Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

І. М. Дудзяний

Програмування мовою Object Pascal

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів вищих навчальних закладів

Львів
Видавничий центр ЛНУ імені Івана Франка
2003

УДК 004.438 Object Pascal (075.8)
Д–81
ББК З973.2-018я73 Object Pascal

Рецензенти:

Ю.М. Рашкевич, д-р техн. наук, проф. (Національний університет "Львівська політехніка")

О.М. Ананьєв, канд. екон. наук, доц. (Львівська комерційна академія)

Рекомендовано до друку Міністерством освіти і науки України як навчальний посібник для студентів вищих навчальних закладів (лист Міністра за № 1/11-2971 від 15 липня 2003 р.)

Д–81 **Дудзяний** І.М. Програмування мовою Object Pascal. Навчальний посібник. - Львів: Видавничий центр ЛНУ імені Івана Франка, 2003. - 328 с.

ISBN 966-613-174-9

Розглянуто основи програмування мовою Object Pascal у середовищі Delphi 6. У підручнику подано теми, які вивчають студенти факультету прикладної математики та інформатики у межах курсів "ЕОМ та програмування" і "Програмне забезпечення".

У посібнику систематизовано основні прийоми програмування мовою Object Pascal: опис типів даних, оголошення змінних і констант, організацію галужень і циклів, опис і використання структурованих типів даних, покажчиків, підпрограм і модулів. Розглянуто поняття і принципи об'єктно-орієнтованого програмування: класи, об'єкти, конструктори і деструктори, інкапсуляцію, поліформізм, наслідування, методи, властивості та події. На достатньо детальному для навчальних цілей рівні описано бібліотеку візуальних компонентів (VCL), яка використовується у середовищах Delphi та C++ Builder. Детально описано технологію розроблення проектів у середовищі Delphi 6.

ББК З973.2-018я73 Object Pascal

ISBN 966-613-174-9

© Дудзяний І.М., 2003

ВСТУП

Мова програмування Pascal була розроблена швейцарським ученим Ніклаусом Віртом у 1968 році і спочатку призначалася для навчальних цілей. Мова є добре детермінованою, тобто все підкоряється певним правилам, винятків з яких не так багато. Відносно невелика кількість базових понять, простий синтаксис і швидкі компілятори досить швидко зробили її популярною у всьому світі.

У 1992 р. фірма Borland International випустила два пакети, які базувалися на мові Паскаль: Borland Pascal 7.0 і Turbo Pascal 7.0. Перший може працювати в трьох режимах: звичайному і захищеному режимах MS DOS і у системі Windows. Turbo Pascal 7.0 працює тільки у звичайному режимі MS DOS і є менш вимогливий до характеристик комп'ютера порівняно з Borland Pascal 7.0.

У лютому 1995 р. Borland International випустила пакет Delphi 1.0 – інтегроване середовище розроблення програмних продуктів на базі мови Object Pascal у операційній системі Windows. З тих пір пакет Delphi постійно розвивають та удосконалюють. На цей час найбільш розповсюдженими є шоста та сьома версії.

Пакет Delphi об’єднує у собі візуальні засоби проектування програмних продуктів, оптимізуючий компілятор з Object Pascal, підтримку сучасних інформаційних технологій, роботу з базами даних тощо. Побудований на архітектурі об’єктно-орієнтованих компонентів, Delphi дає змогу багатократно використовувати уже створений код, зменшуючи тим самим час і зусилля, які витрачаються на розроблення програмних продуктів.

У посібнику подано теми, які вивчають студенти факультету прикладної математики та інформатики у межах курсів “ЕОМ та програмування” та “Програмне забезпечення”. Зауважимо, що систематичне викладення основних прийомів програмування мовою Object Pascal у книзі основано на кодуванні типових алгоритмів роботи з масивами, рядками, записами, множинами, файлами і динамічними структурами даних.

Детально розглянуто поняття і принципи об’єктно-орієнтованого програмування: класи, об’єкти, конструктори і деструктори, інкапсуляцію, поліформізм, наслідування, методи, властивості та події. На достатньому рівні розглянуто бібліотеку візуальних компонентів VCL і технологію розроблення проектів у Delphi.

Усі навчальні програми і проекти, які описано у посібнику, були протестовані автором у пакеті Delphi 6. Але автор не акцентує уваги на виборі і порівнянні різних версій пакета. Більшість програм і проектів без проблем тестували студенти групи ПМа-22 (2002-2003 навч. рік) у пакетах Delphi 5 і Delphi 7, за що автор їм дуже вдячний.

Слово “Паскаль” автор вживає у випадках, коли треба підкреслити відмінність у реалізації певного елемента мови у традиційному Паскалі (зокрема, у Turbo Pascal 7.0) від її реалізації у Object Pascal (див., наприклад, параграф 1.6 і розділи 8, 9).

Для опису виконання послідовності певних операцій у інтегрованому середовищі будемо використовувати систему позначень, яка запозичена у [28]:

Позначення	Виконання операції
➢**Назва команди**	**Вибір** у поточному меню команди з вказаною назвою (встановити курсор миші на команду і однократно натиснути на ліву кнопку миші)
❑**Назва кнопки**	**Натискання** кнопки з вказаною назвою у активному діалоговому вікні (встановити курсор миші на кнопці і однократно натиснути на ліву кнопку миші)
⇩**Назва списку**	**Розкриття** списку (встановити курсор миші на кнопку розкриття списку і однократно натиснути на ліву кнопку миші)
⇗**Елемент списку**	**Вибір** елемента списку (встановити курсор миші на вказаний елемент і однократно натиснути на ліву кнопку миші)
🗀**Назва закладки**	**Вибір** закладки з вказаною назвою (встановити курсор миші на ярлику закладки і однократно натиснути на ліву нопку миші)
Назва поля:=*значення*	**Уведення** *значення* з клавіатури у текстове поле введення, список або лічильник. Значення лічильника можна змінювати і за допомогою кнопок-регуляторів
⦿**Назва перемикача** (радіокнопки)	**Вибір** перемикача з вказаною назвою у активному діалоговому вікні (встановити курсор миші на перемикач і однократно натиснути на ліву кнопку миші)
☑**Назва індикатора** (прапорця, опції)	**Позначення** індикатора з вказаною назвою у активному діалоговому вікні (встановити курсор миші на індикаторі та однократно натиснути на ліву кнопку миші)
Клавіша1+Клавіша2	Одночасне натискання *Клавіші1* і *Клавіші2*

1. ОСНОВНІ КОНСТРУКЦІЇ МОВИ OBJECT PASCAL

План викладу матеріалу

1. Алфавіт мови. Ідентифікатори
2. Операції та вирази
3. Прості типи даних
4. Порожній оператор. Оператор присвоєння
5. Процедури введення/виведення даних у консольному режимі
6. Структура консольної програми
7. Перетворення типів даних

Ключові терміни розділу

- *Алфавіт мови*
- *Ідентифікатори*
- *Ключові слова*
- *Коментарі*
- *Літерали*
- *Символьні константи*
- *Рядки символів*
- *Визначення змінних*
- *Вирази*
- *Пріоритет операцій*
- *Правила обчислення виразів*
- *Поняття типу даного*
- *Цілі типи*
- *Раціональні типи*
- *Логічний тип*
- *Перелічений та обмежений типи*
- *Визначення власного типу*
- *Порядкові типи*
- *Порожній оператор*
- *Оператор присвоєння*
- *Консоль*
- *Консольна програма (застосування)*
- *Процедура введення даних*
- *Процедура виведення даних*
- *Організація паузи*
- *Перетворення типів даних*

1.1. Алфавіт мови. Ідентифікатори

Алфавіт мови Object Pascal містить великі та малі *латинські букви*: `A`, …, `Z`, `a`, …, `z`, _ (символ підкреслення, використовується нарівні з буквами); *арабські цифри*: 0, 1, …, 9; *спеціальні символи*:

```
+ - * / < > = ' @ # $ ^ : ; , . ( ) [ ] { }.
```

Використання символів *кирилиці* допускається у коментарях, рядках символів, назвах файлів (якщо це можливо в операційній системі).

Речення (*оператори*) мови Object Pascal завершують крапкою з комою. У деяких випадках крапку з комою можна опускати. Ці

випадки будуть розглянуті окремо. Зокрема, крапку з комою можна не ставити (а можна й ставити) перед ключовим словом **end**.

Символом-роздільником елементів (*лексем*) мови зазвичай є *символ пропуску* (код 32), який на екрані не зображається. До роздільників відносять і символи керування (коди від 0 до 31), зокрема: *табуляцію* (код 09); складений *символ переходу до нового рядка* (послідовність кодів 13 – “переведення каретки” і 10 – “переведення рядка”). Між будь-якими двома лексемами можна задавати довільну кількість символів-роздільників.

Багато елементів програми мають *назви* (*ідентифікатори*): константи, змінні, типи даних, функції тощо. Назви вводять, щоб відрізнити (ідентифікувати) різні елементи одного типу від інших і оперувати ними. **Ідентифікатор** – послідовність букв і цифр довільної довжини (значущими є перші 64 символи); перший символ має бути буквою. Символ підкреслення `'_'` вважається буквою. Великі і малі букви в ідентифікаторах вважають однаковими символами. Тому, наприклад, ідентифікатори `Suma, suma` позначають *один і той же* елемент програми.

Деякі ідентифікатори мови зарезервовані для службових цілей і їх не можна використовувати з метою найменування констант, змінних і функцій, наприклад: **array, const, do, if** тощо. Такі ідентифікатори називають *ключовими* (*зарезервованими*) *словами* і вони входять у алфавіт мови. У редакторі тексту програми Object Pascal зарезервовані слова виділяють спеціальним шрифтом і/або кольором, щоб можна було зразу ж виявити помилки при наборі та використанні цих слів. Перелік ключових слів:

and	**exports**	**mod**	**shr**
array	**file**	**nil**	**string**
as	**finalization**	**not**	**then**
asm	**finally**	**object**	**threadvar**
begin	**for**	**of**	**to**
case	**function**	**or**	**try**
class	**goto**	**out**	**type**
const	**if**	**packed**	**unit**
constructor	**implementation**	**procedure**	**until**

destructor	**in**	**program**	**uses**
dipinterface	**inherited**	**property**	**var**
div	**initialization**	**raise**	**while**
do	**inline**	**record**	**with**
downto	**interface**	**repeat**	**xor**
else	**is**	**resourcestring**	
end	**label**	**set**	
except	**library**	**shl**	

Крім зарезервованих слів, у Object Pascal є ще *стандартні назви*, які визначають вбудовані типи, директиви чи стандартні процедури/функції, наприклад: `integer`, `real`, `private`, `public`, `sin`, `cos`, `exp` тощо. Стандартні назви при певних обмеженнях можна перевизначити. Ця можливість є чисто теоретичною, і на практиці її бажано не використовувати (завжди можна придумати якусь оригінальну та осмислену назву елемента програми). Наведемо перелік стандартних директив:

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

Слова **private, protected, public, published** та **automated** вважають зарезервованими словами всередині оголошення класу і стандартними директивами ззовні оголошення.

Для пояснення окремих частин програми використовують *коментарі*. **Коментарі** є двох видів:

```
{ текст, який може бути на місці будь-якого пропуску }
// текст до кінця рядка, на якому він з'явився.
```

Замість фігурних дужок у коментарі можна використовувати символи `(*` і `*)`, наприклад: `(* Це також коментар! *)`.

Коментарі, які обежують з обох боків (фігурними дужками або зірочкою з круглою дужкою), можуть займати будь-яку кількість рядків. Текст коментаря ігнорується при компіляції, якщо це не *директиви компілятора*, які мають вигляд `{$ ...}`.

У рядку програми можна записати декілька операторів. Однак з погляду простоти читання тексту цим не треба зловживати. Рекомендується виділяти об'єднані одним змістом оператори в групи, використовуючи для цього відступи і коментарі. Операторні дужки **`begin`** і **`end`** виділяють *складений оператор*. Всі оператори, розміщені між операторними дужками, сприймаються синтаксично як один оператор.

Всі *типи*, *мітки*, *константи*, *змінні*, *функції/процедури*, які використовуються у програмі, мають бути оголошені чи описані до їхнього першого використання.

Програма або модуль Object Pascal завершується оператором **`end.`** (ключове слово **`end`** з символом крапки).

1.2. Операції та вирази

Операції та *вирази* задать певну послідовність дій, але не є закінченими реченнями мови. *Прості вирази* містять знак операції та операнди, наприклад `3.14+x`. Операції можна виконувати з одним або двома операндами. Відповідно до цього розрізняють *унарні* та *бінарні* операції. Операндами можуть бути *константи*, *змінні*, *виклики функцій* і *вирази*.

Константами називають елементи програми, значення яких не міняються в процесі її виконання. Вони бувають або безпосередньо у вигляді значення (*літерала*), або у вигляді ідентифікатора константи, описаного в розділі **`Const`**. Константами у Object Pascal можуть бути цілі, раціональні та шістнадцяткові числа, логічні константи, символи, рядки символів, конструктори множин і ознака невизначеного вказівника **`nil`**.

Цілі літерали (цілі числа) містять лише цифри і знак: `-24`, `73`; *раціональні* – можуть містити також десяткову крапку, показник

степеня і символ `'e'`, який замінює основу 10 в записі числа: `-0.5`, `-1e-5`, `7.2e+15`. Шістнадцяткове число складається зі шістнадцяткових цифр (`0, 1, ...,A, B, C, D, E, F`), де першим стоїть знак долара `$` (код символу 36). Діапазон шістнадцяткових чисел – від `$0` до `$7FFFFFFFFFFFFFFF` (для версій 4 – 6).

Логічні константи – це значення `False` або `True`.

Символьна константа – це символ ANSI, взятий в апострофи (наприклад: `'f'`). Якщо символ не має фізичного зображення, то пишеться знак # і поряд ANSI–код символу без апострофів (наприклад: `#7`). Якщо є необхідність записати сам символ апострофа, то його записують так: `''''` (символ апострофа подвоюється).

Допускається записувати символ через задання його внутрішнього коду, перед яким стоїть символ # (код 35), наприклад: `#97` – символ `'a'`; `#90` – символ `'Z'`.

Рядок символів – це послідовність довільних символів ANSI (крім символу `CR` – повернення каретки), взята в апострофи (наприклад: `'Рядок'`). Якщо у рядку символів треба вказати сам символ апострофа, то від подвоюється, наприклад: `'Ім''я'`.

Рядок символів може бути і порожним (не мати ніяких символів, тільки апострофи, що його позначають). Рядок символів можна формувати з кодів потрібних символів, наприклад, рядок #83#121#109#98#111#108 еквівалентний рядку `'Symbol'`.

У рядку символів можна чередувати частини, записані у апострофах, з частинами, записаними кодами. Таким чином можна вставляти у рядки довільні символи, у т. ч. і символ `CR` (код 13), наприклад: `#7'Помилка!'#13'Натисніть пропуск ...'#7`

Конструктор множини – список елементів множини, взятий у квадратні дужки, наприклад:

```
[1, 2, 5..8, 10]
[blue, red, white, black]
```

Змінними називаються елементи програми, які можуть міняти своє значення у процесі її виконання. Всі без винятку змінні мають бути описані в розділі програми **`var`**.

Виклик функції – це вказівка назви функції, за якою у круглих дужках записують список аргументів (можливо, порожний), як-от: `F(x+c, y)`.

Символи: `'+'`, `'-'` і `'*'` є знаками *бінарних* арифметичних операцій *додавання*, *віднімання* і *множення* відповідно. Символ `'/'` позначає операцію ділення раціональних чисел. Для цілих чисел визначені бінарні операції *цілочисельного ділення* (**div**) і *знаходження залишку* від цілочисельного ділення (**mod**). Для бінарних арифметичних операцій діють такі правила:

- результат операції `X/Y` має тип `Extended` для будь-яких допустимих типів операндів;
- результат операції `X+Y`, `X-Y` чи `X*Y` має тип `Extended`, якщо хоча б один операнд має раціональний тип. В інших випадках результат операції буде мати тип `Int64`, якщо хоча б один операнд має тип `Int64`, а інакше результат матиме тип `integer`;
- результат цілочисельного ділення `X` **div** `Y` дорівнює результату ділення `X/Y`, заокругленому до найближчого цілого. Якщо `Y=0`, то генерується виняток `EDivByZero`;
- результат операцій `X` **div** `Y` і `X` **mod** `Y` завжди має тип `integer`.

Прикладом *унарної* арифметичної операції є операція *зміни знака числа*, яка позначається символом “мінус”, що стоїть перед одним операндом, як-от: `-x+y`. Ще одною унарною арифметичною операцією є операція *підтвердження знака числа*, яка позначається символом “плюс”, що стоїть перед одним операндом, як-от: `+x-y`. Як бачимо, зміст операції залежить від контексту.

Вираз – це послідовність знаків операцій, операндів і круглих дужок, яка задає обчислювальний процес отримання результату певного типу. *Найпростішими* виразами є *константи*, *змінні* та *виклики функцій*.

Порядок обчислення виразу визначають через розміщення круглих дужок, знаків операцій та *пріоритетність* операцій. Операції з найвищим пріорітетом виконуть першими. Якщо у виразі підряд розміщено декілька операцій одного і того ж пріорітету, то їх виконують *зліва направо*.

Крім арифметичних операцій, у Object Pascal є ще операції *відношення*, *логічні* (булівські) операції, *логічні порозрядні* операції, операції з *рядками*, операції з *покажчиками*, операції з *множинами*, операції з *класами*, операція *отримання адреси* (`@`). Всі ці операції будуть описані далі. Перелік і пріоритетність операцій наведено у таблиці 1.1.

