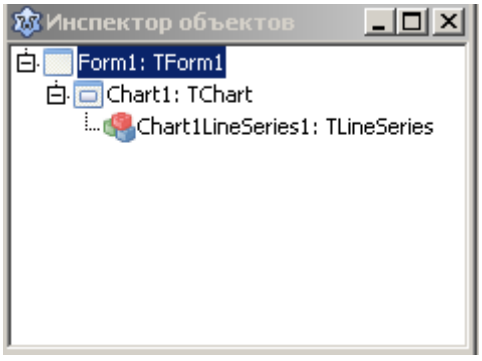
Але спочатку коротко розглянемо компонент TChart, який ми будемо використовувати у програмі для виведення графіків. Основною властивістю компонента є Series – набори даних (серії), на основі яких будуються графіки або діаграми. Якщо двічі клацнути по компоненту, розміщеному на формі, ми потрапимо до редактора серій, рис. 6.88.



Мал. 6.88. Редактор серій

Існують різні типи Series для побудови різних типів графіків або діаграм, наприклад, Line series – для побудови ліній (графіків), Pie series – для побудови кругових діаграм тощо. Виберіть тип серії Line series. В інспекторі об'єктів з'явиться ChartLineSeries1 типу TLineSeries.

Мал. 6.89. Створення об'єкта типу TLineSeries



Створити програмно-об'єкт типу TLineSeries можна оператором

```
Chart1LineSeries1:= TLineSeries.Create(Chart1);
```

Для додавання серії до TChart існує метод

```
procedure AddSeries(ASeries: TBasicChartSeries);
```

Додавання точок у об'єкт типу TLineSeries здійснюється за допомогою функції

```
function AddXY(X, Y: Double): Integer;
```

Властивість ShowPoints: boolean дозволяє показувати або не показувати крапки на графіку.

Властивість SeriesColor: TColor дозволяє вказати колір лінії.

Щоб вивести графік функції, наприклад, sin(x) можна написати таку процедуру:

```
procedure Plot_sin;
var
  i, n: integer;
  Chart1LineSeries1: TLineSeries;
begin
  n: = 100;
  Chart1LineSeries1:= TLineSeries.Create(Chart1);
  Chart1LineSeries1.SeriesColor:= clRed;
  Chart1LineSeries1.ShowPoints:= false;
  Chart1.AddSeries(Chart1LineSeries1);
  Chart1.Title.Visible:= true;
  Chart1.Title.Text.Text:= 'Графік функції sin(x)';
  for i:= 0 to n - 1 do
    Chart1LineSeries1.AddXY(i*Pi*0.02,
```

```
                                    sin(i*Pi*0.02));
end;
```

Отже, почнемо реалізацію методу найменших квадратів. Створіть новий проект, помістіть на форму компоненти TMainMenu, TOpenDialog та TChart, як показано на рис. 6.90. Меню програми має складатися з наступних пунктів:

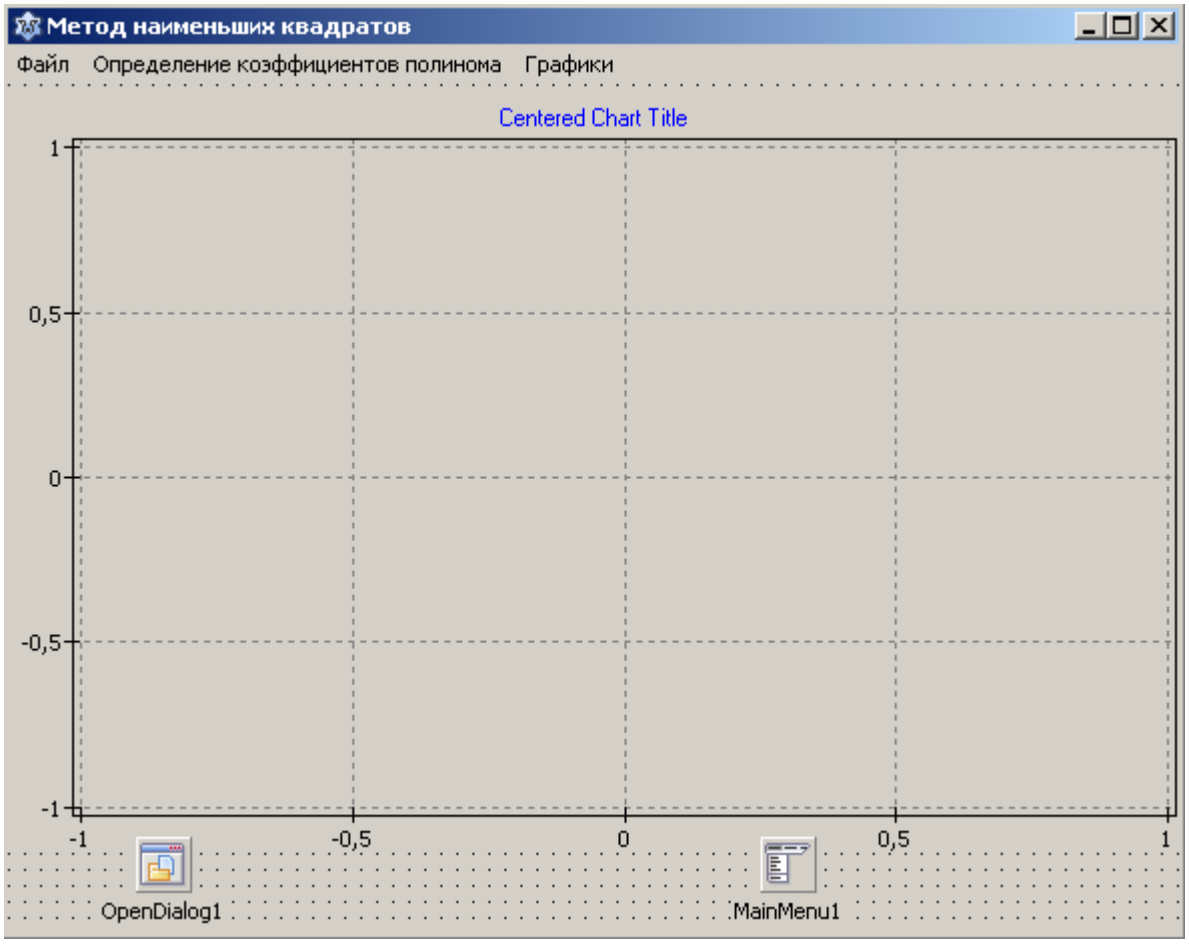
- Файл
- Визначення коефіцієнтів полінома•

Графіки

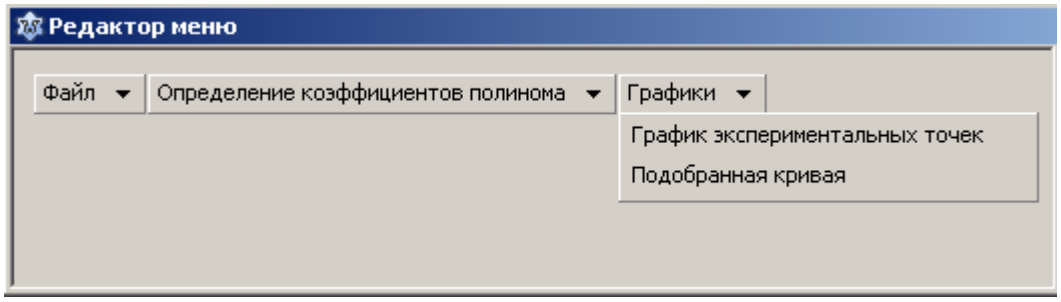
У свою чергу, меню "Файл" має складатися з підпунктів "Відкрити" та "Вихід". Меню "Визначення коефіцієнтів полінома" має містити пункт "Обчислити". І меню "Графіки" має містити пункти "Графік експериментальних точок" та "Піддобрена крива", рис.6.91.

Надайте імена меню в програмі як показано на рис. 6.92.

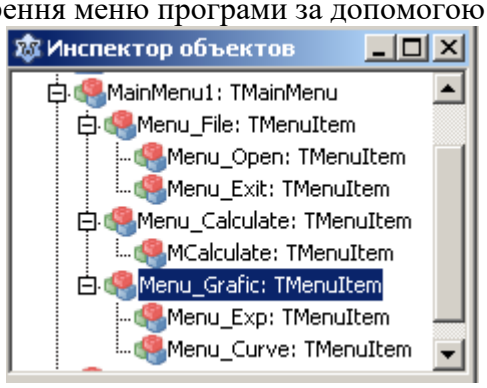
Вставимо в пункти меню поруч із текстами піктограми. Для цього помістіть на форму компонент TImageList і заповніть його відповідними значками. У якості Images MainMenu1 вкажіть ім'я TImageList. Потім для кожного пункту меню вкажіть відповідні індекси у властивостях ImageIndex.