Таблиця 1.1. Перелік і пріоритетність операцій

Пріоритет	**Знаки операцій**	**Назва операції**
1	**`Not`** `@`	Заперечення (або порозрядне заперечення) Отримання адреси

2	`*`
	`/`
	`div`
	`mod`
	`and`
	`shl`
	`shr`
	`as`
		Множення (або перетин множин)
		Ділення
		Цілочисельне ділення
		Залишок від цілочисельного ділення
		Логічне І (або порозрядне І)
		Зсування бітів вліво
		Зсування бітів вправо
		Операція з класами (*як*)

3	`+`
	`–`
	`or`
	`xor`
		Додавання (або об’єднання множин, або конкатенація рядків)
		Віднімання (або різниця множин)
		Логічне АБО (або порозрядне АБО)
		Логічне додавання за модулем 2 (або порозрядне додавання за модулем 2)

4	`= < > <= >= <>` **`in`** **`is`**	Операції відношення (або операції з множинами) Операція з множиною (*є елементом* множини) Операція з класами (*об’єкт належить* класу)

Зауваження. У таблиці 1.1 не вказані операції з покажчиками.

Унарна операція `@` застосовується до операнда довільного типу і повертає результат типу `Pointer`, в якому знаходиться адреса операнда. Якщо операція `@` застосовується до процедури, функції чи методу в об’єкті, її результатом буде адреса точки входу в цю

процедуру. Цю адресу можна використовувати тільки в підпрограмі, яка написана на асемблері, чи у фрагментах *INLINE*.

Порозрядні (побітові) *логічні* операції застосовують до змінних цілих типів і оперують з їхніми двійковими розрядами. Операція **not** є унарною, а всі інші – бінарні. Результат операції **not** має той самий тип, що й її операнд. Результат операцій **and**, **or** чи **xor** має найменший цілий тип, що включає у себе всі можливі значення типів обох операндів. Вирази `X` **shl** `Y` і `X` **shr** `Y` зсувають значення `X` вліво чи вправо на `Y` бітів, що еквівалентно множенню або діленню `X` на 2^Y. Результат має той самий тип що й операнд `X`.

Не слід плутати побітові логічні операції з відповідними логічними (булевими) операціями. Останні повертають булівські значення `false` чи `true`, і головно використовуються для запису перевірки умови в операторах **if**, **while** тощо. Булівські операції приймають операнди булівських типів і повертають результат булівського типу.

Зауважимо, що для роботи з числовими даними (константи, змінні та вирази) є велика кількість вбудованих (стандартних) функцій: `abs`, `sqr`, `sqrt`, `sin`, `cos`, `exp`, `ln` та ін. (див. вбудовану довідку).

1.3. Прості типи даних

У програмі оперують зі *змінними* та *константами*, які попередньо **визначають**, для того, щоб повідомити компілятору:

- назву (ідентифікатор) змінної чи константи;
- розмір і спосіб виділення пам’яті для зберігання значень;
- які допустимі дії можна виконувати зі змінною чи константою;
- можливе початкове значення.

Під *типом даних* розуміють множину допустимих значень цих даних і множину допустимих операцій над ними. Одночасно тип даних визначає і розмір пам’яті для зберігання значень змінної чи константи.

У Object Pascal типи ділять на *стандартні* (вбудовані) і типи, *визначені програмістом*. Деякі вбудовані типи (цілі, символи і рядки) ділять на **фундаментальні** типи (формат збереження даних у

пам'яті строго фіксується і залишається незмінним у всіх реалізаціях мови) та **родові** типи (формат збереження даних у пам'яті не фіксується і встановлюється найбільш оптимальним чином, залежно від платформи та операційної системи).

Стандартні типи класифікують так:

- Прості
  - Порядкові
    - Цілі
    - Символьний
    - Логічний
    - Перелічений
    - Обмежений (діапазон)
  - Раціональні (з плаваючою крапкою)
  - Дати-часу
- Рядки символів
- Структури
  - Множини
  - Масиви
  - Записи
  - Файли
  - Класи
  - Інтерфейси
- Покажчики
- Процедурні
- Тип `variant`

Програміст може визначити власний тип безпосередньо при описі змінної у розділі **`var`**, або, найчастіше, у спеціальному розділі опису типів – **`type`**. Після ключового слова **`type`** для кожного типу слідує конструкція, яка має вигляд:

```
ідентифікатор_типу = визначення_типу;
```

Наприклад, у програмі можна розмістити оголошення:

```
Type TColor=(Blue, Red, White);
Var c1, c2: TColor;
```

Можна використати й еквівалентне оголошення:

```
Var c1, c2: (Blue, Red, White);
```

Діапазони можливих значень і витрати пам'яті для цілих типів даних подано в табл. 1.2, а для раціональних – у табл. 1.3.

Таблиця 1.2. Типи **цілих** даних (обсяг пам'яті у байтах і діапазон значень)

Тип	**Байти**	**Діапазон допустимих значень**
`Byte`	1	Від 0 до 255
`Word`	2	Від 0 до 65 535
`LongWord`	4	Від 0 до 4 294 967 295
`ShortInt`	1	Від –128 до 127
`SmallInt`	2	Від –32 768 до 32 767
`Cardinal`	4	Від 0 до 4 294 967 295
`Integer`	4	Від –2 147 483 648 до 2 147 483 647
`LongInt`	4	Від –2 147 483 648 до 2 147 483 647
`Int64`	8	Від -2^{63} до $2^{63}-1$

Фундаментальними цілими типами є `Byte`, `Word`, `ShortInt`, `SmallInt` і `LongInt`. У програмах доцільно використовувати родові цілі типи `Integer` і `Cardinal`.

Таблиця 1.3. Типи **раціональних** даних

Тип	**Байти**	**Кількість цифр**	**Діапазон допустимих значень**
`Real48`	6	11-12	Від $\pm 2.9e^{-39}$ до $\pm 1.7e^{38}$
`Real`	8	15-16	Від $\pm 5.0e^{-324}$ до $\pm 1.7e^{308}$
`Single`	4	7-8	Від $\pm 1.5e^{-45}$ до $\pm 3.4e^{38}$
`Double`	8	15-16	Від $\pm 5.0e^{-324}$ до $\pm 1.7e^{308}$
`Extended`	10	19-20	Від $\pm 3.6e^{-4932}$ до $\pm 1.1e^{4392}$
`Comp`	8	19-20	Від $-2^{63}+1$до $2^{63}-1$
`Currency`	8	19-20	$\lvert x\rvert<922\ 337\ 203\ 685\ 477.5807$

Раціональний тип даних призначений для зберігання чисел з дробовою частиною. Тип `Currency` використовують для збереження даних про гроші. В разі операцій з величинами типу `Currency` мінімізуються помилки заокруглення, що є важливим для грошових розрахунків.

Раціональні числа зберігаються наближено. Кожний з раціональних типів гарантує правильне зберігання тільки певної кілько-

сті цифр (цифри мантиси), які є вірними цифрами. З математичної точки зору мова йде про відносну похибку.

Похибки у зберіганні раціональних чисел можуть призвести до того, що при відніманні близьких чисел може статися втрата значущості. Це ж пояснює, чому потрібно уникати порівняння раціональних величин на точну рівність. Вважають, що раціональні числа `a` і `b` є рівними (`a=b`), якщо виконується умова `abs(a-b)<eps`. Число `eps` можна визначати так:

```
min(abs(a), abs(b))*10^(-m),
```

де `m` – необхідне число десяткових розрядів, які збігаються.

Раціональні типи є **впорядкованими**, але не порядковими. Тип відносять до впорядкованих, якщо для змінних і виразів цього типу визначені операції відношення (=, <>, <, >, <=, >=). Будь-який порядковий тип є впорядкованим, але ніколи навпаки.

Кожному значенню порядкового типу функція `Ord(x)` підбирає відповідне натуральне число – порядковий номер значення виразу `x` у множині допустимих значень цього типу. Для цілих типів повертається саме значення, для логічного 0 або 1, для символьного – значення від 0 до 255.

До будь-яких порядкових типів також можна застосовувати функцію `Pred(x)`, яка повертає попереднє значення, і `Succ(x)` – наступне значення. Функція `Pred` не визначена для найменшого значення порядкового типу, а `Succ` – для найбільшого.

Змінні логічного типу `Boolean` займають у пам’яті один байт і можуть набувати одне з двох значень `False` (хибне) або `True` (істинне). Цей тип є порядковим (`Ord(False)=0`, `Ord(True)=1`). Результат будь-яких операцій *відношення* має логічний тип і може бути привласнений логічній змінні.

Для сумісності з логічними даними, які використовуються у Windows і деяких системах програмування (Visual C), у Delphi ще визначені такі логічні типи: `ByteBool` (1 байт); `WordBool` (2 байти); `LongBool` (4 байти). Результат операції порівняння чи логічної операції завжди має тип `Boolean`.

Змінні *символьного* типу `Char` дають змогу працювати з окремими символами. На відміну від самого тексту програми симво-

ли, які відповідають малим і великим буквам, розрізняються. Множиною значень типу `Char` є повний набір ASCII-символів (для консольних застосувань) чи ANSI-символів (для віконних застосувань). Функція `Ord` видає код відповідного символу, який може бути від 0 до 255. Оберненою функцією, яка за кодом видає відповідний символ, є функція `Chr`.

Крім родового типу `Char`, у Object Pascal визначено ще два фундаментальні символьні типи: `ANSIChar` і `WideChar`, які зв'язані з двома різними множинами символів: ANSI, кожний символ якого займає 1 байт (тип `ANSIChar`), і Unicode, кожний символ якого займає 2 байти (тип `WideChar`). Тип `Char` зараз еквівалентний `ANSIChar`, але в майбутніх версіях може буде еквівалентним `WideChar`.

Перелічений тип задається множиною тих значень, які він може отримувати. Кожне значення має бути ідентифікатором і розміщуватися у круглих дужках через кому. Кількість елементів у переліку не більше за 65536. Вводити і виводити значення змінних цього типу заборонено. Перелічений тип є порядковим, а тому до змінних такого типу можна застосовувати функції `Ord`, `Pred`, `Succ`. Функція `Ord` повертає порядковий номер значення, починаючи з нуля. Приклади використання перелічених типів:

```
Type Colors = (Red, Green, Blue);
Numbers = (Zero, One, Two, Three, Four, Five);
var z:Colors;   n:Numbers;
begin
z:=Red;         write(Ord(z));    {Виведеться 0}
n:=Four;        write(Ord(n));    {Виведеться 4}
z:=Succ(z);     {z=Green}
for n:=One to Five do write(Ord(n)); {12345}
end.
```

Потрібно зазначити, що стандартні типи `byte`, `word`, `char` і `boolean` також можна вважати варіантами переліченого типу.

Тип *діапазон* також називають *обмеженим* або *інтервальним* типом. Він є підмножиною свого **базового** типу, яким може виступати будь-який порядковий тип крім типу діапазону. Тип діапазон задається **межами** своїх значень у базовому типі (через двокрапку)

і успадковує всі властивості цього базового типу. Обмежені типи використовуються, наприклад, при оголошенні масивів. Проте можуть використовуватися і самостійно.

Є дві стандартні функції, які працюють з типом діапазону: `High(x)` – повертає максимальне значення діапазону, до якого належить змінна `x`; `Low(x)` – повертає мінімальне значення.
Приклади використання типу діапазон:

```
type Numbers = (Zero, One, Two, Three, Four, Five);
Num = Two .. Four; {діапазон на базі типу Numbers}
Abc = 'A' .. 'z'; {всі англійські букви: діапазон на
                    базі типу Char}
Digits = 0 .. 9; {цифри}
var  n:Num;  z, d: Abc;  x:SmallInt;
begin
n:=Four;   writeln(Ord(n));  {4, як у базовому типі}
n:=Succ(n); {Помилка: наступне значення поза
                        діапазоном}
read(z, d);
if z=d then write('Однакові букви');
writeln(Low(z),' .. ', High(z)); { A .. z }
writeln(Low(x),' .. ', High(x)); { -32768 .. 32767}
end.
```

У тексті програми можуть траплятися *директиви* компілятора (або *опції*). Опції {$R+} і {$R-} дають змогу включати і виключати перевірку дотримання меж при роботі з діапазонами. Коли перевірка включена, то при порушенні меж діапазонів відбувається аварійне завершення роботи програми. У іншому випадку відповідальність за можливі помилки лежить на програмістові.

1.4. Порожній оператор. Оператор присвоєння

Алгоритмічні дії над початковими даними і робочими об’єктами мови, необхідні для розв’язання поставленої задачі, описують за допомогою *операторів* Паскаля. Оператори розділяють крапкою з комою, їхня послідовність і складає тіло програми. Найбільш простий випадок – це *лінійні* ділянки алгоритмів, у яких оператори виконують послідовно один за одним у тому порядку, в якому вони

перераховані в програмі. Водночас можуть використовуватися оператори присвоєння та процедури введення/виведення.

Найпростішим оператором мови є *порожній оператор* (`;`), який містить тільки крапку з комою. Цей оператор нічого не робить. Однак він може бути корисний у тих випадках, коли синтаксис потребує наявності оператора, а для реалізації алгоритму жоден оператор не потрібний.

У програмі найчастіше трапляється *оператор присвоєння*, який використовується для задання значення змінним і має такий синтаксис:

```
назва_змінної:=вираз;
```

Наприклад:

```
b:=-5; a:=b*3+c; c:=c+1;
```

Під час виконання оператора присвоєння:

- обчислюється вираз, що стоїть у правій частині оператора;
- результат обчислення перетворюється до типу змінної у лівій частині оператора;
- результат присвоюється змінній лівої частини оператора.

Очевидно, що тип виразу і тип змінної мають бути **сумісні** (множина допустимих значень для типу виразу має міститися у множині допустимих значень для типу змінної).

1.5. Процедури введення/виведення даних у консольному режимі

Програми, які працюють в операційних системах Windows, прийнято називати **додатками** (або **застосуваннями**). В основі таких додатків є **проект** – множина програмних файлів (або **модулів**) та допоміжних файлів (файли ресурсів, файли конфігурації тощо).

Delphi дає можливість створювати і більш прості **консольні** застосування, у яких для введення/виведення даних використовується консоль (об’єднання клавіатури і екрана дисплея). Хоча консольне застосування створюється у Delphi, однак виконується у операційній системі DOS. Як відомо, у DOS використовується система кодування ASCII, а у Windows – ANSI, у яких коди кириличних символів мають різне кодування. Тому повідомлення консольних застосувань мають бути записані латинськими буквами.

Створити консольне застосування можна різними способами. У Delphi 6 найпростіше це зробити так:

➢*File* ➢ *New* ➢*Other…* 🗀*New* ❑*Console Application*

За домовленістю введення даних у консольному режимі здійснюють з клавіатури, а виведення – на екран дисплея. Процедури введення даних мають вигляд:

```
Read(список_змінних);
Readln(список_змінних);
Readln;
```

Змінні у `списку_змінних` відокремлюють комами. Друга процедура відрізняється від першої тим, що після введення переводить курсор на новий рядок, а точніше, у кінці своєї роботи прочитує код клавіші <*Enter*>. Третя процедура використовується для організації паузи – виконання програми продовжиться, як правило, тільки після натиснення клавіші <*Enter*>. Процедури виведення мають вигляд:

```
Write(список_виведення);
Writeln(список_виведення);
Writeln;
```

У `списку_виведення` крім назв змінних можна писати рядкові константи (послідовність символів в апострофах) і навіть вирази (виводимуться їхні значення). Друга процедура відрізняється від першої тим, що після виведення переводить курсор на новий рядок. Третя процедура просто переводить курсор на новий рядок.

Існує *форматоване виведення*, в якому задають кількість позицій, що відведені під число. Для цілого – після виразу або змінної через двокрапку вказують тільки кількість позицій. Для раціональних – додатково через двокрапку вказують кількість цифр у дробовій частині. Разом з тим відбувається заокруглення результату.

1.6. Структура консольної програми

У програмі (консольному застосуванні), написаній на Паскалі, можуть бути такі розділи:

```
Program ..;  { Заголовок програми }
```

```
Uses ..;  { Підключення модулів }
Label ..;  { Розділ оголошення позначок }
Const ..;  { Розділ оголошення констант }
Type ..;  { Розділ оголошення нових типів }
Var ..;  { Розділ оголошення змінних }
Procedure ..;  { Опис власних процедур }
Function ..;  { Опис власних функцій }
Begin { Початок основної програми }
 ...;
 { Оператори }
 ...
End. { Кінець основної програми }
```

Обов'язковою частиною є лише тіло програми, яке починається словом **begin**, а закінчується словом **end** з крапкою. Оператори у Паскалі розділяють крапкою з комою. Заголовок програми є хоч і необов'язковим, але бажаним елементом і складається із зарезервованого слова **program** та ідентифікатора – назви програми, за якою слідує крапка з комою. Порядок оголошень і описів не регламентується.

Приклад 1.1. Програма виведення повідомлення:

```
program prosta;
{$APPTYPE CONSOLE}
{Ця програма не потребує ніяких оголошень і описів}
begin write('Pryvt! My pochaly programuvaty!');
 {Цей рядок тексту з'явиться на екрані}
 Readln
end.
```

Приклад 1.2. Програма обчислення функції $y = \sqrt{\ln^2(\sin x + 2) + x^2}$:

```
program Project2;
{$APPTYPE CONSOLE}
var x,y,t: real; // оголошення змінних
begin
  write('Vvedit argument: ');
```

```
    readln(x); // введення аргументу
    t:=ln(sin(x)+2); // проміжна величина
    y:=sqrt(t*t+x*x); // обчислення функції
    write('y=', y); // виведення результату
    readln // організація паузи
  end.
```

Навіть для найпростіших консольних застосувань Delphi будує **проект** – множину різноманітних файлів. Розширення файлу визначає його зміст. У таблиці 1.4 перераховані основні файли проекту консольного застосування.

Таблиця 1.4. Основні файли проекту консольного застосування

Головний модуль програми (`Project1.dpr`)	Текстовий файл (модуль програми), який зберігає код програми
Файл параметрів проекту (`Project1.dof`)	Текстовий файл, що містить значення параметрів проекту, які впливають на процеси компілювання та компонування
Файл конфігурації системи Delphi 6 (`Project1.cfg`)	Текстовий файл зберігає інформацію про останній сеанс роботи з проектом
`Project1.exe`	Виконавчий файл застосування

Назву `Project1` при першому збереженні проекту можна поміняти на більш осмислену. Переглянути зміст вищевказаних файлів можна у будь-якому текстовому редакторі.

1.7. Перетворення типів даних

У виразах та операторах присвоєння можуть міститися елементи (змінні та константи) різних типів. У цьому випадку здійснюється *приведення* типів, що означає перетворення одного типу даного в інший. Приведення типів у Delphi може бути *явним* і *неявним*.

Неявне приведення типів автоматично здійснює компілятор Delphi за принципом: якщо операція має операнди різних типів, то тип операнда “*молодшого*” типу (тип, який займає менший обсяг

пам'яті) зводиться до типу операнда "старшого" типу (тип, який займає більший обсяг пам'яті). Інакше кажучи, менш точний тип зводиться до більш точного. Наприклад, якщо в операції беруть участь коротке ціле і довге ціле, то коротке ціле зводиться до довгого; якщо беруть участь цілий і дійсний операнди, то цілий зводиться до дійсного і т.д. Таким чином, після подібного зведення типів обидва операнди виявляються одного типу. І результат виконання операції має такий самий тип.

У операторі присвоювання відбувається зведення типу результату обчислення виразу правої частини до типу лівого операнда. Якщо тип лівого операнда "молодший", ніж тип результату, то можливою є втрата точності чи отримання неправильного результату взагалі, а тому такі перетворення спричиняють помилки компіляції. Так, наступна послідовність операторів призводить до помилки:

```
var i:byte; a:real;
begin a:=2; i:=a+1; // Помилка приведення типів
```

Як видно з наведеного прикладу, неявне автоматичне перетворення типів даних не завжди дає бажаний результат. Це можна виправити, застосувавши операцію явного перетворення типів даних, яка має вигляд:

```
ідентифікатор_типу(вираз)
```

Явне приведення типів даних можна здійснювати для будь-яких типів, що мають однаковий розмір. Водночас не допускається взаємне перетворення цілих та дійсних даних. Для перетворення дійсних чисел у цілі використовують функції `Inc` і `Trunc`. А цілі числа перетворюються у дійсні неявно. Так, наприклад, попередній приклад можна записати у вигляді:

```
var i:byte; a:real;
begin a:=2; i:= Trunc(a+1); // Все добре!
```

Явне приведення типів даних може використовуватися як у лівій, так і правій частині оператора присвоювання. Наприклад, можна записати такі оператори (змінна `Ch` має тип `Char`, а `I` – тип `Integer`):

```
Ch:= Char(I);
ShortInt(Ch):=I;
```

Явне приведення типу значення деякого виразу здійсюють після його обчислення, наприклад:

```
Ch:=Char(I-1);
```

? Запитання для самоперевірки

1. Що таке ідентифікатор?
2. Які символи входять в алфавіт мови?
3. Для чого використовують ключові слова?
4. Який коментар можна вставити на місці довільного пропуску?
5. Як задати коментар від певної позиції до кінця рядка?
6. Що розуміють під типом даних?
7. Що повідомляють компіляторові при визначенні змінних/констант?
8. Вкажіть і охарактеризуйте цілі типи даних.
9. Вкажіть і охарактеризуйте раціональні типи даних.
10. Що таке логічний тип?
11. Як зображують числа?
12. Як зображують окремі символи та рядки символів у мові?
13. Що таке символічні константи?
14. Що таке вираз? Вкажіть правила обчислення виразу.
15. Наведіть приклади операцій з однаковим пріоритетом.
16. Наведіть приклади порозрядних логічних операцій.
17. Що таке логічне І/АБО?
18. Що таке консоль?
19. Як позначають процедуру введення даних?
20. Як позначають процедуру виведення даних?
21. Що таке автоматичне перетворення типів даних?
22. Що таке явне перетворення типів даних?

Завдання для програмування

Завдання 1.1. Скласти програму обчислення функції (функцію вибирати відповідно до номера студента у списку студентів підгрупи):

1. $y = \cos^3(2x) - \log_2(x^2 + 2).$
2. $y = \sin^4(3x) - 2e^{x-2}.$
3. $y = \cos^2(2x) - \sin x$.
4. $y = \sin^3(3x) - \cos x$.
5. $y = \sqrt{x^2 + 3} - 2\log_5(x^2 + 1).$
6. $y = \sqrt{x^3 + 3x} + 2\sin x.$
7. $y = (x^2 - 2x)^3 - 4\log_3(x^4 + 1).$
8. $y = \sqrt[3]{x^4 + 3x^3} - 2\sin x$.
9. $y = 2^{x^3 + 2x} - 4\cos x$.
10. $y = 3^{x^3 - 2x} + 4\log_4(x^2 + 3).$
11. $y = \sin^{\frac{2}{5}}(3x) + \cos x$.
12. $y = \sin^{\frac{4}{7}}(3x) + 2\cos x$.
13. $y = \sqrt[5]{x^5 + 3x^3} + 2\sin x$.
14. $y = \sqrt[3]{x^5 + 5x^4} - 4x^{\frac{2}{3}}$.
15. $y = \sqrt{x^7 + 5x} + 10\sin x$.

2. ГАЛУЖЕННЯ І ЦИКЛИ

План викладу матеріалу

1. Складений оператор. Оператори галуження та вибору
2. Оператор циклу з параметром
3. Оператори циклу з умовою
4. Безумовна передача керування
5. Програми з простим повторенням
6. Програмування задач цілочислової арифметики
7. Сумування рядів

Ключові терміни розділу

- *Складений оператор*
- *Умовний оператор*
- *Оператор вибору*
- *Оператор циклу* `for`
- *Оператор циклу* `while … do`
- *Оператор циклу* `repeat … until`
- *Оператор* `break`
- *Оператор* `continue`
- *Вкладені цикли*
- *Оператор* `goto`
- *Операції цілочислової арифметики*
- *Рекуретні формули сумування рядів*

2.1. Складений оператор. Оператори галуження та вибору

Складеним оператором вважається послідовність довільних операторів, взята в операторні дужки (зарезервовані слова **`begin`** і **`end`**). Допускається довільна глибина вкладеності складених операторів. Складений оператор застосовується там, де за синтаксичними правилами мови може стояти тільки один оператор, а нам треба виконати декілька дій. У цьому випадку набір необхідних операторів має бути оформлений як складений оператор, наприклад:

```
begin a:=b+2; b:=b+1 end
```

Оператор галуження (або *умовний* оператор) забезпечує виконання деякого оператора залежно від виконання певної умови, яку задає логічний вираз:

```
if логічний_вираз then оператор;
```

Якщо значення `логічного_виразу` істинне (`true`), виконується `оператор`, а інакше виконується наступний за **`if`** оператор. Гілка

else є розширенням оператора **if** для виконання альтернативної дії:

```
if логічний_вираз then оператор1 else оператор2;
```

Якщо значення логічного_виразу істинне, виконується оператор1, а інакше виконується оператор2. За правилами кожна з гілок може містити або один оператор, або декілька, об'єднаних у складений оператор. Крапку з комою перед **else** вважають помилкою. Рекомендуємо у гілці **then** не використовувати новий оператор **if**, оскільки це може призвести до важкопередбачуваних помилкових результатів. Якщо ж все-таки без цього обійтися не можна, то оператор **if** після **then** треба взяти в операторні дужки.

Приклад 2.1. Обчислити z=max(x, y):

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

Приклад 2.2. Обчислити z=min(x, y, t).

```
if (x<y) and (x<t) then z:=x
    else if y<t then z:=y else z:=t;
```

Приклад 2.3. Обчислити $z = \begin{cases} \min(x,\ y), & \text{якщо}\, x < 0, \\ \max(x,\ y), & \text{якщо}\ x \geq 0. \end{cases}$

```
if x<0 then begin if x<y then z:=x else z:=y end
        else if x>y then z:=x else z:=y;
```

Умовний оператор дає змогу легко реалізувати вибір між двома варіантами. Однак інколи виникає необхідність вибирати один варіант з декількох (більше двох) варіантів. Це можна зробити, використовуючи вкладеність умовних операторів.

Приклад 2.4. Обчислити $y = \begin{cases} a+b, & \text{якщо}\ x < -2; \\ a-b, & \text{якщо}\ -2 \leq x \leq 3; \\ a \cdot b, & \text{якщо}\ x > 3. \end{cases}$

```
if x<-2 then y:=a+b
        else if x<=3 then y:=a-b else y:=a*b;
```

Зручним засобом вибору одного варіанта з множини можливих варіантів є оператор **case** (оператор *вибору*), який має вигляд:

```
case вираз of
     значення1: оператор1;
     значення2: оператор2;
     ...
     значенняN: операторN;
else оператор;
end;
```

Гілка **else** не обов'язкова, і на відміну від оператора **if**, перед нею можна ставити крапку з комою. Якщо значення `виразу` = `значенню1`, тоді виконується `оператор1` і виходимо з **case**. Якщо значення `виразу` = `значенню2`, тоді виконується `оператор2` і виходимо з **case** і т.д. Якщо значення `виразу` ні разу не збіглося з заданими значеннями (`значення1, значення2, …, значенняN`), тоді виконується оператор, що стоїть після **else** (або наступний оператор за **case** – за відсутності **else**). `Вираз` має бути порядкового типу. Якщо для декількох значень `виразу` дії збігаються, то ці константи можна перерахувати через кому перед двокрапкою або навіть задати діапазон.

Хоча використання варіанта **else** є не обов'язковим, досвідчені програмісти для пошуку помилок часто включають **else**, навіть коли враховані всі варіанти.

Приклад 2.5. Задати текстове значення `ocin`, яка є оцінкою.

```
case ocin of
     2: s:='не задовільно';
     3: s=:'задовільно';
     4: s=:'добре';
     5: s=:'відмінно';
else s:='Вихід за діапазон';
end;
```

Приклад 2.6. Введено ціле число. Якщо це цифра, то визначити, парна вона чи ні, а якщо число, то визначити, чи потрапляє воно у діапазон від 10 до 100, якщо ні, то видати відповідне повідомлення.

```
program chyslo;
{$APPTYPE CONSOLE}
var   i:integer;
begin
 write('Vvedit cile chyslo:'); readln(i);
 case i of
       0,2,4,6,8: writeln('Parna cyfra');
       1,3,5,7,9: writeln('Neparna cyfra');
       10...100: writeln('Chyslo vid 10 do 100');
 else writeln('Chyslo - ne cyfra i za diapazonom');
 end;
 readln
end.
```

Для виконання програми часто виникає необхідність багаторазово виконувати одну і ту ж саму послідовність дій (*цикл*). Мова Object Pascal має потужні засоби організації циклів: оператор циклу з параметром (оператор **for**) і два оператори циклу з умовою (оператор **while** і **repeat**).

2.2. Оператор циклу з параметром

Оператор циклу з параметром (оператор **for**) організовує виконання деякого оператора певне число разів. Існує дві форми оператора:

```
for параметр: = pz to kz do оператор;
for параметр: = pz downto kz do оператор;
```

Тут `параметр` циклу (лічильник) – змінна порядкового типу; `pz` і `kz` – вирази, що визначають початкове і кінцеве значення лічильника; `оператор` – один (можливо складений) оператор, який називають *тілом циклу*, що повторюється певне число разів.

На першому кроці циклу параметр набуває значення `pz`. У цей же момент відбувається обчислення `kz` – значення параметра на останньому кроці циклу. Після кожного виконання тіла циклу, якщо параметр циклу не дорівнює `kz`, відбувається зміна параметра на наступне, більше або менше значення залежно від форми

оператора **for**, тобто неявно відбувається виконання одного з двох операторів:

```
параметр: = Succ(параметр); //форма do
параметр: = Pred(параметр); //форма downto
```

У випадку pz>kz у першій формі оператора або pz<kz у другій його формі помилки немає, але цикл не буде виконаний ні разу. Після завершення роботи циклу значення параметра залишається рівним kz.

Рекомендації. Використовувати цикл **for** у разі заздалегідь відомої кількості повторень. Не змінювати параметр у тілі циклу. При використанні кратних (вкладених) циклів застосовувати різні змінні як параметри. Визначати до циклу значення всіх змінних, що використовуються у ньому. Не ставити крапку з комою після **do.**

Приклад 2.7. Обчислити значення *n*!

```
program factoria11;
{$APPTYPE CONSOLE}
var   i, n:byte; f:LongWord;
begin writeln('Vvedit naturalne chyslo:'); readln(n);
  f:=1; for i:=1 to n do f:=f*i;
  writeln('n!=', f); readln
end.
```

2.3. Оператори циклу з умовою

Якщо заздалегідь не відоме число повторень циклу, то використовують цикли з умовою, які є двох типів: **while** і **repeat**.

Цикл типу **while** є *циклом з передумовою*. Він використовується зазвичай у випадках, коли заздалегідь не відома кількість повторення циклу і можливим є випадок, коли цикл не буде виконаний жодного разу. Цикл типу **while** має вигляд:

```
while логічний_вираз do оператор;
```

До тих пір, доки значення логічного_виразу буде істинним, доти буде виконуватися один оператор (можливо складений). Оскільки значення логічного виразу перевіряється на початку кожної

ітерації, то тіло циклу може бути не виконаним ні разу. Отже, у цьому циклі логічний_вираз – це умова продовження циклу.

Інший варіант циклу з умовою – це цикл **repeat**. Його називають *циклом з постумовою*. Він має вигляд:

```
repeat оператор1;… операторN until логічний_вираз;
```

Оператор **repeat** організує виконання декількох операторів, що повторюється доти, доки не стане істинним логічний_вираз. Тіло циклу обов'язково виконається хоч би один раз. Так, у цьому циклі логічний_вираз – це умова виходу з циклу.

Приклад 2.8. Пари додатних раціональних чисел вводять з клавіатури. Обчислити добуток кожної пари і суму всіх чисел.

```
program cycle_while;
{$APPTYPE CONSOLE}
var   x, y, sum:real;   v:char;
 begin sum:=0; v:='Y';
 while (v='Y') or (v='y') do
  begin write('Vvedit chysla x i y >0');
    readln(x,y);
    if (x>0) and (y>0) then
    begin writeln('Dobutok =', x*y:8:3);
          sum:=sum+x+y  end;
    write('Nastupni (Y/N)?'); readln(v);
  end;
 writeln('Suma vsih chysel=', sum:8:3);
 readln
end.
```

Приклад 2.9. У цій задачі використаємо інший цикл з умовою.

```
program cycle_repeat;
{$APPTYPE CONSOLE}
var   x, y, sum:real;   v:char;
begin sum:=0;
 repeat write('Vvedit chysla x, y >0'); readln(x, y);
  if (x>0) and (y>0) then
```

```
  begin
    writeln('Dobutok=',x*y:8:3); sum:=sum+x+y
  end;
  write('Nastupni (Y/N)?'); readln(v)
 until (v='N') or (v='n');
 writeln('Suma vsih chysel= ', sum:8:3); readln
end.
```

Рекомендації. Для того, щоб уникнути зациклювання програми, необхідно забезпечити зміну на кожному кроці циклу значення хоча б однієї змінної, що входить в умову циклу. Після виходу з циклу зі складною умовою (з використанням операцій **and**, **or** чи **xor**), як правило, необхідна додаткова перевірка того, за якою умовою цикл завершений.

На місці оператора у циклі може стояти інший цикл (отримуємо *вкладені цикли*). В усіх трьох циклах, у яких на місці оператора є складений оператор, всередині можна записати оператор **break**. Якщо під час виконання циклу оператор **break** отримує керування, то відбувається негайний вихід з циклу і перехід на наступний оператор, який записаний за цим циклом. Процедура **continue**, яку можна записати **тільки** у тілі циклу, пропускає записані за нею оператори і переходить до наступної ітерації циклу.

2.4. Безумовна передача керування

Оператор безумовного переходу має вигляд:

```
goto <позначка>;
```

Він дає змогу передати керування безпосередньо на потрібний оператор програми. Перед цим оператором має бути розміщена *позначка* – ідентифікатор, яка відокремлена від оператора двокрапкою. Всі позначки мають бути описані в розділі оголошення позначок таким чином:

```
label <список_позначок>;
```

Конкретною позначкою у програмі може бути позначений тільки один оператор. Операторів переходу з однією і тією ж позначкою можна задати будь-яку кількість. Необхідно, щоб розділ оголошення позначок, сама позначка і оператор переходу з її вико-

ристанням були розміщені у межах одного блоку програми (див. тему *процедури і функції*). Крім того, не можна передавати керування всередину структурованих операторів (наприклад: **if**, **for**, **while**, **repeat** та ін.).

Використання позначок затруднює налагодження і тестування програми і є ознакою поганого стилю програмування. Існує лише один випадок, коли професійні програмісти допускають використання **goto**, – це вихід з набору вкладених циклів при виявленні помилок (**break** дає змогу вийти тільки з одного циклу).

2.5. Програми з простим повторенням

У цьому параграфі розглянемо задачі, для розв'язання яких треба виконати перебір певної послідовності значень, причому кожне значення використовується тільки один раз.

Для перебору значень використовуватимемо відповідний оператор циклу. Оскільки конкретне значення послідовності використовується тільки один раз на певній ітерації циклу, то заводити масиви для цих задач немає сенсу.