Мал. 6.90. Форма застосування



Мал. 6.91. Створення меню програми за допомогою редактора меню



Мал. 6.92. Імена меню у програмі

Код програми:

```
unit unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   LResources,   Forms,   Controls,
  Graphics,   Dialogs,   ExtCtrls,   TAGraph,   TASeries,
  Buttons, StdCtrls, Menus, FileUtil;
type
  {TForm1}
  TForm1 = class(TForm)
    Chart1: TChart;
    MainMenu1: TMainMenu;
    MCalculate: TMenuItem;
    Menu_File: TMenuItem;
    Menu_Calculate: TMenuItem;
    Menu_Grafic: TMenuItem;
    Menu_Open: TMenuItem;
    Menu_Exit: TMenuItem;
    Menu_Exp: TMenuItem;
    Menu_Curve: TMenuItem;
    OpenDialog1: TOpenDialog;
    procedure FormCreate(Sender: TObject);
    procedure Menu_ExpClick(Sender: TObject);
    procedure Menu_CurveClick(Sender: TObject);
    procedure Menu_OpenClick(Sender: TObject);
    procedure MCalculateClick(Sender: TObject);

  private
    {private declarations}
```

```
  public
    { public declarations }
  end;

// Процедура рішення СЛАУ методом Гауса
procedure gauss(vector: array of real; b: array of real;
                var x: array of real; n: byte;
                var solve: byte);
// n - розмірність системи,
// solve = 0, якщо рішення єдине,
// solve = 1, якщо система не має рішення,
// solve=2, якщо система має безліч рішень,

// Функція, що підбирається методом
// найменших квадратів
функція fx(t: real): real;

// Функція зведення у ступінь
function stepen(x: real; n: byte): real;

var
  Form1: TForm1;
  n: byte;
  x1, y1: real;
  x, y, z: array of real;
  Fname: string;

implementation
функція fx(t: real): real;
```

```
begin
  Result:= z[0] + z[1]*t + z[2]*t*t +
           z[3]*t*t*t + z[4]*sqr(sqr(t));
end;

function stepen(x: real; n: byte): real;
var
  i: integer;
begin
  Result:= 1;
  for i:= 1 to n do
  Result:= Result*x;
end;
// Реалізація методу Гауса
procedure Gauss(vector: array of real; b: array of real;
                var x: array of real; n: byte;
                var solve: byte);
var
 a: array of array of real; { матриця коефіцієнтів системи,
 двовимірний динамічний масив}
 i, j, k, p, r: integer;
 m, s, t: real;
begin
 SetLength(a, n, n); // Встановлення фактичного розміру
 масиву

 { Перетворення одновимірного масиву на
 двовимірний } k:=0;
 for i:=0 to n-1 do
   for j:=0 to n-1 do
```

```
  begin
    a [i, j]: = vector [k];
    k:=k+1;
  end;
for k:=0 n-2 do
begin
  for i:=k+1 n-1 do
  begin
    if (a[k, k]=0) then
    begin
      {перестановка рівнянь}
      p:=k; // В алгоритмі використовується буква l, але
      вона схожа на 1
            // Тому використовуємо ідентифікатор p
      для r:=i до n-1 до
      початку
        if abs(a[r, k]) > abs(a[p, k]) then p:=r;
      end;
      if p<>k then
      begin
        for j:= до n-1 до
        початку
          t: = a [k, j];
          a [k, j]: = a [p,
          j]; a [p, j]: =
          t;
        end; t: = b
        [k];
        b[k]:=b[p];
        b[p]:=t;
```

```
    end;
```

```
    end; // кінець блоку перестановки рівнянь
    m: = a [i, k] / a
    [k, k]; a[i,
    k]:=0;
    for j:=k+1 n-1 do begin
      a [i, j]: = a [i, j]-m * a
    [k, j]; end;
    b[i]:= b[i]-m*b[k];
  end;
end;
{Перевірка існування
рішення} if a[n-1,n-1] <> 0
then begin
  x[n-1]:=b[n-1]/a[n-1,n-
  1];for i:=n-2 downto 0 do
  begin
    s:=0;
    for j:=i+1 n-1 do begin
      s:=sa[i, j]*x[j];
    end;
    x[i]:=(b[i] + s)/a[i, i];
  end;
  solve: =
0; end
else
if b[n-1] = 0 then
begin
  MessageDlg('Система має нескінченне' +
```

```
            'кількість рішень', mtInformation, [mbOK], 0);
   solve: = 2;
 end
 else
 begin
   MessageDlg('Система не має рішень',
               mtInformation, [mbOK], 0);
   solve:= 1;
 end;
 { звільнення пам'яті,
```
розподіленої для динамічного масиву a:=nil;
```
end;

{TForm1}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Chart1.Title.Text.Text:='Метод найменших квадратів';
  MCalculate.Enabled:= false;
  Menu_Exp.Enabled:= false;
  Menu_Curve.Enabled:= false;
end;

procedure TForm1.Menu_ExpClick(Sender: TObject);
{Процедура виведення графіка
 експериментальних даних за точками}
var
  i: integer;
```

```
  gr1: TLineSeries;
begin
  Chart1.Visible: = true;
  gr1:= TLineSeries.Create(Chart1);
  Chart1.AddSeries(gr1);
  для i:= 0 до n - 1 do
  gr1.AddXY(x[i], y[i]);
end;

procedure TForm1.Menu_CurveClick(Sender: TObject);
{Процедура виведення суміщених
 графіків експериментальних даних
 за точками та підібраною методом
 найменших квадратів кривою, що
 найкраще наближається до
 експериментальних даних}
var
  i: integer;
  gr1, gr2: TLineSeries;
begin
  Chart1.Visible: = true;
  gr1:= TLineSeries.Create(Chart1);
  gr1.ShowPoints: = true; // графік з точками
  gr1.ShowLines: = false; // не з'єднувати крапки
  лініями Chart1.AddSeries(gr1);
  gr2:= TLineSeries.Create(Chart1);
  gr2.ShowLines := true;
  Chart1.AddSeries(gr2);
  для i:= 0 до n - 1 do
  gr1.AddXY(x[i], y[i]);
```

```
  for i:= 0 to n - 1 do
  gr2.AddXY(x[i],
  fx(x[i])));
end;

procedure TForm1.Menu_OpenClick(Sender: TObject);
// процедура вибору, відкриття та читання файлу даних
var
  f: TextFile;
  i: integer;
begin
  if OpenDialog1.Execute then
    Fname:= OpenDialog1.FileName
  else exit;
  Fname:= UTF8ToSys(Fname); //перетворення на системне кодування
  AssignFile(f, Fname);
  Reset(f);
  // відключення контролю помилок введення/виводу
  {$I-}
  // Читання кількості експериментальних точок
  Readln(f, n);
  if IOResult <> 0 then
  begin
    ShowMessage('Помилка при читанні з
    файлу!'); exit;
  end;
  // Розподіл пам'яті під динамічні масиви
  SetLength(x, n);
  SetLength(y, n);
  for i:= 0 to n - 1 do
```

```
  begin
    read(f, x[i]);
    if IOResult <> 0 then
    begin
      ShowMessage('Помилка при читанні з
      файлу!'); exit;
    end;
  end;
  for i:= 0 to n - 1 do
  begin
    read(f, y[i]);
    if IOResult <> 0 then
    begin
      ShowMessage('Помилка при читанні з
      файлу!'); exit;
    end;
  end;
  {$I+}
  CloseFile(f);MCalculate.En
  abled:= true;
end;
procedure TForm1.MCalculateClick(Sender: TObject);
var
  i, j, k, l: integer;
  b, vector: array of real;
  s: real;
  solve: byte;
begin
  SetLength(z, 5);
```

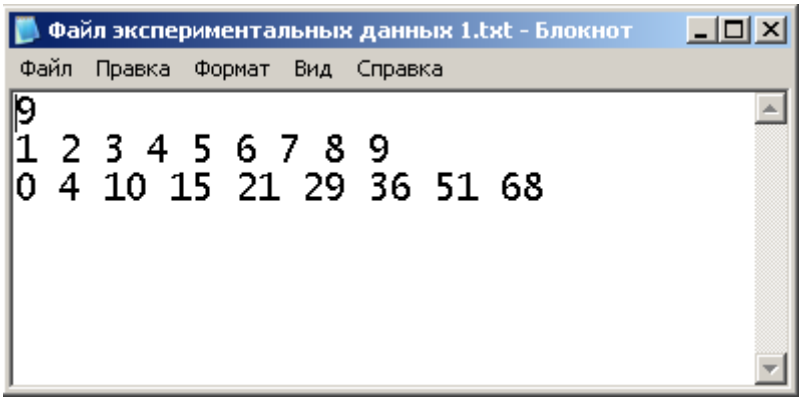
```
  SetLength(b, 5);
  SetLength(vector, 25);
  j: = 0;
  for k:= 0 to 4 do
  for l:= 0 to 4 do
  begin
    s: = 0;
    for i:= 0 to n - 1 do
    s: = s + stepen (x [i], k + l);
    vector[j]:= s;
    j:= j+1;
  end;
  for k:= 0 to 4 do
  begin
    s: = 0;
    for i:= 0 to n - 1 do
    s:= s + y[i]*stepen(x[i], k); b
    [k]: = s;
  end;
  // Рішення СЛАУ
  gauss(vector, b, z, 5, solve);
  if solve = 0 then
  begin
    Menu_Exp.Enabled:=true;
    Menu_Curve.Enabled:=true;
  end;
end;
initialization
  {$I unit1.lrs}
```

```
end.
```

Для того щоб зрозуміти програму згадайте реалізацію методу Гауса рішення системи лінійних рівнянь алгебри, причому у варіанті із застосуванням динамічних масивів. У тій програмі матрицю коефіцієнтів системи ми спочатку перетворювали на одновимірний динамічний масив, оскільки передавати як формальний параметр у функцію або процедуру можна лише одновимірний динамічний масив.