Приклад 2.10. Увести 10 чисел і полічити серед них кількість додатних.

```
program dodatni;
{$APPTYPE CONSOLE}
var   i, kn:byte; x:real;
begin kn:=0; // Лічильник додатних чисел
  for i:=1 to 10 do
  begin write('Vvedit ', i, ' chyslo: '); readln(x);
   if x>0 then kn:=kn+1 // Збільшуємо лічильник
  end;
  writeln('Vvely ',kn,' dodatnyh chysel'); readln
end.
```

Приклад 2.11. Вивести підряд усі букви від 'Z' до 'A'.

```
program bucvy;
{$APPTYPE CONSOLE}
var   z:char;
```

```
begin
  for z:='Z' downto 'A' do write(z);
  readln
end.
```

Приклад 2.12. Протабулювати значення функції $y=\sqrt{x^2+1}\cdot\cos x$ на проміжку $[a;\, b]$ з кроком h.

```
program Tabul;
{$APPTYPE CONSOLE}
var a, b, h, x, y:single;
begin
 write(' Vvedit mezhi a<b i krok h >0 ');
 readln ( a, b, h);
 if a>b then begin y:=a; a:=b; b:=y end;
 if h<0  then h:=-h;
 writeln(' ---------------------------- ');
 writeln(' |   x       |      y       | ');
 writeln(' ---------------------------- ');
 x:=a;
 while x<=b do
  begin
     y:=sqrt(x*x+1)*cos(x);
     write( ' |  ',  x:7:2, ' |  ');
     writeln(y:7:2, '      |  ');
     writeln(' ---------------------------- ');
     x:=x+h
  end;
 readln
end.
```

2.6. Програмування задач цілочислової арифметики

Приклад 2.13. Обчислити N-е число Фіббоначчі. Числа Фіббоначчі визначають за рекурентною формулою:

$$F(0)=F(1)=1;\ F(i)=F(i-1)+F(i-2),\ \text{якщо}\ i\geq 2.$$

```
program Fib;
{$APPTYPE CONSOLE}
var   a, b, z:word;     i, n:byte;
begin
 write('Vvedit nomer chysla Fibonachy, numeracia z 0');
 readln(N);
 a:=1; { a=F(0), a відповідає за F(i-2) }
 b:=1; { b=F(1), b відповідає за F(i-1) }
 for i:=2 to N do
  begin z:=a+b; { z відповідає за F(i) }
        a:=b; b:=z; { наступна пара чисел }
  end;
 writeln(N,' -e chyslo Fibonachy =', b); readln
end.
```

Приклад 2.14. Знаходження найбільшого спільного дільника двох натуральних чисел за допомогою алгоритму Евкліда.

```
program Evklid;
{$APPTYPE CONSOLE}
var   a, b, z:integer;
begin
 write('Vvedit 2 naturalni chysla:'); readln(a,b);
 while b<>0 do begin z:=a mod b; a:=b; b:=z end;
 writeln('Naybilshyj spilnyj dilnyk = ', a); readln
end.
```

Приклад 2.15. Дано натуральне число. Знайти кількість і суму десяткових цифр у записі цього числа.

```
program Cyfry;
{$APPTYPE CONSOLE}
var s, k:byte; a:LongWord;
begin
 s:=0; k:=0;
 write('Vvedit naturalne chyslo: ');
 readln(a);
 while a>0 do
   begin s:=s+a mod 10; inc(k); a:=a div 10 end;
```

```
 writeln('Suma cyfr =  ', s);
 writeln('Kilkist cyfr =  ', k);
 readln
end.
```

Щоб отримати цифри числа, використовуємо операцію отримання залишку (%) від цілочисельного ділення числа на 10. При першому діленні отримаємо цифру з розряду одиниць (наприклад, 167%10=7). Операція цілочисельного ділення вилучає цю цифру з розряду одиниць (167/10=16), а наступна операція отримання залишку поверне число десятків (16%10=6) і т.д. Зауважимо, що водночас отримуємо цифри числа у зворотному порядку. Цей факт можна використати при розв'занні відомої задачі про знаходження *паліндрома* – числа, величина якого не змінюється, якщо порядок цифр у його записі змінити на протилежний.

Приклад 2.16. Дано натуральне число. Перевірити, чи воно є паліндромом.

```
program Palindrom;
{$APPTYPE CONSOLE}
var s, b, a:LongWord;
begin s:=0;
 write('Vvedit naturalne chyslo: ');
 readln(a); b:=a; // копія числа
 while a>0 do
   begin s:=s*10+a mod 10; a:=a div 10 end;
 if b=s then writeln('Palindrom ')
        else writeln('Ne palindrom ');
 readln
end.
```

Подібний алгоритм використовується і в разі подання цілого десяткового числа в іншій системі числення (пригадайте правило переведення: обчислюємо остачу від ділення даного числа на нову основу, отриману частку ділимо знову на нову основу – отримуємо наступну остачу і так далі, доки чергова частка дорівнюватиме нулю. Отримані остачі у зворотному порядку формують число за новою основою).

Приклад 2.17. Дано десяткове натуральне число. Отримати його запис у вісімковій системі числення.

```
program pereved;
{$APPTYPE CONSOLE}
var a:Word; s, b:LongWord;
begin
 s:=0; b:=1;
 write('Vvedit naturalne chyslo: '); readln(a);
 while a>0 do
   begin s:=s+(a mod 8)*b; a:=a div 8; b:=b*10 end;
 writeln('8-ve chyslo=', s);   readln
end.
```

Приклад 2.18. Дано натуральне число, яке більше одиниці. Розкласти його на прості множники.

```
program Project2;
 {$APPTYPE CONSOLE}
var k:Word; a:LongWord;
begin
 write('Vvedit naturalne chyslo: ');
 readln(a); if a<=1 then a:=2;
 write(a,'='); k:=2;
 while a>1 do
   if a mod k=0
    then begin write(k,'*'); a:=a div k end
    else inc(k);
 writeln('1');   readln
end.
```

2.7. Сумування рядів

Приклад 2.19. Написати програму обчислення функції $\mathrm{Ch}\,x$ (косинус гіперболічний) за допомогою ряду Тейлора зі заданою точністю $\varepsilon > 0$ за формулою: $y = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + ... + \frac{x^{2n}}{(2n)!} + ...$

Цей ряд є збіжним при $|x| < \infty$. Для збіжного ряду модуль загального члена $u_n = \dfrac{x^{2n}}{(2n)!}$ при збільшенні n прямує до нуля. Для досягнення потрібної точності треба сумувати члени ряду, абсолютна величина яких більша за ε. Для деякого n нерівність $|u_n| \geq \varepsilon$ перестає виконуватися, і обчислення припиняють. Пряме обчислення ряду (знаходження степенів та факторіалу) є недоцільним: дуже швидко з'являються великі числа, при діленні яких втрачається точність. Для обчислення треба скористатися рекурентною формулою $u_{n+1} = u_n \times T$, де T – деякий множник. Визначимо множник T:

$$T = \frac{u_{n+1}}{u_n} = \frac{x^{2(n+1)}}{(2(n+1))!} \cdot \frac{(2n)!}{x^{2n}} = \frac{x^2}{(2n+1)(2n+2)}.$$

Заздалегідь передбачити, скільки членів ряду треба просумувати, неможливо. У циклах такого роду черезз різні причини можливе зациклення. Тому для надійності у таких програмах треба задавати величину максимальної кількості ітерації, яка слугуватиме для аварійного виходу з циклу.

```
program SumaRiadu;
 {$APPTYPE CONSOLE}
uses math;
const MaxITER=500; // Максимум ітерацій
      eps=0.000001; // точність обчислень
var x,y,u:single; n:word;
begin write('Vvedit argument: '); readln(x);
 n:=0; u:=1; y:=u; //1-ий член ряду і поч. знач. суми
 while (abs(u)>eps) and (n<MaxITER) do
  begin u:=u*x*x/((2*n+1)*(2*n+2)); y:=y+u;inc(n) end;
 writeln('Znach. nabl.= ', y:7:3);  y:=cosh(x);
 writeln('Znach. tochne= ', y:7:3);  readln
end.
```

❓ Запитання для самоперевірки

1. Які форми має умовний оператор?
2. Для розв'зання яких задач зручно використовувати оператор вибору?
3. Запишіть і охарактеризуйте оператор циклу `for`.

4. Запишіть і охарактеризуйте оператор циклу `while … do`.
5. Запишіть і охарактеризуйте оператор циклу `repeat … until`.
6. У яких випадках застосовують оператор `break`?
7. У яких випадках застосовують оператор `continue`?
8. Запишіть і охарактеризуйте оператор `goto`.

💻 Завдання для програмування

Завдання 2.1. Скласти програми для розв'язання таких задач:

1. Прочитати з клавіатури групу 12 чисел. Знайти серед них найменше та номер його першого входження у групу.
2. Прочитати з клавіатури групу 10 чисел. Знайти серед них найбільше та номер його останнього входження у групу.
3. Серед 15 чисел обчислити суму всіх від'ємних значень.
4. Серед 15 чисел визначити кількість додатних, від'ємних і нульових.
5. Ввести з клавіатури групу чисел. Перше число N>0 означає розмір чергової за ним групи чисел. Обчислити суму чисел, що належать проміжку [-3; 5].
6. Ввести з клавіатури групу чисел. Перше число N>0 означає розмір чергової за ним групи чисел. Визначити відсоток додатних чисел.
7. Прочитати з клавіатури групу 15 чисел. Знайти середнє арифметичне чисел, розміщених за порядком за першим ненульовим. Вважати, що обов'язково є хоча б одне ненульове значення, причому не на останньому місці.
8. Прочитати з клавіатури групу 15 чисел. Знайти середнє арифметичне чисел, розміщених за порядком за першим ненульовим. Вважати, що ненульових значень може не бути взагалі.
9. Прочитати з клавіатури групу 12 чисел. Знайти різницю між найбільшим і найменшим числом.
10. Обчислити суму добутків таких пар чисел: 4 – 5, 5 – 6, ..., 21 – 22.
11. Прочитати з клавіатури групу 15 чисел. Знайти перше за порядком число, що дорівнює нулю, визначивши його позицію.
12. Прочитати з клавіатури групу 15 чисел. Знайти порядковий номер першого ненульового числа.
13. Прочитати з клавіатури групу 15 чисел. Знайти порядковий номер останнього ненульового числа.
14. Прочитати з клавіатури групу, яка складається не менше як з трьох чисел. Третє за порядком число означає кількість чисел у групі, у тім числі дане. Обчислити суму всіх чисел.

15. Прочитати з клавіатури групу 10 чисел. Знайти суму модулів тих чисел, які передують першому нульовому, або усіх чисел групи, якщо нульових немає.

Завдання 2.2. Протабулювати значення функції $y = \mathrm{f}(x)$ на проміжку $[a; b]$ у n точках. Функцію вибирати з переліку функцій завдання 2.1 (див. стор. 24) згідно з номером студента у списку студентів підгрупи.

Завдання 2.3. Скласти програми розв'язання таких задач:

1. Дано натуральне число. Знайти кількість і добуток десяткових цифр у записі цього числа.
2. Дано натуральне число. Знайти середнє арифметичне цифр у записі цього числа.
3. Дано натуральне число. Знайти першу за порядком позицію (зліва направо), у якій знаходиться найменша цифра у записі цього числа. Вивести також і цю найменшу цифру.
4. Дано натуральне число. Знайти останню за порядком позицію (зліва направо), у якій знаходиться найменша цифра у записі цього числа. Вивести також і цю найменшу цифру.
5. Дано натуральне число. Знайти першу за порядком позицію (зліва направо), у якій знаходиться найбільша цифра у записі цього числа. Вивести також і цю найбільшу цифру.
6. Дано натуральне число. Знайти останню за порядком позицію (зліва направо), у якій знаходиться найбільша цифра у записі цього числа. Вивести також і цю найбільшу цифру.
7. Знайти найменше спільне кратне двох натуральних чисел.
8. Дано десяткове натуральне число. Отримати його запис у сімковій системі числення.
9. Дано вісімкове натуральне число. Отримати його запис у десятковій системі числення.
10. На проміжку $[a; b]$, де a і b – натуральні числа, вивести всі трійки Піфагорових чисел (тобто, $k^2 + l^2 = m^2$).
11. Дано десяткове натуральне число. Отримати його запис у трійковій системі числення.
12. Дано четвіркове натуральне число. Отримати його запис у десятковій системі числення.
13. Обчислити суму часток та суму остач таких виразів: $(5p^2+3)/(3p)$, p=1,...,10.
14. Обчислити середнє арифметичне остач таких виразів: $(4p^2+2)/(2p)$, p=1,...,12.

15. Дано натуральне число $m > 1$. Знайти найбільше натуральне k, при якому $2^k < m$.

Завдання 2.4. Протабулювати значення функції $y = \mathrm{f}(x)$ на проміжку $[a; b]$ у n точках. Обчислення функції реалізувати за допомогою розкладу у ряд Тейлора. Обчислені значення порівнювати зі значеннями відповідних бібліотечних функцій. Функцію вибирати з переліку функцій згідно з номером студента у списку студентів підгрупи:

1. $y = \ln\frac{x+1}{x-1} = 2\sum_{n=0}^{\infty}\frac{1}{(2n+1)x^{2n+1}}; |x| > 1.$ 2. $y = \mathrm{e}^{-x} = \sum_{n=0}^{\infty}\frac{(-1)^n \cdot x^n}{n!}; |x| < \infty.$

3 $y = \ln x = -\sum_{n=0}^{\infty}\frac{(x-1)^{n+1}}{(n+1)(x+1)^{n+1}}; \quad x > \frac{1}{2}.$

4. $y = \ln(1-x) = -\sum_{n=0}^{\infty}\frac{x^n}{n}; \quad -1 \le x < 1.$ 5. $y = \ln\frac{x+1}{1-x} = 2\sum_{n=0}^{\infty}\frac{x^{2n+1}}{2n+1}; \quad |x| < 1.$

6. $y = \mathrm{arctg}\, x = \frac{\pi}{2} + \sum_{n=0}^{\infty}\frac{(-1)^{n+1}}{(2n+1)x^{2n+1}}; |x| > 1.$

7. $y = \mathrm{arc\,ctg}\, x = \frac{\pi}{2} + \sum_{n=0}^{\infty}\frac{(-1)^{n+1} \cdot x^{2n+1}}{2n+1}; \quad |x| \le 1.$

8. $y = \mathrm{arctg}\, x = \sum_{n=0}^{\infty}\frac{(-1)^n \cdot x^{2n+1}}{2n+1}; \quad |x| \le 1.$ 9. $y = \cos x = \sum_{n=0}^{\infty}\frac{(-1)^n \cdot x^{2n}}{(2n)!}; \quad |x| < \infty.$

10. $y = \ln x = \sum_{n=0}^{\infty}\frac{(-1)^n (x-1)^{n+1}}{n+1}; 0 < x \le 2.$

11. $y = \frac{\sin x}{x} = \sum_{n=0}^{\infty}\frac{(-1)^n \cdot x^{2n}}{(2n+1)!}; \quad |x| < \infty.$

12. $y = \ln(1-x) = -\sum_{n=0}^{\infty}\frac{x^n}{n}; \quad -1 \le x < 1.$ 13. $y = \mathrm{e}^{-x} = \sum_{n=0}^{\infty}\frac{(-1)^n \cdot x^n}{n!}; |x| < \infty.$

14. $y = \ln\frac{x+1}{1-x} = 2\sum_{n=0}^{\infty}\frac{x^{2n+1}}{2n+1}; \quad |x| < 1.$ 15. $y = \mathrm{e}^{-x^2} = \sum_{n=0}^{\infty}\frac{(-1)^n x^{2n}}{(2n+1)!}; |x| < \infty.$

3. РОБОТА З РЯДКАМИ СИМВОЛІВ

План викладу матеріалу

1. Рядки символів у Паскалі
2. Типові задачі опрацювання рядків символів у Паскалі
3. Рядки символів у Object Pascal
4. Основні процедури/функції опрацювання рядків символів типу `AnsiString`

Ключові терміни розділу

- *Тип* **`String`**
- *Функція* `Copy`
- *Функція* `Delete`
- *Порівняння рядків символів*
- *Тип* `AnsiString`
- *Функція* `Length`
- *Функція* `Pos`
- *Функція* `Insert`
- *Операція зчеплення*
- *Покажчики на рядки*

3.1. Рядки символів у Паскалі

Тип **`String`** (*рядок сиволів* або просто *рядок*) у Паскалі широко використовується для опрацювання текстів. Цей тип є стандартним і багато в чому схожий на одновимірний масив символів `Array [0..N] of Char`. Значення `N` відповідає кількості символів у рядку і може змінюватися від 0 до 255. Символи, що входять у рядок, займають позиції від 1 до `N`. Початковий байт рядка (з індексом 0) містить інформацію про його довжину, тобто це символ з кодом, який дорівнює довжині рядка.

Можна також описувати змінні типу **`String`**`[k]`, де `k` – ціле число не більше за 255 (так визначають рядки з довжиною не більшою за `k`) . Цей тип вже не є стандартним.

З символами рядка можна працювати як з елементами масиву символів, але на відміну від масивів, рядки можна вводити цілком, порівнювати один з одним і зчеплювати операцією “+”.

Приклад 3.1. Використання операції зчеплення.

```
var s1, s2, x, y:string; z:char;
begin
 x:='object';  y:='pascal';
 s1:=x+' '+y; // s1='object pascal'
```

```
    s2:=''; // порожний рядок
    for z:='a' to 'z' do s2:=s2+z; // s='abcd...xyz'
    writeln(s1, '  ', s2); readln
  end.
```

Порівнюють рядки посимвольно до першого незбігу їхніх кодів. Якщо один з рядків закінчився до першого незбігу, то він вважається меншим. Порожний рядок є меншим за будь-який непорожний рядок, наприклад:

```
'abcd'>'abcD' {'d'>'D'}; 'abcd'>'abc' {'d'>''};
'abc'<'axxc' {'b'<'x'};  'abcd'='abcd'
```

Наявна низка стандартних функцій і процедур роботи з рядками:

- `Length(s: string):integer` – видає поточну довжину рядка `s`;
- `Concat(s1, s2,.., sn: string): string` – повертає рядок, який є з'єднанням рядків, тобто `s1+s2+..+sn`;
- `Copy(s: string; Index,Count:integer): string` - повертає фрагмент рядка `s`, який починається у позиції `Index` і має довжину `Count`;
- `Pos(s,s1: string):byte` – шукає перше входження підрядка `s` у рядку `s1` і повертає номер першого символу з `s` у рядку `s1` або 0, якщо такого входження немає;
- `Delete(var s: string; Index, Count:integer)` – видаляє з рядка `s` фрагмент, який починається у позиції `Index` і має довжину `Count`;
- `Insert(s: string; var s1: string; Index:integer)` – вставляє у рядок `s1` підрядок `s`, починаючи зі заданої позиції `Index`.

Мова Паскаль дає змогу перетворювати числові значення у рядки і навпаки. Процедура `str(x:n:d, s)` перетворює число `x` у рядок – зображенням цього числа, в якому є не менше `n` символів і з них `d` знаків після коми. Параметри `n` і `d` – необов'язкові. Процедура `val(s, x, e)` отримує з рядка `s` число `x`, якщо це можливо. При успішному результаті `e=0`.

3.2. Типові задачі опрацювання рядків символів у Паскалі

У мові Паскаль можливі два способи опрацювання рядків символів:

- опрацювання рядка як **єдиного цілого** (єдиного об'єкта). Цей спосіб дає можливість присвоювати рядку (видаляти чи вставляти) деяку послідовність символів за один раз;
- опрацювання рядка як масиву символів. Цей спосіб дає можливість звертатися до окремого символу рядка через його індекс, наприклад: `s[i]` – `i`-ий символ рядка `s`. Однак певну аналогію рядків з масивами символів не можна ототожнювати як повну їхню ідентичність. Так, найбільш поширеною помилкою є робота з елементами рядка без урахування його поточної довжини.

Зауважимо, що звернутися до `i`-го символу рядка `s` можна і за допомогою першого способу, а саме: `Copy(s, i, 1)`.

Далі подаємо розв'язання деяких типових задач оброблення рядків символів у Паскалі.

Приклад 3.2. Підрахувати кількість повторення у рядку символу `'x'`.

```
var s:string; i,k:byte;
begin
 write('Enter s'); readln(s);
 k:=0; for i:=1 to length(s) do
       if s[i]='x' then inc(k);
 write('k=',k);  readln
end.
```

Приклад 3.3. Замінити у рядку кожен символ `'!'` на символ `'.'`.

```
var s:string; i:byte;
begin
   write('Enter s'); readln(s);
   for i:=1 to length(s) do
    if s[i]='!' then s[i]:='.';
   write(s); readln
end.
```

Приклад 3.4. Замінити у рядку кожен символ `'.'` на `'...'`.

```
  var s:string; i:byte;
```

```
begin  write('Enter s');
 readln(s); i:=1;
 While i<=length(s) do
  if Copy(s,i,1)='.'
   then begin Insert('..',s,i+1);
              i:=i+3 end
   else inc(i);
 write(s); readln
end.
```

Приклад 3.5. Замінити у рядку всі `'...'` на `'.'`.

```
var s:string;
begin
 write('Enter s'); readln(s);
 While pos('...',s)<>0 do
       delete(s,pos('...',s),2);
 write(s); readln
end.
```

Приклад 3.6. Вияснити, чи трапляються у рядку символи `',-'`.

```
var s:string; i:byte;
begin
 write('Enter s'); readln(s);
 i:=pos(',-',s);
 if i=0 then write('No')
        else write('Yes');  readln
end.
```

Приклад 3.7. Відомо, що у рядку є хоча б один символ `','` (кома). Знайти номер першого і останнього входження `','` у рядок.

```
var s:string; i,k,m:byte;
begin
 write('Enter s: '); readln(s);
 k:=pos(',',s); i:=length(s); m:=0;
 while (i>=1) and (m=0) do
  begin if copy(s,i,1)=',' then m:=i;
        dec(i)
  end;
 write('1 vh=',k,' Ost.vh=',l);readln
```

```
end.
```

Приклад 3.8. Видалити у рядку всі початкові пропуски.

```
var s:string; i:byte;
begin
 write('Enter s'); readln(s);
 While pos(' ',s)=1 do delete(s,1,1);
 write('ww'+s); readln
end.
```

Приклад 3.9. Видалити у рядку всі кінцеві пропуски.

```
var s:string;
begin
 write('Enter s: '); readln(s);
 while copy(s,length(s),1)=' ' do
            delete(s,length(s),1);
 write(s+'ww'); readln
end.
```

Приклад 3.10. Порахувати у рядку усі букви 'b' і видалити ті з них, перед якими стоїть буква 'a'.

```
var s:string; c:char; i,k:byte;
begin write('Enter s: '); readln(s);
 c:=chr(0); // позначка видалення
 s:=' '+s; k:=0;
 for i:=2 to length(s) do
  if s[i]='b' then
   begin inc(k);
    if s[i-1]='a' then s[i]:=c
   end;
 while pos(c,s)<>0 do
         delete(s,pos(c,s),1);
 write(s); write('k=', k); readln
end.
```

Приклад 3.11. Вияснити, чи рядок s є паліндромом (без урахування пропусків).

```
var s,s1:string; i,n:byte;
begin write('Enter s1: '); readln(s1);
```

```
  s:=s1;
  while pos(' ',s)<>0 do
        delete(s,pos(' ',s),1);
  i:=1; n:=length(s);
  while (i<=n div 2)
    and (s[i]=s[n-i+1]) do inc(i);
  if i<=n div 2 then write('Ne ');
  write('Palindrom'); readln
end.
```

3.3. Рядки символів у Object Pascal

Мова Object Pascal має декілька різних типів рядків, так що можливий вибір того чи іншого типу рядка залежно від ситуації:

- **`AnsiString`** – основний тип рядка у Object Pascal, який містить символи `AnsiChar` і теоретично не має обмежень у довжині. Цей тип сумісний з рядками, що мають останній нульовий символ;
- **`ShortString`** – залишився у мові для зворотної сумісності (відповідає типу `String` у стандарті Паскаля). Його максимальна довжина – 255 символів;
- **`WideString`** – за своєю функціональністю подібний до типу `AnsiString` з тою лише різницею, що він містить символи типу `WideChar`;
- **`PChar`** – покажчик на рядок символів типу `Char` з останнім нульовим символом (аналог `char*` або `lpstr` у мові C);
- **`PAnsiChar`** – покажчик на рядок символів типу `AnsiChar` з останнім нульовим символом;
- **`PWideChar`** – покажчик на рядок символів типу `WideChar` з останнім нульовим символом.

За домовленістю при описі змінних за допомогою родового типу `string` компілятор створює рядок типу `AnsiString`, як-от:

```
Var S: string; // змінна s типу AnsiString
```

Для зміни опису типу рядка за домовленістю використовують директиву компілятора `$H`. Її додатне значення (включене за до-

мовленістю) визначає `AnsiString`, а від'ємне – `ShortString`, наприклад:

```
var {$H+} S1: string; // змінна S1 типу AnsiString
    {$H-} S2: string; // змінна S2 типу ShortString
```

Виключенням з цього правила є рядок з визначеним у описі розміром (не більше 255). Такий рядок завжди використовує тип `ShortString`, наприклад:

```
var S: string[63]; // змінна s типу ShortString
                   // може містити до 63 символів
```

Тип `AnsiString`, відомий як "*довгий рядок*", був започаткований у Delphi 2.0 у відповідь на вимогу користувачів скасувати 255-символьне обмеження на довжину рядка.

Хоча у використанні тип `AnsiString` нічим не відрізняється від типу `string` у стандартному Паскалі, пам'ять для нього виділяється динамічно (у ході виконання програми), а для її звільнення використовується технологія "*garbage collect*". Завдяки цьому `AnsiString` є типом з керованим часом життя (*lifetime managed*). Object Pascal сам автоматично виділяє пам'ять для тимчасових рядків, тому програміст не піклується про це, як у мовах C/C++/C.

Змінна типу `AnsiString` є покажчиком на динамічно виділену ділянку пам'яті, у якій зберігається рядок. Тому може бути декілька змінних типу `AnsiString`, які показують на один і той самий рядок. Тому у виділеній ділянці пам'яті зберігається сам *рядок*, його *поточна довжина* і *лічильник покажчиків* на рядок. *Нульовий* останній *символ* у виділеній ділянці записується автоматично.

Одразу після оголошення рядок типу `AnsiString` не має довжини, а, значить, не займає пам'яті. Для виділення пам'яті під рядок йому потрібно присвоїти деяке значення, присвоїти інший покажчик або використати процедуру `SetLength()`, як показано нижче:

```
var s:string;// опис рядка, без виділення пам'яті
...
begin
...
```

```
  s:='значення'; // виділення необхідної пам'яті
 {або}
  s:=otherString; // збільшення лічильника otherString
 {або}
  setLength(s,4);// виділення пам'яті для 4-х символів
 ...
```

Треба бути обережним з індексацією елементів рядка:

```
 var s:string;
 begin s[1]:='k';// Помилка - пам'ять ще не виділена
 ...
```

Цей приклад працює коректно:

```
 var s:string;
 begin setLength(s,5); s[1]:='k';// Все, як звичайно
 ...
```

Зауважимо, що індекси у рядках типу `AnsiString` **відраховуються від 1** (на відміну від інших типів довгих рядків, у яких індекси відраховуються від 0).

Завдяки наявності *лічильника покажчиків* на рядок операція копіювання зводиться до копіювання покажчика і збільшення лічильника на одиницю. Реальне копіювання відбувається тільки в необхідних випадках, наприклад:

```
var s1, s2: string;
begin
 s1:='рядок'; // покажчик s1; лічильник=1
 s2:=s1; // покажчик s2 = покажчику s1; лічильник=2
 s2:=s2 + ' змінюється тільки зараз';
 // новий покажчик s2 показує на нову ділянку, в яку
 // копіюється зміст ділянки s1 + нове значення
 // лічильники у s1 і s2 дорівнюють 1
...
```

Якщо значення *лічильника покажчиків* на рядок стає рівним нулю, то виділена ділянка пам’яті, у якій зберігається рядок, звільняється (рядок знищується). Якщо за допомогою індексного звертання у рядку змінюється тільки один символ, то рядок копіюється тільки в тому випадку, якщо *лічильник покажчиків* містить число,

яке більше за 1. Все це дає змогу ефективно використовувати пам'ять.

Під час роботи з рядками, що мають нульовий символ у кінці, часто використовують спеціальні типи покажчиків (`Pchar`, `PAnsiChar` і `PwideChar`). Змінні цих типів часто треба передавати у різні функції та процедури як арґументи.

Покажчик `PChar` забезпечує сумісність з Delphi 1.0 і роботу з Win32 API. Пам'ять для `PChar` не виділяється (і не звільняється) автоматично. Отже, використовуючи цей тип, програміст виконує операцію самостійно. Довжина `PChar` може досягати 4 ГБ. Зазвичай виділяють пам'ять за допомогою функції `StrAlloc()`, проте можливе застосування таких функцій, як: `AllocMem()`, `GetMem()`, `StrNew()` або навіть функції API `VirtualAlloc()`.

Рядок типу `AnsiString` завжди закінчується нульовим символом. Тому при роботі з функціями Win32 API або іншими, що мають параметри типу `Pchar`, достатньо перетворити тип `AnsiString` у тип `PChar`. Наступний приклад демонструє виклик функції Win32 `GetWindowsDirectory()`, яка своїми параметрами має покажчик на буфер типу `PChar` і його розмір:

```
var s:string;
begin setLength(s, 256); // пам'ять для буфера
getWindowsDirectory(PChar(s), 256);
...
```

3.4. Основні процедури/функції опрацювання рядків символів типу AnsiString

Наведемо визначення найбільш вживаних процедур/функцій опрацювання рядків сиволів типу `AnsiString`:

- **`AnsiCompareStr`**`(const s1, s2:`**`String`**`):integer` – порівнює два рядки `s1` і `s2` з урахуванням регістра. Повертає значення > 0 при `s1` > `s2`, значення < 0 при `s1` < `s2` і значення 0 при `s1` = `s2`. Може працювати з кириличними символами, на відміну від подібної функції **`CompareStr`**;
- **`AnsiCompareText`**`(const s1, s2:`**`String`**`):integer` – працює аналогічно функції **`AnsiCompareStr`**, але без урахування ре-

гістра. Може працювати з кириличними символами, на відміну від функції **CompareText**;

- **AnsiLowerCase**(s: **String**)**:String** – повертає новий рядок, в якому всі символи приведені до нижнього регістра. Не впливає на початковий рядок s. Може працювати з кириличними символами, на відміну від подібної функції **LowerCase**;
- **AnsiUpperCase**(s: **String**)**:String** – повертає новий рядок, в якому всі символи приведені до верхнього регістра. Не впливає на початковий рядок s. Може працювати з кириличними символами, на відміну від подібної функції **UpperCase**;
- **Concat**(s1, s2,.., sn: **String**)**:String** – повертає рядок, який є з'єднанням рядків, тобто s1+s2+..+sn;
- **Copy**(s: **String**; Index,Count:integer)**:String** – повертає підрядок s, який починається у позиції Index і має довжину Count;
- **CurrToStr**(value: Currency)**:String** – перетворює значення value типу **Currency** у рядок;
- **Delete**(**var** s: **String**; Index, Count: integer) – видаляє з рядка s фрагмент, який починається у позиції Index і має довжину Count;
- **FloatToStr**(value: Extended)**:String** – перетворює значення value раціонального типу у рядок з точністю до 15 цифр;
- **FloatToStrF**(value: Extended; Format: TFloatFormat; precision, digits: integer)**:String** – перетворює раціональне значення value у рядок, використовуючи заданий Format. Параметри precision і digits задають точність і число розрядів. Тип TfloatFormat визначає коди форматування раціональних чисел. Можливі значення формату визначають такі **правила форматування**:

  - ffGeneral – узагальнений числовий формат. Число перетворюється за форматом *фіксованої крапки* або *науковим*. Початкові нулі знищуються, десяткова крапка ставиться тільки в разі потреби. Формат *фіксованої крапки* використовується, якщо кількість розрядів зліва від десяткової крапки не перевищує заданої точності *Precision* і коли саме числове значення не менше 0.00001. У іншому випадку використовується *науковий формат*, в якому параметр *Digits* визначає мінімальне число розрядів степеня – від 0 до 4;

- `ffExponent` – науковий формат. Число перетворюється у рядок символів вигляду `'-d.ddd...E+dddd'`, де `d` – означає цифру. Загальна кількість цифр, включаючи одну перед десятковою крапкою, задається параметром *Precision*. Після символу `'E'` завжди слідує знак "+" або "-" і до чотирьох цифр. Параметр *Digits* визначає мінімальне число розрядів степеня – від 0 до 4;

- `ffFixed` – формат з фіксованою крапкою. Число перетворюється у рядок символів вигляду `'-ddd.ddd...'`. Принаймні одна цифра завжди є перед десятковою крапкою. Число цифр після десяткової крапки задається параметром `Digits`, значення якого може лежати у межах від 0 до 18. Якщо число розрядів зліва від десяткової крапки більше вказаного параметром `Precision`, то використовується науковий формат;

- `ffNumber` – числовий формат. Число перетворюється у рядок вигляду `'-d ddd ddd.ddd...'`. Цей формат збігається з `ffFixed` за винятком наявності у ньому роздільників тисяч;

- `ffCurrency` – монетарний формат. Число перетворюється у рядок, що відображає грошову суму. Формат контролюється глобальними змінними `CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator,` які задаються на закладці *Currency Format* (Грошова одиниця) елемента *International* (Мова і стандарти) контрольної панелі Windows. Число цифр після десяткової крапки задається параметром *Digits*, який може лежати в межах від 0 до 18.

- **Insert**`(s: string; var s1: string; Index:integer)` – вставляє у рядок `s1` підрядок `s`, починаючи зі заданої позиції `Index`;
- **IntToHex**`(Value, Digits: integer)`**:String** – перетворення цілого числа/числового виразу `Value` у рядок, який має щонайменше `Digits` шістнадцяткових цифр;
- **IntToStr**`(Value: integer)`**:String** – перетворення цілого числа/числового виразу `Value` у рядок;
- **Length**`(s:` **String**`):integer` – видає поточну довжину рядка `s`;
- **Pos**`(s,s1:` **String**`):byte` – шукає перше входження підрядка s у рядку `s1` і повертає номер першого символу з `s` у рядку `s1` або 0, якщо такого входження немає;

- **str**(x:n:d; **var** s: **String**) – перетворює число x у рядок s – зображенням цього числа, в якому є не менше n символів і з них d знаків після коми. Параметри n і d – необов'язкові;
- **StrToCurr**(const s: **String**):Currency – перетворення рядка символів s у число типу Currency;
- **StrToFloat**(const s: **String**):Extended – перетворення рядка символів s у число раціонального типу;
- **StrToInt**(const s: **String**):Integer – перетворення рядка символів s у число цілого типу;
- **StrToIntDef**(const s: **String**; def: Integer):Integer – перетворення рядка символів s у число цілого типу. Якщо рядок не допускає перетворення, то за домовленістю повертається def;
- **Trim**(const s: **String**): **String** – повертає рядок, який відповідає заданому, але без символів пропуску і символів керування до і після значущих символів;
- **TrimLeft**(const s: **String**):**String** – повертає рядок, який відповідає заданому, але без символів пропуску і символів керування до значущих символів;
- **TrimRight**(const s: **String**):**String** – повертає рядок, який відповідає заданому, але без символів пропуску і символів керування після значущих символів;
- **Val**(s: **String**; **var** x: Integer; **var** e: Integer) – отримує з рядка s ціле число x, якщо це можливо. При успішному результаті e=0.

У разі виникнення помилок під час виконання функцій перетворення рядків у числа генерується виключення EconvertError. Це не стосується фунції **StrToIntDef**, яка при виникненні помилки перетворення заносить у результат вказане значення за домовленістю.

- **? Запитання для самоперевірки**

1. Охарактеризуйте тип String.
2. Що робить функція Length?
3. Що робить функція Copy?
4. Що робить функція Pos?
5. Що робить процедура Delete?
6. Що робить процедура Insert?

7. Як порівнюються рядки символів?
8. Як виконується операція зчеплення рядків?
9. Охарактеризуйте тип `AnsiString`.
10. Що таке лічильник покажчиків на рядок?