Тому у програмі методу найменших квадратів у процедурі MCalculateClick формується одновимірний динамічний масив vector, який насправді є матрицею коефіцієнтів системи. Його ми і передаємо (разом із вектором вільних членів) у процедуру Gauss.

Перед запуском програми підготуйте у будь-якому текстовому редакторі файл наступного виду:



Мал. 6.93. Файл даних

Файл має таку структуру. Перший рядок містить кількість точок. Другий рядок містить значення Zi, третій рядок містить значення Pi.

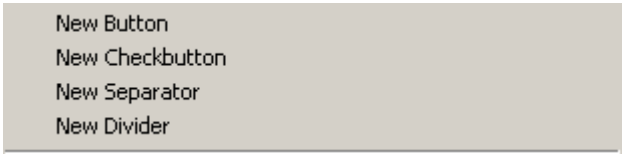
Зверніть увагу, що до завантаження файлу з експериментальними даними пункти меню "Обчислити", "Графік експериментальних точок" і "Піддобрена крива" необхідно зробити недоступними. Після завантаження

файлу, пункт меню "Обчислити" треба зробити доступним, а пункти "Графік експериментальних точок" та "Піддобрена крива", як і раніше, недоступні. Тільки після того, як будуть виконані необхідні обчислення, пункти "Графік експериментальних точок" та "Піддобрена крива" стають доступними.

6.3.11.2. Компонент TToolBar

Якщо у вашій програмі є досить велике та розгалужене мене, то майже завжди має сенс додати до програми так звані кнопки швидкого доступу. Ці кнопки будуть дублювати найчастіше використовувані пункти меню. Наприклад, якщо у програмі відкривається багато файлів, то однією з зручностей, які надає користувачеві, буде наявність кнопки для швидкого доступу до цієї операції. У великих програмах зазвичай потрібно розмістити кілька кнопок швидкого доступу. Однак якщо вони розкидані по різних місцях, це ускладнює роботу користувача. Найчастіше кнопки швидкого доступу поєднують у вигляді інструментальної панелі.

Компонент TToolBar являє собою панель для розміщення кнопок швидкого доступу. У принципі цей компонент можна вставляти будь-які компоненти, але оскільки він призначений, перш за все, для створення інструментальних панелей, в нього і поміщають кнопки. Причому TToolBar містить у собі спеціальний компонент TToolButton – інструментальна кнопка. Щоб помістити інструментальну кнопку в TToolBar, натисніть праву клавішу миші і виберіть NewButton, мал. 6.94.



Мал. 6.94. Додавання нової кнопки

Зображення на кнопці визначається властивістю ImageIndex і визначає індекс зображення в компоненті TImageList. У момент виконання програми деякі кнопки на інструментальній панелі з тих чи інших причин можуть бути недоступні, наприклад, до виконання певних умов. Програміст може вказати інші зображення для цих кнопок. Для цього потрібно помістити на форму ще один компонент TImageList і власне DisabledImages вказати його ім'я. Так само властивість HotImages дозволяє вказати контейнер зображень для кнопок, коли над ними проходить вказівник миші. На практиці, найчастіше, воліють все ж таки мати один контейнер зображень для всіх випадків.

New Checkbutton задає різновид кнопки, який після натискання на неї залишається в натиснутому стані. Це так звана кнопка, що западає. Щоб віджати її, необхідно натиснути на неї ще раз.

New Separator задає роздільник (сепаратор). Дозволяє розділити групи кнопок за функціональним призначенням. Під час виконання програми на цьому місці з'являються дві вертикальні лінії.

New Divider теж роздільник, але в момент виконання з'являється одна вертикальна лінія.

Цим чотирма різновидами кнопок відповідають наступні значення властивості Style:

- tbsButton – звичайна інструментальна кнопка;
- tbsCheck - кнопка, що западає;
- tbsSeparator - подвійний роздільник;
- tbsDivider – одинарний роздільник.

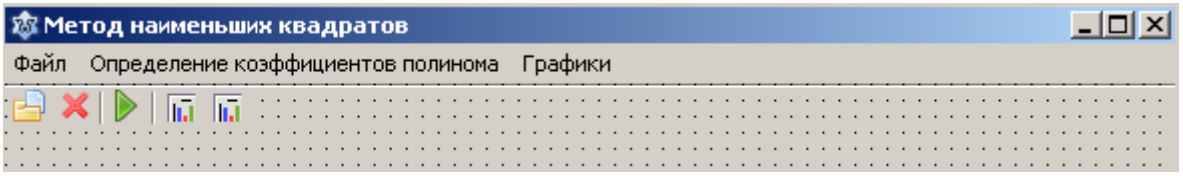
Крім того, властивість Style має ще одне значення tbsDropDown. При натисканні на кнопку з'являється меню, що випадає. Але для цього треба в якості MenuItem вказати ім'я відповідного пункту меню. Якщо при проектуванні цього пункту меню було встановлено такі властивості, як

ImageIndex, Hint, Enabled, Visible, всі ці властивості автоматично присвояться і цій кнопці.

Властивість Caption дозволяє вивести поруч із кнопкою деякий текст, при цьому текст буде видно, якщо властивість ShowCaptions компонента TToolBar встановлена в true. Зазвичай в інструментальних панелях тексти кнопок не показують, але тоді важливого значення набуває властивість Hint (підказка). Адже навіть якщо малюнок кнопки є досить інформативним, користувачеві іноді буває важко визначити, для чого призначена ця кнопка. Властивість Hint: string дозволяє вивести підказку під час наведення на кнопку покажчика миші. Підказка буде показана за умови, що властивість ShowHint = true.

Властивість Wrap дозволяє керувати розміщенням кнопок у декілька рядів. Якщо вам необхідно, щоб після поточної кнопки наступна кнопка розташовувалась у новому ряду, вам потрібно встановити властивість Wrap поточної кнопки в true.

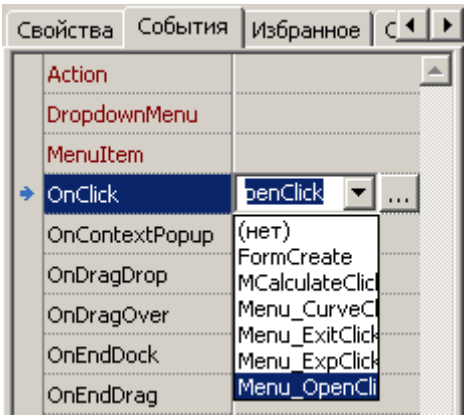
Давайте доопрацюємо попередню програму. Помістіть на форму компонент і спроектуйте вид інструментальної панелі, як показано на рис. 6.95.



Мал. 6.95. Вигляд інструментальної панелі програми

Дуже важливо мати на увазі, що панель інструментів ми створили для того, щоб продублювати деякі пункти меню (у нашому випадку все). Обробники подій OnClick для цих пунктів вже існують і тому немає жодної необхідності писати нові обробники! Для кожної кнопки просто вкажіть імена готових обробників. Наприклад, виділіть першу інструментальну кнопку. Вона дублюватиме пункт меню Файл->Відкрити.

В інспекторі об'єктів у вкладці Події в рядку OnClick натисніть кнопку з трикутником і виберіть потрібний обробник, мал. 6.96.



Мал. 6.96. Вибір потрібного оброблювача

У коді програми відбудуться мінімальні зміни. В обробнику `OnCreate форми додайте оператори:`

```
TB_Calculate.Enabled:= false;
TB_Graf_Exp.Enabled:= false;
TB_Graf_Curve.Enabled:= false;
```

Щоб три останні кнопки були недоступні. У оброблювач `Menu_OpenClick після оператора`

```
MCalculate.Enabled:= true;
```

додайте оператор

```
TB_Calculate.Enabled:= true;
```

А в обробнику MCalculateClick після оператора

```
Menu_Curve.Enabled:=true;
```

додайте оператори

```
TB_Graf_Exp.Enabled:= true;
TB_Graf_Curve.Enabled:= true;
```

Ну і, зрозуміло, до класу форми будуть додані об'єкти класів `TToolBar` та `TToolButton`.

Подивіться, як ми мінімальними зусиллями реалізували такий складний елемент графічного інтерфейсу як панель інструментів.

6.3.11.3. Компонент TActionList

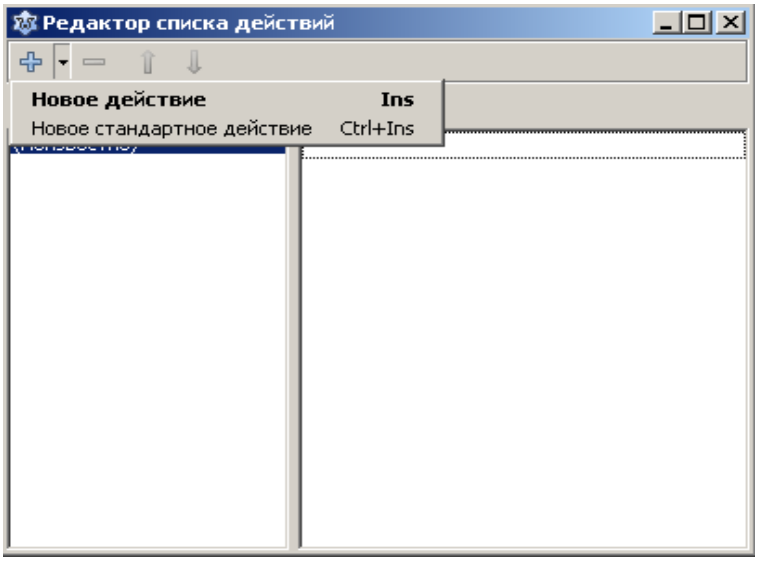
У попередній програмі ми призначали для кнопки інструментальної панелі той самий обробник, що й для відповідного пункту меню. Є інший, загальніший, метод уніфікації поведінкової сторони програми. Це так званий механізм дій Actions. Цей метод є кращим і рекомендованим розробки складних програм. Основна перевага методу – код програми стає прозорішим і структурованішим, а процес проектування та розробки програми більш стандартизованим і менш трудомістким. При цьому під дією розуміється організація реакції програми на якісь дії користувача, наприклад, натискання на кнопку на панелі інструментів. Чому ж ми говоримо дії, а чи не реакції. Тому що ми в програмі передбачаємо коло можливих дій користувача та організуємо відповідні реакції на ці дії. Отже, вихідним посилом є дію – Action. А на дії користувача, які не були передбачені, програма просто не реагуватиме. Наприклад, якщо ви не передбачили обробник для натискання якоїсь кнопки, то скільки б ви не натискали на цю кнопку, жодного результату не буде.

Механізм дій дає зручний засіб для організації централізованої реакції програми на ті чи інші дії користувача. Компонент

TActionList містить список назв дій та пов'язаних з ними імен обробників, які ці дії реалізують. Щоб зв'язати якийсь компонент з необхідною дією, достатньо розкрити в Інспекторі об'єктів список у властивості Action і вибрати потрібну назву. При цьому всі властивості цієї дії та обробник автоматично перенесуться в даний компонент, включаючи зображення, напис на компоненті, а також пов'язана з ним підказка.

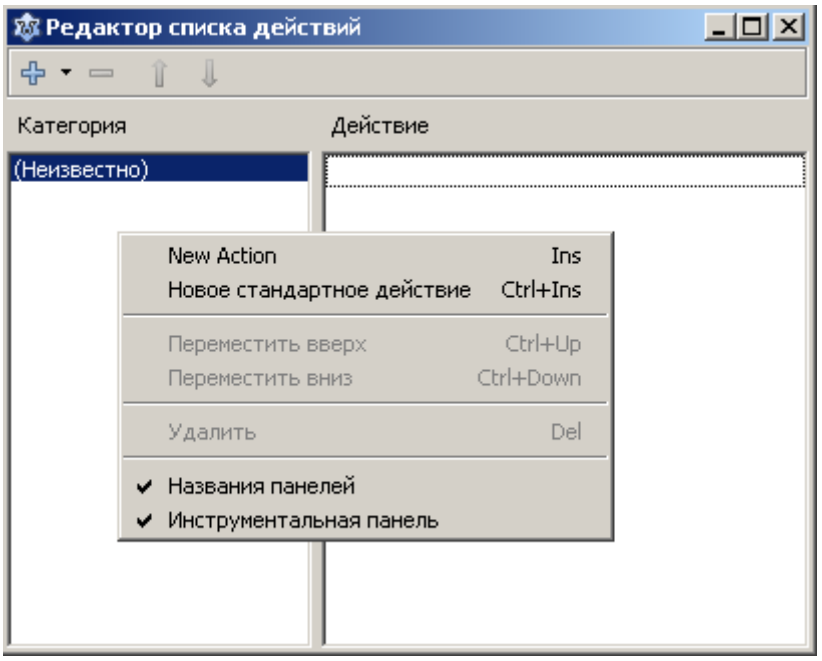
Помістіть на форму компоненти TActionList та TImageList. Занесіть у TImageList кілька піктограм. У якості Images компонента TActionList вкажіть ім'я компонента TImageList. Двічі клацніть на компоненті TActionList на формі. Відобразиться редактор списку дій. Для того щоб додати нову дію ви можете:

- Натиснути кнопку із зображенням знака плюс. У правому вікні редактора з'явиться нова дія зі стандартним ім'ям Action1;
- Натиснути на маленький трикутник поруч із кнопкою із зображенням знака плюс. З'явиться список, що розкривається, рис. 6.97 у якому ви можете вибрати "Нова дія" або "Нова стандартна дія";



Мал. 6.97. Редактор списку дій

- У редакторі списку дій клацніть правою кнопкою миші. З'явиться контекстне меню, рис. 6.98;

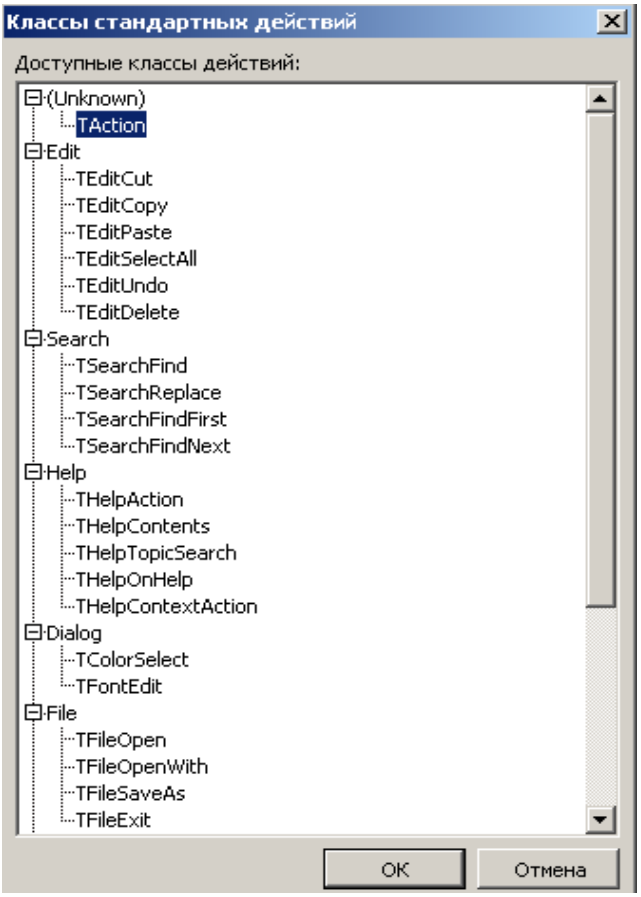


Мал. 6.98. Додавання нової дії

- І нарешті, ви можете використовувати клавішу Ins для введення нової дії або натиснути Ctrl+Ins для введення нової стандартної дії.