• 💻 Завдання для програмування

Завдання 3.1. Скласти програми для розв'язання таких задач:

1. Відомо, що у рядку є хоча б один символ `'″'` (лапки). Знайти номер першого входження `'″'` у рядок і порахувати кількість цих входжень.
2. Відомо, що у рядку є хоча б один символ `'″'` (лапки). Знайти номер останнього входження `'″'` у рядок.
3. Відомо, що у рядку є хоча б один символ `'″'` (лапки). Знайти кількість входжень `'″'` у рядок і додати цей символ у кінець рядка.
4. Замінити у рядку кожен символ `'#'` на символ `'*'` і порахувати кількість таких замін.
5. Відомо, що у рядку є хоча б одна послідовність символів `';%'`. Замінити у рядку кожну таку послідовність символом `'*'`.
6. Прочитати з клавіатури два рядки. Якщо вони однакові (за довжиною та символами), то надрукувати слово `'Yes'`, інакше – `'No'`.
7. Прочитати з клавіатури рядок символів. Усі символи `'A'` замінити на `'a'`; усі символи `'B'` замінити на `'b'`.
8. Перевірити, чи збігаються у рядку перші та останні чотири літери.
9. Задано деякий рядок з 14-ти символів, і відомо, що в ньому є хоча б один символ `'*'`. Вияснити, в якій половині рядка трапляється вперше зліва ця літера.
10. Останній у рядку символ `'+'` змінити на символ `'–'`.
11. Прочитати з клавіатури рядок символів. Зробити його копію, замінивши водночас букви `'R'`, `'S'`, `'T'` на `'K'`, `'L'`, `'M'` відповідно.
12. Прочитати з клавіатури рядок символів. Перелічити в ньому окремо кількість символів `'('` та `')'`.
13. Порахувати у рядку усі послідовності символів `'cd'` і видалити ті з них, перед якими стоїть буква `'b'`.
14. Порахувати у рядку усі послідовності символів `'cd'` і видалити ті з них, після яких стоїть буква `'b'`.
15. Замінити у рядку всі `'@@@'` на `'##'`.

Завдання 3.2.

Термінологія. *Текст* – довільна послідовність символів, що знаходиться у рядку (`string`). *Слова* у тексті розділяються пропусками. *Речення* – послідовність слів, що завершується крапкою (знаком оклику чи запитання). Скласти програми для розв'язання таких задач:

1. Визначити у тексті максимальну довжину послідовності символів, що не є буквами.
2. Знайти максимальну за довжиною монотонну (неспадну або незростаючу) підпослідовність натуральних чисел, заданих у тексті так, що числа відокремлюються пропусками.
3. Визначити всі слова тексту, що складаються з тих самих букв, що і перше слово цього тексту.
4. У тексті знайти пари слів, що є дзеркальними відображеннями (типу `'ole'` і `'elo'`).
5. У тексті знайти всі симетричні слова (типу `'oko'`, `'ABBA'` і т.п.).
6. Знайти всі слова, що трапляються у кожному з двох речень тексту.
7. Відредагуйте деякий текст, видаливши у ньому всі слова з непарними номерами і перевертаючи слова з парними номерами (типу `'How do you do?'` ⇒ `'od od?'`).
8. Для кожного символу тексту вказати, скільки разів він трапляється у тексті. Повідомлення про символ виводити тільки один раз.
9. Деякий текст має два речення. Знайти найкоротше слово першого рядка, якого нема у другому реченні.
10. Відредагуйте деякий текст, видаливши у ньому всі слова, що мають у своєму складі входження тільки двох букв (типу `'Akka ababa cece knopka.'` ⇒ `'knopka.'`).
11. Замінити в англомовному тексті закінчення слів `'ing'` на `'ed'`, стиснувши водночас текст.
12. В англомовному тексті знайти слово, в якому кількість голосних букв (*a*, *e*, *i*, *o*, *u*) є найбільшою.
13. Характеристика слова – кількість *різних* символів, що у нього входять. Впорядкувати слова у тексті за спаданням їхніх характеристик.
14. Відстань між двома словами *однакової* довжини – це кількість позицій, у яких стоять різні символи. В деякому тексті знайти пару найбільш віддалених слів заданої довжини.
15. Визначити всі слова тексту, що складаються з тих самих букв, що і останнє слово цього тексту.

4. СТРУКТУРОВАНІ ТИПИ ДАНИХ

План викладу матеріалу

1. Масив. Опис масиву
2. Одновимірні масиви (вектори)
3. Алгоритми пошуку в одновимірному масиві
4. Алгоритми сортування в одновимірному масиві
5. Матричні задачі
6. Записи (комбіновані типи)
7. Множини
8. Типізовані константи
9. Тип `Variant` у Object Pascal

Ключові терміни розділу

- *Поняття масиву*
- *Розмірність масиву*
- *Лінійний пошук*
- *Бінарний (двійковий) пошук*
- *Сортування обміном*
- *Типові матричні задачі*
- *Опис запису. Поля*
- *Поняття множини*
- *Операції над множинами*
- *Опис типізованої константи*
- *Тип масиву. Опис масиву*
- *Одновимірний масив*
- *Пошук з бар'єром*
- *Сортування вибором*
- *Сортування включенням*
- *Поняття запису*
- *Оператор приєднання*
- *Опис множини*
- *Поняття типізованої константи*
- *Тип* `Variant`

4.1. Масив. Опис масиву

Масив – це **впорядкована структура** однотипних даних, яка зберігає ці дані у пам'яті комп'ютера послідовно. Доступ до елемента масиву здійснюється через його індекси. **Тип масиву** описують таким чином:

```
назва_типу_масиву = array [діапазони_індексів]
                          of тип_елемента_масиву;
```

Як `тип_елемента_масиву` можна використати будь-який тип Паскаля крім файлового. `Діапазони_індексів` – це один діапазон або декілька діапазонів, які перераховані через кому. Як `діапазони_індексів` не можна використати діапазони з базовими типами `LonWord` чи `Int64`.

Після опису типу масиву у розділі **var** можна описати змінні цього типу (масиви). Тип масиву можна вказувати і безпосередньо при описі змінних-масивів (у розділі **var**), але тоді програємо у наглядності та універсальності.

Приклад 4.1. Три способи опису одного і того ж типу масиву:

```
type M1 = array [0..5] of integer;
     M2 = array [char] of M1;
     M3 = array [-2..2] of M2; {1}
     M3 = array [-2..2] of array [char] of array
                                [0..5] of integer; {2}
     M3 = array [-2..2, char, 0..5] of integer; {3}
```

Опис масиву A за допомогою типу M3:

```
     var A:M3;
```

Безпосереднє задання типу масиву:

```
     var A: array [-2..2, char, 0..5] of integer;
```

Звертатися до елементів масиву можна таким чином:

```
     read(A[- 1,'a', 3]); read(A[1]['x'][0]);
     A[1]['c', 1]:=100;
     …
```

Кількість діапазонів в описі типу масиву (кількість індексів) визначає **розмірність** масиву. У прикладі 1 тип M1 визначає одновимірний масив, M2 – двовимірний масив, а M3 – тривимірний масив. Одновимірні масиви, за аналогією з термінами лінійної алгебри, часто називають **векторами**, а двовимірні – **матрицями** (перший індекс задає номер рядка, а другий – номер стовпця). Розмірність при визначенні масивів може бути довільною (на практиці – не більше трьох).

Приклад 4.2. Опис типу вектора і матриці:

```
type vector = array [0..5] of integer;
     matrix = array [1..4, 1..5] of real;
```

Для описаних типів можна визначити, наприклад, такі змінні:

```
     var v1, v2:vector; m1, m2, m3: matrix;
```

У Object Pascal за допомогою одного оператора присвоювання можна передати копії всіх елементів одного масиву (у правій частині) іншому масивові (у лівій частині) такого ж типу, наприклад:

```
v1:= v2; m1:= m2; m3:= m2;
```

З іншого боку, оголошення

```
var w1: array [1..8] of real;
    w2: array [1..8] of real;
```

створює **різні** типи масивів, а тому оператор

```
w1:= w2;
```

викличе повідомлення про помилку.

Інші операції над масивами виконуються тільки поелементно.

Елементи масивів можуть мати довільний тип, крім файлового типу. Наприклад, оголошення

```
var s1: array [1..10] of char;
```

визначає масив символів (фактично, **рядок** символів). З масивом символів у багатьох випадках можна працювати як з рядком фіксованої довжини, наприклад:

```
s1:= 'abcdefghij';
```

Наведений вище оператор присвоювання не спричинятиме помилок, якщо кількість сиволів у рядку не перевищуватиме 10.

Оголошення змінних чи типізованих констант типу масиву можна поєднувати з заданням елементам масиву початкових значень. Список значень з кожного виміру беруть у круглі дужки, а самі значення і списки розділяють комами.

Приклад 4.3. Оголошення змінних і типізованих констант типу масиву з одночасним заданням елементам масиву початкових значень:

```
type matr = array [1..3, 1..4] of byte;
var a1: array [1..6] of integer=(1, 2, 3, 4, 5, 6);
    a2: matr = ((11, 12, 13, 14),
                (21, 22, 23, 24),
                (31, 32, 33, 34));
const a3: array [1..4] of integer =(10, 20, 30, 40);
```

4.2. Одновимірні масиви (*вектори*)

Над одновимірним масивом виконують дві базові операції: *отримують* деякий елемент з масиву і *розміщують* елемент у масиві (порівняння і присвоювання). Паскаль не має засобів уведення/виведення всіх елементів масиву відразу, тому введення і виве-

дення значень здійснюють поелементно. *Ініціалізують* масиви через введення даних, присвоювання значень і задання початкових значень при оголошенні масиву.

Іноді в тексті навчальної програми введення займає більше місця, чим запис алгоритму рішення, що утруднює сприйняття програми. Тому в прикладах цього параграфа ми часто будемо опускати ініціалізацію масиву, а користуватимемося тільки математичним формулюванням задачі типу: “Дано натуральне число `n` і дійсні числа a_1, a_2,..., a_n. Знайти …”.

У прикладних програмах введення, як правило, виконують або з клавіатури, або з файлу. Зупинимося докладніше на введенні елементів масиву і їхнього числа `n` із клавіатури. У Паскалі кількість *елементів масиву, їхню впорядкованість і тип задають явно до початку виконання програми,* Тому, якщо границі масиву точно невідомі, то їх вибирають “із запасом” (це стосується переважно верхньої межі діапазону індекса). Наприклад, після опису

```
a: Array [1.. 100] of Real;
```

уведене `n` має належати діапазону `1..100`, а після опису

```
a: Array [Byte] of Real;
```

діапазону типу `Byte` (від 0 до 255). Зазвичай верхню межу діапазону індекса описують у вигляді константи у розділі **`const`**.

Є велика кількість розмаїтих задач на одновимірні масиви. Як і всі задачі взагалі, умовно їх можна розділити на три типи:

1) прості задачі, які розв’язують в “одне розуміння”;
2) стандартні задачі;
3) задачі, рішення яких потребує знання допоміжних алгоритмів, спеціальних методів і прийомів.

Очевидно, що без уміння вирішувати задачі перших двох типів неможливо вирішувати нестандартні задачі. Розглянемо кілька прикладів задач першого типу.

Приклад 4.4. У масиві *a* кожний елемент дорівнює 0 чи 1. Замінити всі нулі одиницями і навпаки.

Досить одного оператора присвоювання у тілі циклу:

```
for i:= 1 to n do a[i]:=1-a[i];
```

Приклад 4.5. У масиві кожен елемент дорівнює 0, 1 чи 2. Переставити елементи масиву так, щоб спочатку розміщувалися всі 0, потім усі 1 і, нарешті, 2. Додаткового масиву не заводити.
Можна не переставляти елементи масиву, а підрахувати кількості 0, 1 і заповнити масив заново у потрібний спосіб:

```
k0:=0; k1:=0;
for i:= 1 to n do
 if a[i] = 0
  then k0:=k0+1 else if a[i] = 1 then k1:=k1+1;
for i:= 1 to n do a[i]:=2;
for i:= 1 to k0 do a[i]:=0;
for i:= k0+1 to k0+k1 do a[i]:=1;
```

Перелічимо **стандартні** задачі на одновимірні масиви:

1) знаходження найбільшого/найменшого елемента;
2) знаходження суми/добутку елементів (безумовне й умовне);
3) підрахунок/заміна елементів, що задовольняють заданій умові;
4) пошук заданого елемента:
   а) у неупорядкованому масиві; б) в упорядкованому масиві;
5) визначення заданого розміщення елементів;
6) видалення елемента, включення елемента в задану позицію;
7) перерозміщення (інвертування, циклічне зсування) елементів;
8) випадкова вибірка елементів (з повтореннями і без повторень);
9) злиття двох упорядкованих масивів в один упорядкований масив;
10) сортуваня масиву (прості методи).

Задачі 1 – 3 і 5 – 9 будуть розглянуті в прикладах цього пункту. Задачам 4 і 10 відведемо окремі пункти. До складних задач можна віднести поліпшені методи сортування масивів, пошук моди і медіани масиву; алгоритми генерування комбінаторних об’єктів, алгоритми на графах і т.д. Їх можна знайти у [6, 7, 21] і інших.

До стандартних задач віднесемо і задачу *обчислення значення многочлена* у деякій точці x, оскільки коефіцієнти многочлена зазвичай зберігають в одновимірному масиві.

Приклад 4.6. Обчислення значення многочлена степеня `n` у точці x, коефіцієнти якого знаходяться в масиві A, за схемою Горнера:

$$P_n(x) = A[0]\cdot x\text{^}n + A[1]\cdot x\text{^}(n-1) +. .. + A[n-1]\cdot x + A[n] =$$
$$= (...((A[0]\cdot x + A[1])\cdot x + A[2])\cdot x +. .. + A[n\text{-}1])\cdot x + A[n].$$

```
program Scheme_Gorner;
{$APPTYPE CONSOLE}
const nmax = 20;
var A: array [0..nmax] of integer;
i, j, n: integer; x, p: real;
begin
 write('Stepin mnohochlena = ');   read(n);
 writeln(' Cili koeficienty: ');
 for i:=0 to n do read(A[i]);
 write(' Znachennia x = ');  readln(x);
 p:=0;
 for i:=0 to n do p:=p*x+A[i];
 write(' Pn(x) = ', p); readln
end.
```

Приклад 4.7. Дано натуральне число n і послідовність дійсних чисел a_1, a_2,..., a_n. Знайти перше від початку послідовності найменше число a_i (i=1..n) і останнє за порядком у послідовності найбільше число a_j (j=1..n) та поміняти їх місцями.

Для рішення задачі достатньо знайти *порядкові номери* першого від початку найменшого числа і останнього за порядком найбільшого числа послідовності. У назві програми відображено номер задачі у переліку стадартних задач.

```
program Standart_1;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of real;
 x: real; i, n, kmin, kmax: byte;
begin
 repeat write('Kilkist chysel vid 1 do ',nmax,': ');read(n);
 until (n>=1) and (n<=nmax);
 writeln(' Chysla: ');
 for i:=1 to n do read(a[i]);
 kmin:=1; kmax:=1;
```

```
  for i:=2 to n do
   begin
    if a(kmin)>a[i] then begin kmin:=i end;
    if a(kmax)<=a[i] then begin kmax:=i end
   end;
  x:= a[kmin]; a[kmin]:= a[kmax]; a[kmax]:=x;
  for i:=1 to n do write(a[i]:6:2,'  ');
  readln
 end.
```

Приклад 4.8. Дано натуральне число n і послідовність дійсних чисел a_1, a_2,..., a_n. Знайти суму додатних чисел і середнє арифметичне від'ємних чисел. Значення суми присвоїти першому елементу масиву, а середнє арифметичне – останньому елементу масиву.

```
 program Standart_2;
 {$APPTYPE CONSOLE}
 const nmax = 30;
 var a: array [1..nmax] of real;
     sd, sv: real; i, n, nv: byte;
 begin  // Введення n та елементів масиву - див. приклад 4.7!
  sd:=0; sv:=0; nv:=0;
  for i:=1 to n do
   begin if a[i]>0 then sd:=sd+a[i];
    if a[i]<0 then begin sv:=sv+a[i]; inc(nv) end
   end;
  a[1]:=sd;
  if nv=0 then writeln('Vidyemnyh nema') else a[n]:=sv/nv;
  { Виведення масиву a - див. приклад 4.7! } readln
 end.
```

Приклад 4.9. Дано натуральне число n і послідовність натуральних чисел a_1, a_2,..., a_n. Замінити всі члени послідовності a_1, a_2,..., a_n, які більші від 100, числом 101. Визначити кількість таких замін.

```
program Standart_3_1;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of word; i, n, n100: byte;
```

```
begin  // Уведення n та елементів масиву - див. приклад 4.7!
 n100:=0;
 for i:=1 to n do
  if a[i]>100 then begin a[i]:=101; inc(n100) end;
 writeln('Zrobleno ',n100, ' zamin!');
 { Виведення масиву a - див. приклад 4.7!}  readln
end.
```

Приклад 4.10. Дано натуральне число n і послідовність цілих чисел a_1, a_2,..., a_n. Визначити, скільки разів трапляється у цій послідовності найменше число. Замінити цим найменшим числом відповідну кількість елементів від початку масиву.

```
program Standart_3_2;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of integer; min: integer;
    i, n, nmin: byte;
begin  // Уведення n та елементів масиву - див. приклад 4.7!
 min:= a[1]; nmin:=1;
 for i:=2 to n do
  if a[i]<min then begin min:=a[i]; nmin:=1 end
                   else if a[i]=min then inc(nmin);
 for i:=1 to nmin do a[i]:=min;
 { Виведення масиву a - див. приклад 4.7!} readln
end.
```

Якщо умова a[i]<min справджується, то знайдено **нове** найменше число, і відлік кількості повторень найменшого числа треба починати з 1 (*скидання накопиченого значення* кількості повторень). Якщо ж умова a[i]<min не справджується, то з двох можливих значень (a[i]=min чи a[i]>min) нас буде цікавити одне: a[i]=min – означає, що знайдено елемент, який збігається з поточним найменшим числом (збільшуємо лічильник кількості повторень).

Скидання накопиченого значення деякої величини – досить поширений прийом, який використовують при розв'язуванні багатьох задач.