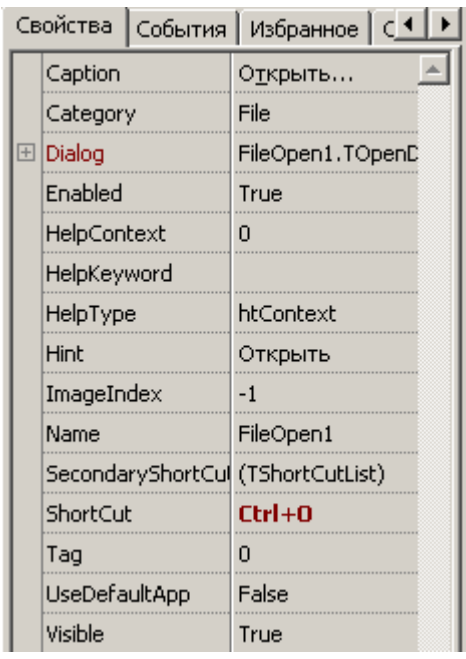
Тепер, що таке стандартна дія. Розробниками Lazarus створені спеціальні класи, що реалізують типові дії, що найчастіше зустрічаються, так що для них навіть не потрібно писати обробники! Якщо ви виберете "Нова стандартна дія", з'явиться вікно зі списком стандартних дій, мал. 6.99.

На малюнку ви бачите, що стандартні події розбиті на категорії. Там ви бачите імена відповідних класів. Решта дій (за термінологією Lazarus unknown – невідомі) ставляться до класу TAction. Але краще говорити просто дії чи нестандартні дії.

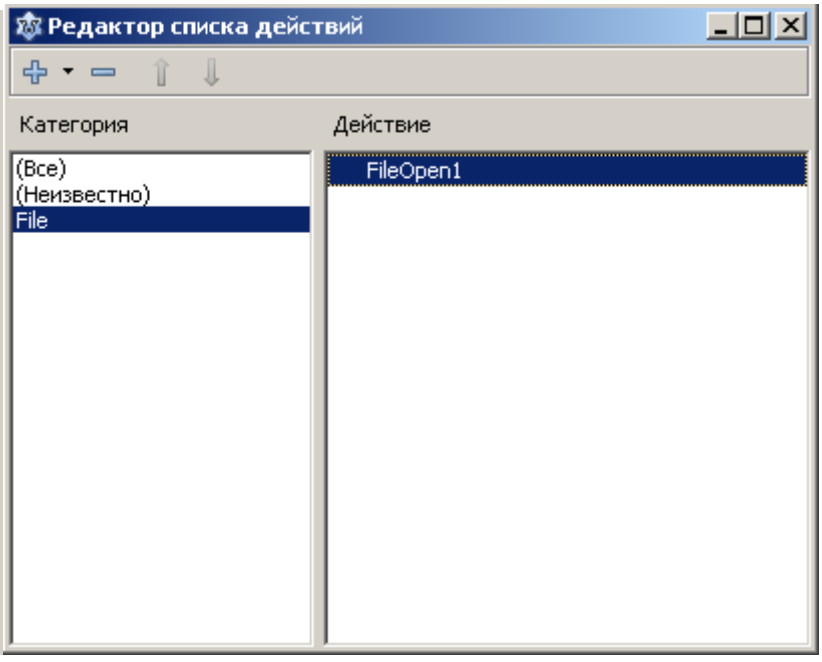


Мал. 6.99. Стандартні дії

Виберемо, наприклад, стандартну дію TFileOpen, рис. 6.100, 6.101.

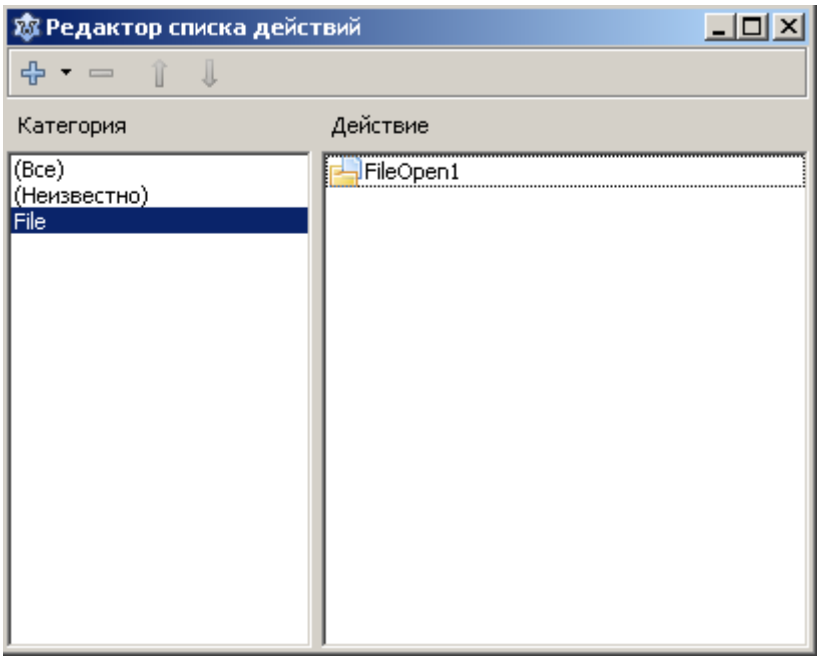


Мал. 6.100. Властивості



Мал. 6.101. Додавання стандартної дії

Ми бачимо, що в Інспекторі об'єктів автоматично заповнилися властивості Caption, Hint, ShortCut. Вкажіть необхідне значення властивості ImageIndex. У редакторі списку дій поруч із ім'ям дії відразу з'явиться відповідне зображення, рис. 6.102.



Мал. 6.102. Додавання піктограми

Розкривши властивості Dialog, ви можете налаштувати властивості стандартного діалогу відкриття файлу. При цьому зверніть увагу, ми на форму компонентів TOpenDialog не переносили! Він уже реалізований у класі TFileOpen.

При додаванні нестандартної дії вам необхідно буде заповнити властивості цієї дії, зокрема згадані вище властивості Caption, Hint, ShortCut. І обов'язково написати обробник OnExecute, який реалізує потрібну дію. Ця подія виникає, коли користувач ініціював дію, наприклад, натиснув кнопку. Окрім цієї події для кожної дії визначено ще дві події OnHint та OnUpdate. Подія OnHint виникає під час показу підказки, коли користувач затримав покажчик миші над інтерфейсним компонентом. Оброблювач визначено наступним чином:

```
procedure FileOpen1Hint(var HintStr: string;
                        var CanShow: Boolean);
```

Ви можете замінити текст підказки Hint, вказавши новий текст у параметрі `HintStr`.

Подія OnUpdate виникає, коли користувач нічого не робить. Ви можете використовувати цю подію для виконання підготовчих операцій.

На закінчення зазначимо, що у стандартних діях діалогів, які потребують вибору (категорії Search, Dialog, File) відсутні події OnExecute та OnUpdate. Замість них введені події BeforeExecute (виникає перед викликом діалогу), OnAccept (виникає, якщо користувач зробив вибір) та OnCancel (виникає, якщо користувач у діалозі вибір не зробив).

Наприклад, щоб отримати ім'я файлу, вибраного користувачем у стандартній дії FileOpen1, треба в обробнику OnAccept записати

```
Fname:= FileOpen1.Dialog.FileName;
```

Напишемо тепер програму методу найменших квадратів із застосуванням механізму дій, заодно визначимо порядок чи методику розробки таких додатків.

Насамперед, необхідно продумати та визначити список тих дій (на папері), які ми реалізовуватимемо. Записується це у вільній формі, щоб було зрозуміло самому собі.

Отже, ми знаємо, що експериментальні дані містяться у текстовому файлі. Тому перша дія очевидна. Необхідно надати користувачеві можливість у діалозі відкрити файл із експериментальними даними. Назвемо цю дію "Відкрити файл". Заодно відзначаємо для себе, що найпростіше для цього використовувати стандартну дію. Друга дія - викон-

нитку необхідні обчислення, згідно з алгоритмом методу найменших квадратів. Нехай ця дія називатиметься "Обчислення". Потім необхідно дати користувачеві можливість переглянути графік, побудований за заданими експериментальними точками. Назвемо цю дію "Побудова графіка за заданими експериментальними точками". І, нарешті, потрібно побудувати суміщений графік. Графік за експериментальними точками та графік подібної кривої для того, щоб користувач міг візуально оцінити точність отриманих результатів. Надамо цій дії назву "Побудова сумісного графіка". Передбачимо ще одну стандартну дію – вихід із програми. Після того, як список дій складений, вже більш-менш ясно, як виглядатиме головне меню нашої програми. Зазначаємо, які пункти меню слід продублювати в інструментальній панелі. Також обов'язково відзначте, які пункти меню та кнопки інструментальної панелі мають бути спочатку доступними, а які ні. І коли слід недоступні пункти та кнопки зробити доступними.

Звичайно, при розробці великих та складних програм неможливо все заздалегідь передбачити та врахувати. У міру просування роботи над програмою додаватимуться нові дії і, можливо, видалятимуться деякі старі. Але складання хоча б приблизного списку дій перед початком роботи необхідне. Це дозволить вам уявити загальну структуру вашої майбутньої програми і значно допоможе вам у подальшій роботі.

Отже, використовуючи складений список дій, у редакторі списку дій створіть такі дії, рис. 6.103.

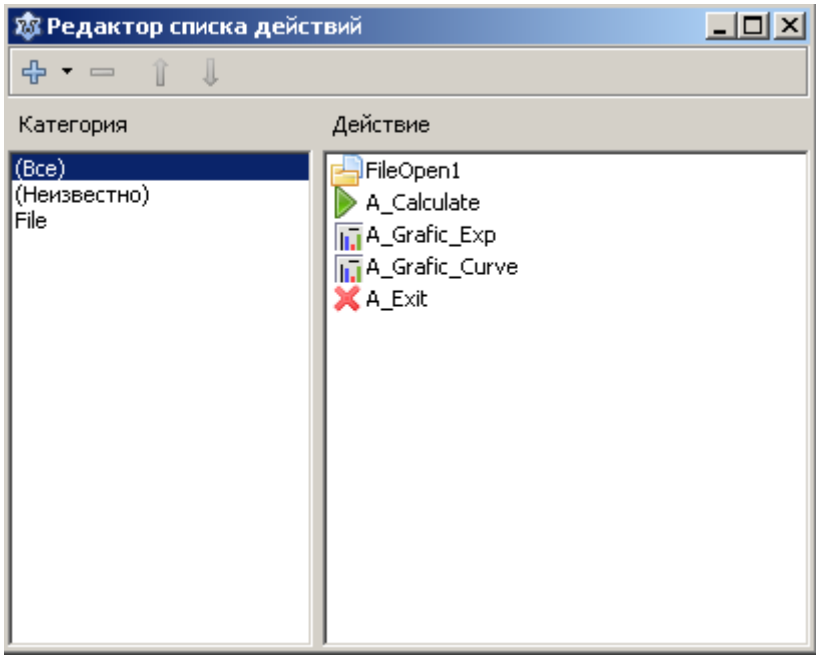
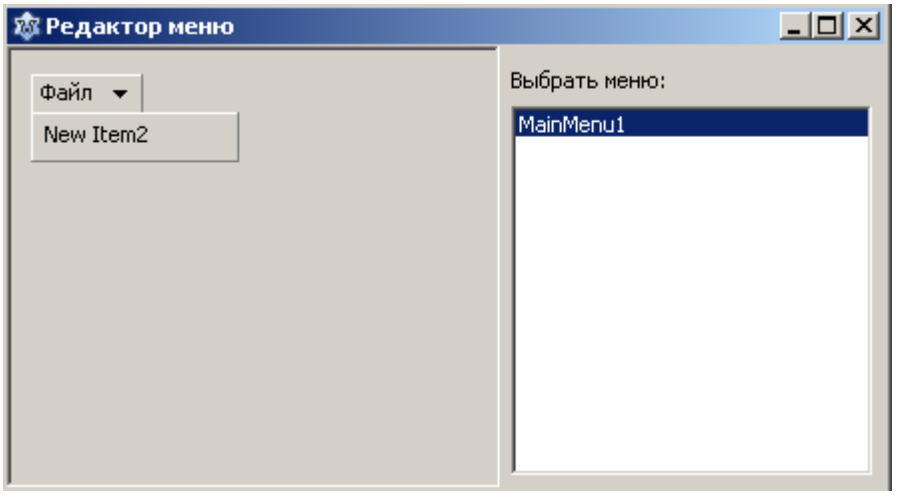


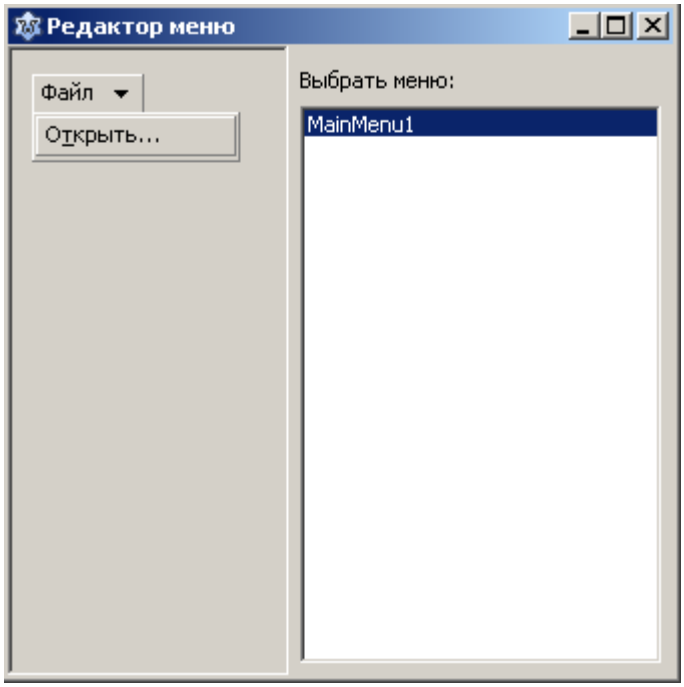
Рис.6.103. Список дій

Перенесіть на форму компонент TMainMenu. Створіть пункт меню "Файл". Дайте йому ім'я Menu_File. Створіть підпункт. Введіть ім'я Menu_Open. Інші властивості поки що не заповнюйте! Мал. 6.104.

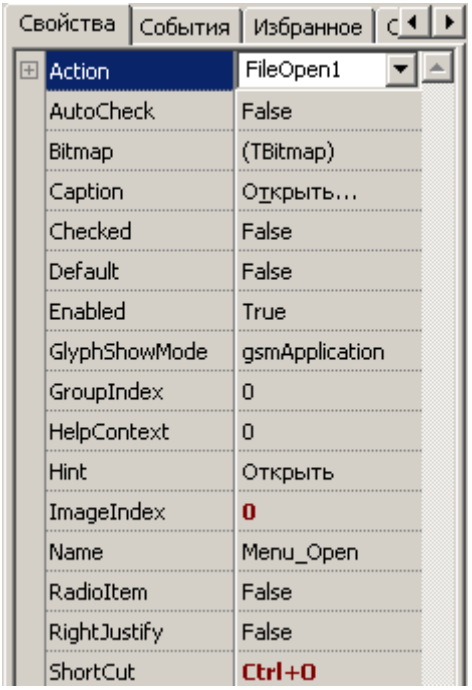


Мал. 6.104. Створення меню програми

Тепер у властивості Action розкрийте список і виберіть дію FileOpen1. Властивість Caption Menu_Open замість New Item2 автоматично змінилася на "Відкрити…", рис. 6.105, 6.106.



Мал. 6.105. Створення меню програми



Мал. 6.106. Властивості

Крім того, автоматично заповнилися властивості Hint, ImageIndex та ShortCut! Тепер нам залишилося написати обробник для дії FileOpen1. Оскільки ця стандартна дія - відкриття файлу, потрібно написати обробник події OnAccept.

```
procedure TForm1.FileOpen1Accept(Sender: TObject);
// процедура вибору, відкриття та читання файлу даних
var
  f: TextFile;
  i: integer;
  Fname: string;
begin
  Fname:= FileOpen1.Dialog.FileName;
  Fname:= UTF8ToSys(Fname); //перетворення на системне кодування
  AssignFile(f, Fname);
  Reset(f);
```

```
// відключення контролю помилок введення/виводу
{$I-}
// Читання кількості експериментальних точок
Readln(f, n);
if IOResult <> 0 then
begin
  ShowMessage('Помилка при читанні з
  файлу!'); exit;
end;
// Розподіл пам'яті під
масиви SetLength(x, n);
SetLength(y, n);
for i:= 0 to n - 1 do
begin
  read(f, x[i]);
  if IOResult <> 0 then
  begin
    ShowMessage('Помилка при читанні з
    файлу!'); exit;
  end;
end;
for i:= 0 to n - 1 do
begin
  read(f, y[i]);
  if IOResult <> 0 then
  begin
    ShowMessage('Помилка при читанні з
    файлу!'); exit;
  end;
```

```
  end;
  {$I+}
  CloseFile(f);A_Calculate.Enabl
  ed:= true;
end;
```

Обробник практично збігається з обробником Menu_OpenClick попередньої програми. Але є суттєві відмінності. По-перше, як ми вже зазначали, ми компонент TOpenDialog у явному вигляді взагалі не використовуємо. А ім'я файлу ми отримуємо за допомогою властивості FileName класу TFi-leOpen. По-друге, щоб зробити доступними пункт меню "Обчислити" та відповідну кнопку на панелі інструментів у попередній програмі в обробнику Menu_OpenClick ми змушені були записувати два оператори

```
MCalculate.Enabled:=   true;
TB_Calculate.Enabled:= true;
```

А тут достатньо написати один

```
A_Calculate.Enabled:= true;
```

Сформуйте тепер інші пункти меню та призначте їм відповідні дії з TActionList. Напишіть потрібні обробники.

Тепер помістіть форму TToolBar. Додайте необхідні кнопки і так само для кожної кнопки у властивості Action встановіть відповідні дії.