Приклад 4.11. Дано натуральне число n і послідовність цілих чисел a_1, a_2,..., a_n. Визначити найбільшу кількість однакових чисел, які розміщені поряд.

```
program Standart_5_1;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of integer; i, n, k, q: byte;
begin  // Введення n та елементів масиву - див. приклад 4.7!
       // Послідовність однакових чисел назвемо серією
 k:=1; // Кількість елементів поточної серії
 q:=1; // Шукана величина
 for i:=2 to n do
  if a[i]=a[i-1] then inc(k) // Продовження/початок серії
                 else        // Кінець серії
                  begin if q<k then q:=k; //Формування max
                        k:=1 // Нова ініціалізація
                  end;
 if q<k then q:=k; // Врахування останньої серії
 writeln('Max povtoriv ', q); readln
end.
```

Приклад 4.12. Дано натуральне число n і послідовність цілих чисел a_1, a_2,..., a_n. Визначити, чи є у послідовності розміщені поряд два однакові числа, що дорівнюють 12.

```
program Standart_5_2;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of integer; i, n: byte; f: boolean;
begin  // Введення n та елементів масиву - див. приклад 4.7!
 i:=0; f:= false;
 repeat inc(i); if a[i]=12 then begin inc(i); f:=a[i]=12 end
 until f or (i>=n-1);
 if f then writeln('Yes!') else writeln('No!');  readln
end.
```

Приклад 4.13. При включенні нового елемента у задану позицію k (див. рис. 4.1) необхідно попередньо розсунути масив, тобто пересунути

підмасив a_k,..., a_n вправо на одну позицію. Переміщення підмасиву треба виконувати з кінця і перевіряти, чи достатньо пам'яті виділено під масив.

6	9	5	1	12	18	24		
6	9	5	1	1000	12	18	24	

Рис. 4.1. Включення елемента (1000) у п'яту позицію

```
program Standart_6;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of real; x: real;  i, k: byte;
begin  // Введення n та елементів масиву - див. приклад 4.7!
  write('Novyy element= ');   readln(x);
  write('Pozycia= ');  readln(k);
 if n< nmax
  then
    begin for i:=n downto k do a[i+1]:=a[i]; a[k]:=x; inc(n);
         { Виведення масиву a - див. приклад 4.7!}
    end
  else begin writeln('Malo misctcia u masyvi') end; readln
end.
```

Видаляючи k-ий елемент з масиву, необхідно зсунути підмасив a_{k+1},..., a_n вліво на одну позицію. Відповідний фрагмент програми виглядає так:

```
…
n:=n-1;
for i:=k to n do a[i]:=a[i+1];
```

Циклічне зсування масиву характерне тим, що розмір масиву не змінюється. Циклічне зсування масиву вправо на одну позицію зображено на рис. 4.2. При зсуванні вправо напрям перегляду масиву і команди присвоювання у циклі такі ж самі, як і при включенні елемента у масив, а при зсуванні вліво – такі ж самі, як і при видаленні елемента з масиву.

6	9	5	1	12	18	24		
24	6	9	5	1	12	18		

Рис. 4.2. Циклічне зсування масиву вправо на одну позицію

Фрагмент програми циклічного зсування масиву вправо на k позицій має вигляд:

```
…
for j:=1 to k do
 begin x:=a[n];  // x – проміжна величина
   for i:=n downto 2 do a[i]:=a[i-1];
   a[1]:=x;
 end;
```

Приклад 4.14. Дано натуральне число n і послідовність дійсних чисел a_1, a_2,..., a_n. Переставити члени послідовності так, щоб спочатку розмістилися всі її невід'ємні члени, а потім – всі від'ємні. Порядок розміщення як серед невід'ємних членів, так і серед від'ємних має зберегтися.

```
program Standart_7;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of real; x: real; i, j,n, k: byte;
begin write('Kilkist chysel= '); read(n);
 writeln('Chysla: '); for i:=1 to n do read(a[i]);
 i:=1; // початковий індекс невід'ємного члена
 for k:=1 to n do
  if a[i]<0
  then begin x:=a[i]; // збереження від'ємного члена
             // зсування масиву вліво на 1 позицію
             for j:=i to n-1 do a[j]:=a[j+1];
             a[n]:=x // від'ємний член у кінець масиву
       end
  else inc(i); // наступна позиція невід'ємного члена
 for i:=1 to n-1 do write(a[i]:6:2,', ');
 write(a[n]:6:2);  readln(i)
end.
```

Якщо a[i]<0, то ми розміщуємо його в кінці масиву, попередньо зсунувши підмасив від a[i+1] до a[n] на одну позицію вліво. Перший від'ємний член спочатку займе останнє місце, а далі при виявленні нового від'ємного члена буде зсуватися на одну позицію вліво.

Приклад 4.15. Дано натуральне число n і послідовність дійсних різних чисел $a_1, a_2,..., a_n$. Вибрати випадковим чином k $(1 \le k \le n)$ елементів цього масиву так, щоб жодне з них не повторилося.

```
program Standart_8;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of real; x: real; i, j,n, k: byte;
begin // Уведення n та елементів масиву a - див. приклад 4.7!
 write('Kilkist vybranykh chysel= '); read(k);
 for i:=1 to k do
  begin j:= i+Random(n-i+1); // випадковий індекс
        x:=a[j]; a[j]:= a[i]; a[i]:=x // обмін
  end;
 { Виведення масиву a - див. приклад 4.7!} readln(i)
end.
```

Вибираємо випадковий індекс j з проміжку $[1; n]$, і відповідний йому елемент a[j] міняємо місцями з першим елементом масиву. Наступного разу індекс j вибираємо з проміжку $[2; n]$, і елемент a[j] міняється місцями з другим елементом масиву і т.д. Вибрані елементи розміщуються на початку масиву і не беруть участі у наступних вибірках (вибірка без повторень).

Приклад 4.16. Дано натуральні числа n, m і два масиви дійсних чисел $a_1, a_2,..., a_n$ і $b_1, b_2,..., b_m$, які впорядковані за неспаданням. Утворити з елементів цих масивів впорядкований за неспаданням масив c.

```
program Standart_9;
{$APPTYPE CONSOLE}
const nmax = 30; mmax = 30;
var a: array [1..nmax] of real; b: array [1..mmax] of real;
   c: array [1..nmax+mmax] of real;  i, j, n, m, k: byte;
begin // Уведення n та елементів масиву a - див. приклад 4.7!
      // Уведення m та елементів масиву b - див. приклад 4.7!
 i:=1; j:=1;
 while (i<=n) and (j<=m) do
  if a[i]<=b[j] then begin c[i+j-1]:=a[i]; inc(i) end
                else begin c[i+j-1]:=b[j]; inc(j) end;
```

```
 while i<=n do begin c[m+i]:=a[i]; inc(i) end;
 while j<=m do begin c[n+j]:=a[j]; inc(j) end;
 { Виведення масиву c - див. приклад 4.7!} readln(i)
end.
```

4.3. Алгоритми пошуку в одновимірному масиві

Алгоритми пошуку застосовують для знаходження у масиві індексу елемента з потрібними властивостями. Розрізняють постановки задачі пошуку для першого і останнього входження елемента. Будемо вважати, що у всіх нижче викладених алгоритмах, проводиться пошук у масиві *a* з *n* цілих чисел індексу елемента, який дорівнює *x*.

Розрізняють також пошук у впорядкованому та невпорядкованому масивах. У невпорядкованому масиві пошук виконують за допомогою послідовного перегляду всього масиву і називають *лінійним пошуком*.

Лінійний пошук здійснюють у циклі **while** або **repeat** з подвійною умовою. Перша умова контролює індекс на приналежність масиву (`i<=n`). Друга умова – це умова пошуку: у циклі **while** – умова продовження пошуку (`a[i]<>x`), а у циклі **repeat** – умова завершення пошуку (`a[i]=x`). У тілі циклу є тільки один оператор, який змінює індекс масиву. Після виходу з циклу необхідно перевірити, за якою з умов ми вийшли. У операторі **if** звичайно повторюють першу умову циклу.

Приклад 4.17. Лінійний пошук у одновимірному масиві першого входження елемента, який дорівнює *x*.

```
program Poshuck_1;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of integer; x: integer; i, n: byte;
begin  // Уведення n та елементів масиву a - див. приклад 4.7!
 write('Vvesty x: '); read(x); i:=1;
 while (i<=n) and (a[i]<>x) do i:=i+1;
  if i<=n then write('1-she vhojennia',x,'v ', i,'misci')
        else write(' Ne znaydeno!');
 readln(i)
end.
```

Приклад 4.18. Лінійний пошук у одновимірному масиві останнього входження елемента, який дорівнює *x*. Наведемо основний фрагмент програми і використаємо цикл `repeat`.

```
…
 i:=n+1; repeat i:=i-1 until (i<1) or (a[i]=x);
 if i>=1 then write('Ostan. vhojennia',x,'v ', i,'misci')
        else write(' Ne znaydeno!');
```

Існує *пошук з бар'єром*. Ідея пошуку з бар'єром полягає в тому, щоб не перевіряти кожного разу у циклі умову, зв'язану з межами масиву. Це можна забезпечити, встановивши у масиві **бар'єр**: будь-який елемент, який задовольняє умові пошуку. Тим самим буде обмежена зміна індексу. Існує два способи установки бар'єра: додатковим елементом або замість крайнього елемента масиву.

Вихід з циклу, в якому тепер залишається тільки умова пошуку, може відбутися або на знайденому елементі, або на бар'єрі. Отже, після виходу з циклу слід перевірити, чи не бар'єр ми знайшли? Обчислювальна складність пошуку з бар'єром є меншою, ніж у лінійного пошуку, але також є величиною того ж порядку, що і `n` – кількість елементів масиву.

Приклад 4.19. Лінійний пошук з бар'єром у одновимірному масиві першого входження елемента, який дорівнює *x*.

```
program Poshuck_3;
{$APPTYPE CONSOLE}
const nmax = 30;
var a: array [1..nmax] of integer; x: integer; i, n: byte;
begin write('Kilkist chysel= '); read(n);
 if n=nmax then n:=n-1;
 writeln('Chysla: '); for i:=1 to n do read(a[i]);
 write('Vvesty x: '); read(x);
 a[n+1]:=x; // установка бар'єра додатковим елементом
 i:=1; while a[i]<>x do i:=i+1;
  if i<=n then write('1-she vhojennia',x,'v ', i,'misci')
        else write(' Ne znaydeno!');
 readln(i)
end.
```

Наведемо також основний фрагмент програми, якщо бар'єр є на останньому місці масиву.

```
     …
begin write('Kilkist chysel= '); read(n);
 writeln('Chysla: '); for i:=1 to n do read(a[i]);
 write('Vvesty x: '); read(x);
 y:=a[n]; // збереження останнього елемента
 a[n]:=x; // установка бар'єра на останнє місце масиву
 i:=1; while a[i]<>x do i:=i+1;
 if (i<n) or (y=x)
        then write('1-she vhojennia',x,'v ', i,'misci')
        else write(' Ne znaydeno!');
 a[n]:=y; // відновлення останнього елемента масиву
 readln(i)
end.
```

Розглянемо пошук у *впорядкованому* масиві. У цьому випадку його можна значно прискорити, застосовуючи метод половинного ділення масиву (*двійковий* або *бінарний пошук*).

Нехай маємо масив *a* з n елементів, який впорядковано за неспаданням ($a_i \le a_{i+1}$). Ідея алгоритму полягає в тому, що масив кожний раз ділиться пополам і вибирається та частина, де може знаходитися потрібний елемент. Ділення продовжується доти, доки частина масиву для пошуку складатиметься з одного елемента. Після цього залишається тільки перевірити цей елемент, що залишився, на виконання умови пошуку. Подаємо фрагмент програми, що реалізує метод половинного ділення масиву:

```
    …
left:=1; right:=n; // ліва і права межі пошуку
while left<right do
 begin m:=(left + right) div 2; // середина проміжку
  if x>a[m] then left:=m+1 else right:=m;
 end;
 if a[left]=x
     then write('1-she vhojennia',x,'v ', left,'misci')
     else write(' Ne znaydeno!');
```

Цей фрагмент дає змогу встановити позицію першого входження *x* у масив *a*. Наступні входження *x* знаходять за допомогою лінійного пошуку в *a*, починаючи з позиції `left+1`.

У процесі роботи алгоритму двійкового пошуку розмір фрагмента, де цей пошук має продовжуватися, кожний раз зменшується приблизно у два рази. Це забезпечує обчислювальну складність алгоритму порядку $\log_2 n$, де *n* – кількість елементів масиву.

4.4. Алгоритми сортування в одновимірному масиві

Сортування масиву – це перерозміщення елементів масиву у заданому порядку (домовимося розглядати випадок впорядкування масиву за неспаданням: $a_1 \le a_2 \le ... \le a_n$). Під *внутрішнім* сортуванням розуміють сортування, яке виконується в оперативній пам'яті ЕОМ без використання допоміжних масивів. *Основна мета* сортування – *полегшити наступний пошук*.

Сортування за допомогою **вибору**. Ідея методу полягає в тому, що знаходять максимальний елемент масиву і міняють місцями з останнім елементом (з номером *n*). Потім максимум шукають серед елементів з першого до передостаннього і ставлять на *n*-1 місце і т. д. Необхідно знайти *n*-1 максимум. Можна шукати не максимум, а мінімум і ставити його на перше місце, друге і т. д. Обчислювальна складність сортування за допомогою вибору – величина порядку $n \cdot n$, яку звичайно записують як $O(n^2)$. Це пояснюється тим, що кількість порівнянь при пошуку першого максимуму дорівнює *n*-1; потім *n*-2, *n*-3, …, 1; разом: $n \cdot (n-1)/2$.

Наведемо фрагмент програми, що реалізує сортування за допомогою **вибору**:

```
for k:=n downto 2 do
 begin m:=1; { m - місце max }
  for i:=2 to k do if a[i]>a[m] then m:=i;
  x:=a[m]; a[m]:=a[k]; a[k]:=x;
 end;
```

Сортування за допомогою **обміну** (метод "*бульбашки*"). Ідея методу полягає в тому, що послідовно порівнюють пари сусідніх елементів масиву. Якщо вони розміщені не в тому порядку, то

здійснюємо перестановку, міняючи місцями пару сусідніх елементів. Після одного такого проходження на останньому місці (номер n) виявиться максимальний елемент (“спливає” перша “бульбашка”). Наступне проходження має розглядати елементи до передостаннього і т. д. Усього потрібно n-1 проходжень. Обчислювальна складність такого сортування $O(n^2)$.

Наведемо фрагмент програми, що реалізує сортування за допомогою **обміну**:

```
 for k:=n-1 downto 1 do //k - кількість пар порівняння
  for i:=1 to k do
   if a[i]>a[i+1] then // міняємо місцями сусідні елементи
    begin x:=a[i]; a[i]:=a[i+1]; a[i+1]:=x end;
```

Сортування за допомогою **включення**. Ідея методу полягає в тому, що кожного разу, маючи вже впорядкований масив з k елементів, ми додаємо ще один елемент, включаючи його в масив таким чином, щоб впорядкованість не порушилася. Сортування може проводитися одночасно з введенням масиву. На початку сортування впорядкована частина масиву містить тільки один елемент. Різні методи пошуку місця для елемента, що включається, приводять до різних модифікацій сортування включенням. При використанні лінійного пошуку обчислювальна складність сортування включенням складає $O(n^2)$, а при використанні двійкового пошуку – $O(n \cdot \log_2 n)$, де n– кількість елементів.

Подаємо фрагмент програми, що реалізує сортування за допомогою **включення** при *лінійному пошуку* місця включення:

```
// k - кількість елементів у впорядкованій частині масиву
for k:=1 to n-1 do
 begin x:=a[k+1]; i:=k;
       while (i>0) and (a[i]>x) do
            begin a[i+1]:=a[i]; i:=i-1; end;
       a[i+1]:=x;
 end;
```

А тепер подаємо фрагмент програми, що реалізує сортування за допомогою **включення** при *бінарному пошуку* місця включення:

```
for k:=1 to n-1 do
begin x:=a[k+1]; left:=1; right:=k;
 while left<right do //двійковий пошук останнього входження
   begin m:=(left+right+1) div 2;
    if x>=a[m] then left:=m else right:=m-1;
   end;
 if x>=a[left] then left:=left+1;
 {зсуваємо на 1 праворуч частину масиву, звільняючи
  місце для включення x}
 for i:=k downto left do a[i+1]:=a[i];
 a[left]:=x;
end;
```

4.5. Матричні задачі

У стандартній матричній задачі може бути потрібно:

- отримати нову матрицю за певним правилом;
- знайти деяку величину, використовуючи елементи цієї матриці;
- перетворити матрицю певним чином;
- визначити, чи володіє матриця певною властивістю;
- виконати над матрицею (матрицями) певну операцію.

У разі реалізації матричних задач використовують **вкладені** (складні) цикли, які містять у собі один або декілька інших циклів. Такі цикли використовувалися уже в алгоритмах сортування.

Приклад 4.20. Дано натуральне число n. Отримати дійсну матрицю $a[i,\ j]_{i,\ j=1,\ldots,n}$, для якої виконуються умови:

$$a[i,j]=\begin{cases}(i+j)^2, & \text{якщо } i<j;\\ 2, & \text{якщо } i=j;\\ \sqrt{2i+5j}, & \text{якщо } i>j.\end{cases}$$

Індекс i позначає рядок квадратної матриці, а j – стовпець. Рівні індекси ($i=j$) мають елементи головної діагоналі. Умова $i<j$ вказує на те, що елементи розміщені над головною діагоналлю, а $i>j$ – елементи знаходяться під головною діагоналлю.

```
program Matrix_1;
{$APPTYPE CONSOLE}
const n = 10;
var a: array [1..n, 1..n] of real; i, j: 1..n;
begin
 for i:=1 to n do a[i,i]:=2; // головна діагональ
 for i:=1 to n-1 do // над головною діагоналлю
   for j:=i+1 to n do a[i,j]:=sqr(i+j);
 for i:=2 to n do // під головною діагоналлю
   for j:=1 to i-1 do a[i,j]:=sqrt(2*i+5*j);
 for i:=1 to n  do // виведення матриці
   for j:=1 to n do
     begin Write(a[i,j]:6:2, ' ');Writeln end
 readln
end.
```

Приклад 4.21. Дано натуральне число n. Визначити, скільки додатних елементів містить матриця $a[i,\ j]_{i,\ j=1,\ldots,n}$, у якої $a[i,\ j]=\sin(i+j)$.