Остаточний код програми буде наступним:

```
unit Unit1;
```

```
{$mode objfpc}{$H+}
interface
uses
  Classes,  SysUtils,  FileUtil,  LResources,  Forms,
  Controls, Graphics, Dialogs, ActnList, Menus, StdActns,
  ComCtrls, TAGraph, TASeries;
type
  {TForm1}
  TForm1 = class(TForm)
    A_Grafic_Curve: TAction;
    A_Grafic_Exp: TAction;
    A_Calculate: TAction;
    ActionList1: TActionList;
    Chart1: TChart;
    A_Exit: TFileExit;
    FileOpen1: TFileOpen;
    ImageList1: TImageList;
    MainMenu1: TMainMenu;
    Menu_Curve: TMenuItem;
    Menu_Exp: TMenuItem;
    Menu_Calculate: TMenuItem;
    MCalculate: TMenuItem;
    Menu_Grafic: TMenuItem;
    Menu_Exit: TMenuItem;
    Menu_File: TMenuItem;
    Menu_Open: TMenuItem;
    ToolBar1: TToolBar;
    TB_Open: TToolButton;
    TB_Exit: TToolButton;
```

```
    TB_Divide_1: TToolButton;
    TB_Calculate: TToolButton;
    TB_Divide_2: TToolButton;
    TB_Graf_Exp: TToolButton;
    TB_Graf_Curve: TToolButton;
    procedure A_CalculateExecute(Sender: TObject);
    procedure A_ExitExecute(Sender: TObject);
    procedure A_Grafic_CurveExecute(Sender: TObject);
    procedure A_Grafic_ExpExecute(Sender: TObject);
    procedure FileOpen1Accept(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
procedure gauss(vector: array of real; b: array of real;
                var x: array of real; n: byte;
                var solve: byte);
// Процедура рішення СЛАУ методом Гауса
// n - розмірність системи,
// solve = 0, якщо рішення єдине,
// solve = 1, якщо система не має рішення,
// solve=2, якщо система має безліч рішень,

функція fx(t: real): real;
// Функція, що підбирається методом
// найменших квадратів
function stepen(x: real; n: byte): real;
```

```
// Функція зведення у ступінь
var
  Form1: TForm1;
  n: byte;
  x1, y1: real;
  x, y, z: array of real;
implementation
функція fx(t: real): real;
begin
  Result:= z[0] + z[1]*t + z[2]*t*t +
           z[3]*t*t*t + z[4]*sqr(sqr(t));
end;
function stepen(x: real; n: byte): real;
// процедура зведення на цілий ступінь
var
  i: integer;
begin
  Result:= 1;
  for i:= 1 to n do
  Result:= Result*x;
end;
// Реалізація методу Гауса
procedure Gauss(vector: array of real; b: array of real;
                var x: array of real; n: byte;
                var solve: byte);
var
 a: array of array of real; { матриця коефіцієнтів системи,
 двовимірний динамічний масив}
 i, j, k, p, r: integer;
```

```
 m, s, t: real;
begin
 SetLength(a, n, n); // Встановлення фактичного розміру
 масиву
 { Перетворення одновимірного масиву на
 двовимірний } k:=0;
 for i:=0 n-1 do for
   j:=0 n-1 do begin
     a [i, j]: = vector [k];
     k:=k+1;
   end;
 for k:=0 n-2 do
 begin
   for i:=k+1 n-1 do
   begin
     if (a[k, k]=0) then
     begin
       {перестановка рівнянь}
       p:=k; // В алгоритмі використовується буква l, але
       вона схожа на 1
             // Тому використовуємо ідентифікатор p
       для r:=i до n-1 до
       початку
         if abs(a[r, k]) > abs(a[p, k]) then p:=r;
       end;
       if p<>k then
       begin
         for j:= до n-1 до
         початку
           t: = a [k, j];
```

```
          a [k, j]: = a [p,
          j]; a [p, j]: =
          t;
        end; t: = b
        [k];
        b[k]:=b[p];
        b[p]:=t;
      end;
    end; // кінець блоку перестановки рівнянь
    m: = a [i, k] / a
    [k, k]; a[i,
    k]:=0;
    for j:=k+1 n-1 do begin
      a [i, j]: = a [i, j]-m * a
    [k, j]; end;
    b[i]:= b[i]-m*b[k];
  end;
end;
{Перевірка існування
рішення} if a[n-1,n-1] <> 0
then begin
  x[n-1]:=b[n-1]/a[n-1,n-
  1];for i:=n-2 downto 0 do
  begin
    s:=0;
    for j:=i+1 n-1 do begin
      s:=sa[i, j]*x[j];
    end;
    x[i]:=(b[i] + s)/a[i, i];
```

```
   end;solve:
   = 0;
 end
 else
 if b[n-1] = 0 then
 begin
   MessageDlg('Система має нескінченне' +
             'кількість рішень', mtInformation, [mbOK], 0);
   solve: = 2;
 end
 else
 begin
   MessageDlg('Система не має рішень',
                mtInformation, [mbOK], 0);
   solve:= 1;
 end;
 { звільнення пам'яті,
```

розподіленої для динамічного масиву a:=nil;

```
end;
{TForm1}
procedure TForm1.FileOpen1Accept(Sender: TObject);
// процедура вибору, відкриття та читання файлу даних
var
  f: TextFile;
  i: integer;
  Fname: string;
begin
  Fname:= FileOpen1.Dialog.FileName;
```

```
Fname:= UTF8ToSys(Fname); //перетворення на системне кодування
AssignFile(f, Fname);
Reset(f);
// відключення контролю помилок введення/виводу
{$I-}
// Читання кількості експериментальних точок
Readln(f, n);
if IOResult <> 0 then
begin
  ShowMessage('Помилка при читанні з
  файлу!'); exit;
end;
// Розподіл пам'яті під
масиви SetLength(x, n);
SetLength(y, n);
for i:= 0 to n - 1 do
begin
  read(f, x[i]);
  if IOResult <> 0 then
  begin
    ShowMessage('Помилка при читанні з
    файлу!'); exit;
  end;
end;
for i:= 0 to n - 1 do
begin
  read(f, y[i]);
  if IOResult <> 0 then
  begin
```

```
      ShowMessage('Помилка при читанні з
      файлу!'); exit;
    end;
  end;
  {$I+}
  CloseFile(f);A_Calculate.Enabl
  ed:= true;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Chart1.Title.Text.Text:='Метод найменших
  квадратів'; A_Calculate.Enabled:= false;
  A_Grafic_Exp.Enabled:= false;
  A_Grafic_Curve.Enabled:= false;
  Chart1.Visible: = false;
end;
procedure TForm1.A_ExitExecute(Sender: TObject);
begin
  Close;
end;
procedure TForm1.A_Grafic_CurveExecute(Sender: TObject);
{Процедура виведення суміщених графіків
експериментальних даних за точками та
підібраною методом найменших квадратів
кривою, що найкраще наближається до
експериментальних даних}
var
  i: integer;
  gr1, gr2: TLineSeries;
```

```
begin
  Chart1.Visible: = true;
  gr1:= TLineSeries.Create(Chart1);
  gr1.ShowPoints: = true; // графік з точками
  gr1.ShowLines: = false; // не з'єднувати крапки
  лініями Chart1.AddSeries(gr1);
  gr2:= TLineSeries.Create(Chart1);
  gr2.ShowLines := true;
  Chart1.AddSeries(gr2);
  для i:= 0 до n - 1 do
  gr1.AddXY(x[i], y[i]);
  for i:= 0 to n - 1 do
  gr2.AddXY(x[i], fx(x[i]));
end;
procedure TForm1.A_Grafic_ExpExecute(Sender: TObject);
{Процедура виведення графіка
експериментальних даних за точками}
var
  i: integer;
  gr1: TLineSeries;
begin
  Chart1.Visible: = true;
  gr1:= TLineSeries.Create(Chart1);
  Chart1.AddSeries(gr1);
  для i:= 0 до n - 1 do
  gr1.AddXY(x[i], y[i]);
end;
procedure TForm1.A_CalculateExecute(Sender: TObject);
var
```

```
  i, j, k, l: integer;
  b, vector: array of real;
  s: real;
  solve: byte;
begin
  SetLength(z, 5);
  SetLength(b, 5);
  SetLength(vector, 25);
  j: = 0;
  for k:= 0 to 4 do
  for l:= 0 to 4 do
  begin
    s: = 0;
    for i:= 0 to n - 1 do
    s:= s + stepen(x[i], k+l);
    vector[j]:= s;
    j:= j+1;
  end;
  for k:= 0 to 4 do
  begin
    s: = 0;
    for i:= 0 to n - 1 do
    s:= s + y[i]*stepen(x[i], k); b
    [k]: = s;
  end;
  gauss(vector, b, z, 5, solve); // Рішення СЛАУ
  if solve = 0 then
  begin
    A_Grafic_Exp.Enabled:= true;
```

```
    A_Grafic_Curve.Enabled:= true;
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

6.3.11.4. Створення програм із змінними розмірами вікон

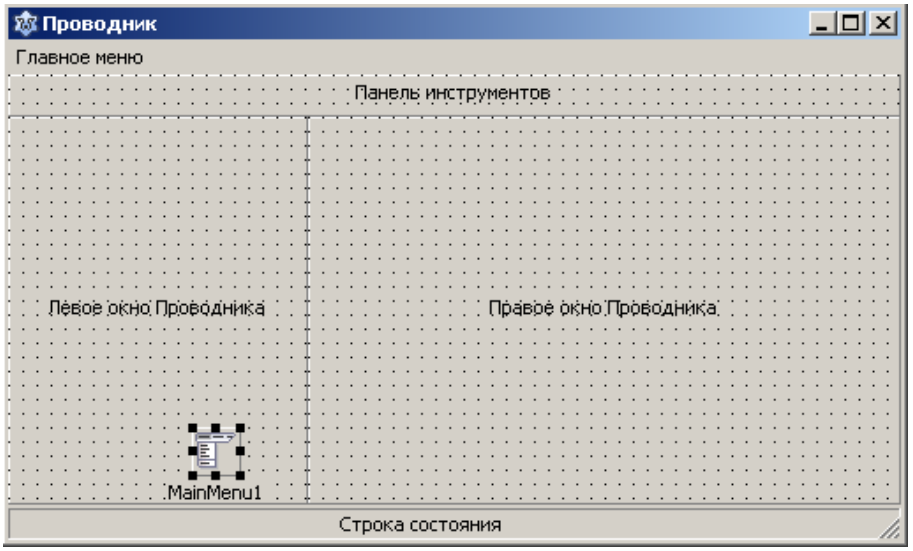
Для цього використовуються компоненти TPanel і TSplitter. Компонент TPanel – панель є контейнером, на якому можуть розміщуватись будь-які інші компоненти. Компонент дозволяє створювати на формі окремі незалежні області та групувати в них функціонально пов'язані інтерфейсні елементи.

У більшості компонентів, у тому числі й у TPanel є властивість Align – вирівнювання. Властивість може приймати такі значення:

- alNone – вирівнювання відсутнє;
- alLeft – компонент займає ліву частину клієнтської області компонента-контейнера (на панель можна поміщати скільки завгодно інших компонентів TPanel) або форми;
- alRight – компонент займає праву частину клієнтської області компонента-контейнера;
- alTop – компонент займає верхню частину клієнтської області компонента-контейнера;
- alBottom – компонент займає нижню частину клієнтської області компонента-контейнера;
- alClient – компонент займає всю вільну клієнтську область компонента-контейнера;
- alCustom – вирівнювання визначається батьківським компонентом.

Помістіть на порожню форму компонент TPanel і поекспериментуйте з різними значеннями властивості компонента Align. При цьому враховуйте такі моменти. Відповідно до стандартів графічного інтерфейсу у вікні програми першою має йти рядок меню. Якщо ви помістите на форму компонент TMainMenu і створите хоча б один пункт меню, то навіть якщо ви надаєте властивості Align TPanel значення alClient, панель не займе всю область форми. Тобто. меню не буде закрите панеллю!

Значення alTop та alBottom мають більший пріоритет, ніж alLeft та alRight. Виходячи зі сказаного, при проектуванні зовнішнього вигляду, наприклад Провідника, можна створити приблизно такий каркас програми, рис. 6.107.



Мал. 6.107. Приклад проектування зовнішнього вигляду програми

Спочатку за допомогою TMainMenu створюємо хоча б один пункт меню. Потім розміщуємо компонент TStatusBar – рядок стану. Компонент має за промовчанням властивість Align = alBottom. Поміщаємо на форму три компоненти TPanel. Першій панелі присвоюємо значення Align = alTop, в ньому ми будемо розміщувати панель інструментів, другій панелі присвоюємо значення Align = alLeft, сюди ми помістимо компонент TTreeView, причому властиво-

Align цього компонента присвоюємо значення alClient. Третій панелі надамо значення Align = alClient, на нього ми помістимо компонент TListView і його властивості Align надамо значення alClient.

Тепер, якщо користувач натисне кнопку розкриття вікна на весь екран, всі компоненти будуть пропорційно змінювати свої розміри і ваше застосування в цьому випадку збереже "пристойний" вигляд.

Практично всі компоненти мають також властивість Anchors за допомогою якого здійснюється прив'язка компонента до батьківського компонента. Зазвичай Lazarus в залежності від значення властивості Align компонента автоматично підставляє найбільш відповідні значення та властивості Anchors.

Група властивостей TPanel дає змогу керувати весняним виглядом панелі. Це такі властивості, як BevelInner, BevelOuter, BevelWidth, BorderStyle, BorderWidth. Просто спробуйте задавати різні значення цим властивостям і ви все зрозумієте.

Часто потрібно змінювати розміри вікна програми в цілому, а розміри компонентів у вікні. Наприклад, змінити розміри лівого вікна Провідника (компонент TTreeView). Причому в даному випадку користувач може змінювати лише ширину вікна. Тому правильніше говорити про зміну меж панелей, на яких розташовані компоненти. Для цього застосовується компонент TSplitter – роздільник (вкладка Additional). Щоб вставити роздільник між двома панелями, необхідно спочатку помістити першу панель, встановити стиль вирівнювання (властивість Align). Для нашого випадку задаємо для першої панелі Align = alLeft. Потім поміщаємо на форму TSplitter. За умовчанням він має властивість Align = alLeft. Тому роздільник автоматично притиснеться до панелі. Тепер поміщаємо на форму другу панель і задаємо Align=alClient. Все, тепер під час роботи програми користувач може переміщати межі панелей, ухопившись мишею за роздільник.

Післямова

Як сказали б люди в Стародавньому Сході, глечик моїх думок показує дно. На жаль, нами залишилася нерозглянута низка питань. Це, зокрема, графіка та мультимедіа, бази даних, мережеві програми, створення Інтернет-додатків та багато іншого. Але розміри книги набули вже настільки жахливі розміри, що настав час зупинитися!

Автор висловлює надію, що і в такому вигляді книга принесла вам користь, прочинила вікно у великий і прекрасний світ програмування! Пробудила інтерес до цієї надзвичайно творчої професії, якою є професія програміста.

Безумовно, для оволодіння хоча б азами цієї професії мало прочитання однієї книги і, безумовно, знання грають теж дуже велику роль. Але, і автор це неодноразово наголошував, щоб стати хорошим програмістом, перш за все потрібні талант і працьовитість.

Якщо ви, шановний читачу, вирішили пов'язати своє подальше життя з програмуванням, автор бажає вам успіхів і успіхів у цьому важкому, тернистому, але надзвичайно цікавому шляху!

І на закінчення, всі свої пропозиції та зауваження щодо змісту книги надсилайте на адресу:mansurov2002@inbox.ru

З повагою, автор.

Література

1. Демидович Б.П., Марон І.А. "Основи обчислювальної математики", М.: "Наука", 1970.
2. Копченова Н.В., Марон І.А. "Обчислювальна математика в прикладах і завданнях", М.: "Наука", 1972.
3. Гутер Р.С., Овчинський Б.В. "Елементи чисельного аналізу та математичної обробки результатів досвіду", М.: "Наука", 1970.
4. Архангельський А.Я. "Програмування в Delphi 7", М.: "Біном", 2003.
5. Іванова Г.С., Нічушкіна Т.М., Пугачов Є.К. "Об'єктно-орієнтоване програмування", М.: Вид-во МДТУ ім. н.е. Баумана, 2003.
6. Кенту М. "Delphi 7: Для професіоналів", СПб.: Пітер, 2004.
7. Єршов А.П. "Введення у теоретичне програмування", М.: "Наука", 1977.
8. Вірт Н. "Алгоритми та структури даних", М.: Світ, 1989.
9. Батіг Д.Е. "Мистецтво програмування", т.1. "Основні алгоритми", М.: "Світ", 1976.
10. Батіг Д.Е. "Мистецтво програмування", т.3. "Сортування та пошук", М.: "Світ", 1978.
11. Ламуатьє Ж.П. "Вправи з програмування на Фортрані-IV", М: Світ, 1978.
12. Дж. Бакнелл "Фундаментальні алгоритми та структури даних у Delphi", СПб ДіаСофтЮП, 2003 р.
13. Сайт FreePascal.ru -http://www.freepascal.ru/
14. Форум спільноти ALT Linux –http://forum.altlinux.org/
15. Форум програмістів та сисадмінів –http://www.cyberforum.ru/
16. http://freepascal.org/
17. http://lazarus.freepascal.org/

Алфавітний покажчик