Шукане число можна знайти і без оголошення масиву, використовуючи тільки формулу, що задає вигляд елементів масиву:

```
program Matrix_2;
{$APPTYPE CONSOLE}
const n = 10;
var i, j: 1..n; k: byte;
begin k:=0;
 for i:=1 to n do
  for j:=1 to n do if sin(i+j)>0 then inc(k);
  Write('k=',k); readln
end.
```

Приклад 4.22. Дано натуральне число n. Визначити суму тих елементів матриці $a[i,\ j]_{i,\ j=1,\ldots,n}$, які розміщені на головній діагоналі або над нею, що перевищують усі елементи, які розміщені під головною діагоналлю.

Серед елементів матриці, які розміщені під головною діагоналлю, знайдемо найбільший елемент. Тоді елементи, які розміщені на

головній діагоналі або над нею і більші за знайдений найбільший елемент, будуть задовольняти умові задачі.

```
program Matrix_3;
{$APPTYPE CONSOLE}
const n = 10;
var a: array [1..n, 1..n] of real; i, j: 1..n; max, s: real;
begin
 for i:=1 to n do //ініціалізація матриці випадковими числами
  for j:=1 to n do a[i,j]:=random*10;
 max:=a[2, 1];
 for i:=3 to n do
  for j:=1 to i-1 do
   if a[i,j]>max then max:= a[i,j];
 s:=0;
 for i:=1 to n do
  for j:=i to n do
   if a[i,j]>max then s:=s+a[i,j];
  Write('s=', s); readln
end.
```

Приклад 4.23. Дано натуральне число n і деяка квадратна матриця $a[i, j]_{i,j=1,\ldots,n}$. Перетворити цю матрицю так, щоб перший рядок помінявся місцями з останнім, другий – з передостаннім і т.д.

```
program Matrix_4;
{$APPTYPE CONSOLE}
const n = 10;
var a: array [1..n, 1..n] of real; i, j: 1..n; x: real;
begin
 for i:=1 to n do //ініціалізація матриці
  for j:=1 to n do a[i,j]:=i+j;
 for i:=1 to n div 2 do
  for j:=1 to n do
   begin
        x:=a[i,j];a[i,j]:=a[n-i+1,j];a[n-i+1,j]=x
   end;
 for i:=1 to n  do // виведення матриці
```

```
   for j:=1 to n do
     begin Write(a[i,j]:6:2, ' ');Writeln end
 readln
end.
```

Приклад 4.24. Написати фрагменти програм, які реалізують основні операції над матрицями: а) додавання (віднімання) двох матриць; б) множення матриці на скаляр; в) множення двох матриць; г) транспонування квадратної матриці.

а) Матриці, які беруть участь в операціях додавання (віднімання) мусять мати однакові розміри, адже ці операції виконують поелементно:

```
for i:=1 to m do
  for j:=1 to n do c[i,j]:=a[i,j]+b[i,j];
```

Для знаходження різниці матриць достатньо змінити “+” на “-”.

б) Множення матриці на скаляр також виконують поелементно:

```
for i:=1 to m do
  for j:=1 to n do c[i,j]:=a[i,j]*p;
```

в) Матриці, які беруть участь в операції множення, мають такі розміри: $m\overset{n}{\rangle}A \times n\overset{k}{\rangle}B = m\overset{k}{\rangle}C$. Тоді будь-який елемент c_{ij} $(i = 1...m; j = 1...k)$ знаходять за формулою $c_{ij} = \sum_{l=1}^{n} a_{il} \cdot b_{lj}$:

```
for i:=1 to m do
  for j:=1 to k do
   begin s:=0;
         for l:=1 to n do s:=s+a[i,l]*b[l,j];
         c[i,j]:=s
    end;
```

г) Транспонування квадратної матриці полягає у заміні рядків матриці відповідними стовпцями і навпаки (тобто елементи d_{ij} і d_{ji} мають помінятися місцями). Під час транспонування головна діагональ матриці залишається без змін, а обмін здійснюємо **тільки** для елементів, які розміщені над цією діагоналлю (елементи під діагоналлю поміняються автоматично внаслідок симетрії):

```
for i:=1 to n-1 do
  for j:=i+1 to n do
   begin s:=d[i,j]; d[i,j]:=d[j,i]; d[j,i]:=s end;
```

4.6. Записи (комбіновані типи)

Запис (або комбінований тип) – це структура даних, яка може зберігати інформацію різних типів, об'єднаних під однією назвою. Компоненти запису називають *полями*. Опис типу запису має такий вигляд:

```
type
назва_типу = record
   список_назв_полів_1:  тип_1;
   ...
   список_ назв_полів N:  тип_N;
case поле_вибору: порядковий_тип of
   список_значень_1: (варіант_1);
   ...
   список_значень_К: (варіант_К);
end;
```

Список полів може складатися з двох розділів: постійної і варіантної частини (**case**). Варіантної частини запису може й не бути.У постійній частині йде перелік назв полів запису з вказівкою їхніх типів. Синтаксис такий самий, як і у розділі **var.** Типи полів запису можуть бути будь-якими. У свою чергу, тип запису може використовуватися для створення нових масивів і записів. Глибина вкладеності не обмежена. Приклад оголошення типу запису:

```
type men = record fio, adress: string; year: byte; end;
```

Після опису типу можна визначити відповідні змінні, наприклад:

```
var a, c:  men;
```

Можна оголошувати запис і безпосередньо під час опису змінних, не вводячи типу, наприклад:

```
var a, c: record fio, adress: string; year: byte; end;
```

Для звернення до полів запису застосовують **складну назву**, яка містить назву змінної-запису, крапку та назву поля, наприклад:

```
a.fio:= 'Жук П.І.'; a.year:= 1981;a.adress:= 'Свободи, 40/10';
```

Для зручності звернення до полів записів може використовуватися оператор приєднання

```
with змінна do оператор;
```

який дає змогу не вказувати перед кожним полем назву змінної типу запису, наприклад:

```
with a do
 begin fio:= 'Жук П.І.'; year:= 1981;
       adress:= 'пр. Свободи, 40/10';
 end;
```

У `списку_значень` варіантної частини запису вказують одне або декілька значень (розділених крапкою з комою) заданого простого типу, а `варіант` – це розділений крапками з комами список можливих полів, які не задані у постійній частині запису. `Поля_вибору` може й не бути. Якщо воно є, то його сприймають як постійне поле запису. Якщо його немає, вказують тільки тип, який має бути порядковим, але він не впливає ні на кількість перерахованих нижче значень, ні на типи цих значень.

Всі варіанти **розміщують в одному і тому ж місці пам'яті**, якого виділяють стільки, скільки потрібно для максимального за розміром варіанта. Це приведе до того, що зміна одного варіантного поля впливатиме на всі інші: збільшаться можливості перетворення типів.

Приклад 4.25. Використання варіантів.

```
program Project2;
{$APPTYPE CONSOLE}
var R: Record rem:  string;
        Case byte of
        3:    (n:integer);
        5:    (x, y, z:char);
        ord('a'):  (i,j:byte);
       end;
begin
 R.rem:=' запис з варіантами'; R.n:=25000;
 write(R.i,R.x,R.j,R.y); {виведеться: 168и97a}
 {ord('и')=168, ord('a')=97, 168+97*256=25000}
 readln
end.
```

Значення вибору можуть повторюватися. Назви полів запису є постійними й, отже, можуть повторюватися, але тільки на різних рівнях, наприклад:

```
var Rec: Record x: real; I: byte;
          Sr: Record a: real; x, y: integer; end;
         end;
```

4.7. Множини

Множина – основне математичне поняття. У математиці розглядають скінченні і нескінченні множини. У Паскалі допускаються тільки скінченні множини, що складаються не більш ніж з 256 різних елементів. Всі елементи мають бути значеннями одного порядкового типу, який називають *базовим типом* цієї множини. Базовий тип можна задати діапазоном або переліком з урахуванням допустимої кількості елементів і того, що елементи мають лежати у діапазоні від 0 до 255. Зі стандартних типів як базові підходять типи `Byte` і `Char`.

Скінченні множини задаються переліком їхніх елементів. У Паскалі елементи множини беруть у квадратні дужки. Множина може, взагалі, не мати елементів, тобто бути *порожньою,* тоді її зображують конструкцією [ ] . Приклади:

Математичний запис:	Запис на Паскалі:
∅	[]
{2, 3, 5, 7, 17, 19}	[2, 3, 5, 7, 17, 19]
{0, 1, 2, 3, 4}	[0, 1, 2, 3, 4] або [0..4]

У квадратних дужках можуть знаходитися допустимі вирази зі змінними, наприклад, [`i, j`] – множина, що містить поточні значення змінних `i, j` типу `Byte`. Як і в математиці, вважають, що порядок перерахування елементів ролі не грає і, що кожний елемент враховується тільки один раз, тому [`true, false`] еквівалентно [`false, true`], а [1, 4, 2] еквівалентно [1, 2, 4]. Записи типу [5..5], [9..0] відповідають одноелементній множині [5] і порожній множині.

Після завдання базового типу сукупність значень відповідного множинного типу визначається автоматично. У неї входять всі можливі множини, що є довільними комбінаціями значень базового типу. Як відомо, всяка *n*-елементна множина має 2^n підмножин, де *підмножина* – це будь-яка частина множини. Для опису множинного типу використовують конструкцію вигляду:

```
Set of базовий_тип
```

Наприклад, множинний тип

```
Set of 1..3
```

використовує як базовий тип діапазон цілих чисел від 1 до 3 і його значеннями можуть бути: [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3] і []. Запис

```
Set of Boolean
```

визначає значення: [`true`], [`false`], [`true`, `false`] і [].

Розглянемо ще один приклад:

```
Type CharSet = set of Char, // Тип множини символів
var MyAlphaSet: CharSet;  // Змінна типу CharSet
…
MyAlphaSet:=['a', 'v'];  // Оператор присвоєння
```

Набір операцій над множинами у Паскалі відображає відповідний набір операцій математики:

Математичний запис	Запис у Паскалі	Коментар
A=B	`A = B`	Множини A і B збігаються
A≠B	`A <>`	Множини A і B не збігаються
A⊇B	`A >= B`	Всі елементи множини B належать множині A
A⊆B	`A <= B`	Всі елементи множини A належать множині B
x∈ A	`x` **`in`** `A`	Значення `x` належить базовому типу множини A і входить у A
A∪B	`A + B`	Об’єднання множин A і B

A∩B	`A * B`	Перетин множин A і B
A\B	`A - B`	Різниця множин A і B

Перші п’ять операцій у таблиці – операції відношення. Вони двомісні і повертають логічне значення `true` або `false`. Останні три операції створюють похідні множини. Операнди мають бути порівнюваними, тобто побудованими на одному базовому типі.

Є дві спеціальні процедури для роботи з одиночними елементами множини: `include(A, x)` і `exclude(A, x)`, які відповідно включають або виключають елемент `x` з множини `A`.

Множини зберігаються у пам’яті дуже компактно: кожне значення базового типу представляється лише одним бітом. У пам’ять комп’ютера заноситься 1 або 0 залежно від того, містить множина цей елемент чи ні. Отже, після опису змінних

```
Vaf N, M: Set of 0..7;
```

і вказаної нижче серії присвоювань внутрішнє представлення множини M буде змінюватися так, як показано праворуч:

M := [];	00000000
M := [0, 1, 5, 7];	11000101
M := [3..6];	00011110

Такий спосіб збереження множин у пам’яті дає змогу звести операції об’єднання, перетину і різниці до порозрядних логічних операцій над відповідними послідовностями бітів. Наприклад, перетин множин виконується шляхом порозрядного логічного множення бітів:

N := [1, 2, 3, 5, 6];	01110110
M := [0, 1, 5, 7];	11000101
N * M = [1, 5]	01000100

Приклад 4.26. Підрахувати кількість цифр та знаків `'+'` і `'-'` у рядку символів.

```
program Project1;
{$APPTYPE CONSOLE}
var s:string; i,k:byte;
```

```
 begin
  write('Enter s'); readln(s);
  k:=0; for i:=1 to length(s) do
         if s[i] in ['0'..'9','+','-'] then inc(k);
  write('k=', k); readln
 end.
```

Приклад 4.27. Підрахувати кількість різних цифр у десятковому записі натурального числа n.

```
program Project2;
{$APPTYPE CONSOLE}
var n:longword; k:byte; d:0..9; sd: set of 0..9;
begin
 write('Enter n'); readln(n); sd:=[];
 repeat d:=n mod 10; sd:= sd+[d]; n:=n div 10 until n=0;
 k:=0; for d:=0 to 9 do if d in sd then inc(k);
 write('k=', k); readln
end.
```

Приклад 4.28. Дано текст з малих латинських букв, який закінчується крапкою. Надрукувати перші входження букв у текст, зберігаючи їхній взаємний порядок.

```
program Project3;
{$APPTYPE CONSOLE}
var c: char; st: set of 'a'..'z'; s:sting;
begin st:=[]; s:=’’;
 write('Enter c'); readln(c);
 while c<>'.' do
  begin
   if not(c in st) then begin st:=st+[c]; s:=s+c; end
   write('Enter c'); readln(c)
  end;
 write('s=', s); readln
end.
```

Приклад 4.29. Дано текст, який закінчується крапкою. Надрукувати перші входження малих латинських букв у текст, зберігаючи їхній взаємний порядок.

```
program Project4;
{$APPTYPE CONSOLE}
var c: char; VsiLat, st: set of 'a'..'z';
    s:string;
begin VsiLat:=['a'..'z']; st:=[]; s:='';
 write('Enter c'); readln(c);
 while c<>'.' do
  begin
   if (c in VsiLat) and (not(c in st))
    then  begin st:=st+[c]; s:=s+c; end;
   write('Enter c'); readln(c)
  end;
 write('s=', s); readln
end.
```

4.8. Типізовані константи

У Паскалі допускається використання типізованих констант. В розділі опису констант їх записують так:

```
ідентифікатор:тип = значення
```

Типізованим константам можна присвоювати інші значення у ході виконання програми, тому фактично вони являють собою змінні з початковим значенням.

Типізована константа у процедурі чи функції ініціалізується лише один раз: до початку роботи програми. При повторному вході у процедуру чи функцію ініціалізація типізованої константи не відбувається і вона зберігає те значення, яке мала до моменту попереднього виходу із блоку.

Типізовані константи можуть бути будь-якого типу, крім файлів, об’єктів і класів. Не можна також описувати типізовану константу-запис, якщо хоча б одне з її полів є полем файлового типу, об’єктом чи класом.

Оскільки типізована константа фактично не відрізняється від змінної, її не можна використовувати як значення при описі інших констант або границь типу діапазону.

Крім звичайних константних виразів, для задання значення типізованої константи використовують також константні адресні

вирази. Константний адресний вираз – це вираз, значенням якого є адреса глобальної змінної, типізованої константи, процедури чи функції. Константний адресний вираз не може звертатись до локальних змінних процедур чи до динамічних змінних, оскільки їхні адреси неможливо обчислити під час компіляції.

Приклади оголошення констант простих типів і типу **String**:

```
{-------- Правильні оголошення -------}
const C: =(white, red, black) = red;
 Name: String = 'зміна'; Yr: Word = 1989;
 X: Real = 0.1; Min: Byte= 0; D: 1..31 = 1; A: Char = 'Y';
{-------- Неправильні оголошення ------}
Mas:array [Min..23] of Real; // межа – типізована константа
 a, b, c : Byte = 0; // не допускається список
 var NameF: String = 'prog'; // розділ змінних
```

Описуючи типізовані константи – *масиви*, початкові значення відокремлюють одне від одного комами, а список обмежують круглими дужками, наприклад:

```
 const
  C: array[1..3] of String[5]=('white', 'red', 'black');
  Vect: array[1..5] of Byte = (0, 2, 4, 6, 8);
```

Описуючи типізовані константи – *масиви символів*, можна використовувати той факт, що всі символьні масиви і рядки зберігаються в запакованому форматі, тому для масиву-константи типу *Char* допускається задання рядка символів відповідної довжини. Два наступних описи є ідентичними:

```
  const
   Digit: array[0..9] of Char =
  ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
   DigChr: array[0..9] of Char = '0123456789';
```

Описуючи типізовані константи – *багатовимірні масиви*, множину констант, що відповідає кожному виміру, обмежують додатковими круглими дужками і відділяють від сусідньої множини комами. У результаті утворюються вкладені структури масивів, причому глибина вкладення має відповідати розмірності масиву. Множини констант, що знаходяться усередині, зв’язуються зі зміною крайнього правого індексу масиву (див. параграф 4.1).

Опис типізованих констант типу “*запис*” має вигляд:

```
ідентифікатор:тип =(список_значень_полів)
```

де `ідентифікатор` задає назву константи; `тип` – тип запису; `список_значень_полів` – це послідовность вигляду:

```
назва_поля: константа
```

Приклад:

```
Type Point=record x,y: real end;
 Vect=array[0..1] of Point;
 Month=(Jn, Fb, Mr, Ap, May, Jn,Jly, Ag, Sp, Oct, Nov, Dec);
 Date=record D: 1..31; M: month; Y: 1900..1999 end;
Const Or: Point=(x: 0; y: 1997);
 Line: Vect=((x: -3.1; y: 1.5),(x: 5.9; y:3.0));
 SomeDay: Date=(D: 16; M: May: Y: 1997);
```

У варіантних константах-записах допускається вказування тільки того варіанта полів, який відповідає встановленій попередньо константі поля-ознаки, наприклад:

```
type
 Forma = record
         case Boolean of
          True: (BPlace: Sting [40]);
          False:(C: String [20]; E: String [20];
                 EDate: array [1..3] of Word;
                 Count: Word)
         end;
const
 Percon1: Forma = (C: 'Китай'; E: 'Україна';
                   EDate: (16, 3, 89); Count: 12);
 Percon2: Forma = (BPlace: 'Москва');
```

У разі опису типізованих констант типу “*множина*” кожний елемент може являти собою або окрему константу відповідного типу, або діапазон, наприклад:

```
type
  Digits= set of 0..9;
  CharDig= set of '0'..'9';
 const
  DigSet1: Digits = [0, 2, 4, 6, 8];
```

```
DigSet2: Digits = [1..3, 5..7];
CharDigSet1:CharDig = ['0', '2', ' 4', '6', '8'];
CharDigSet2: CharDig = ['1'..'3','5'..'7'];
CharSet: set of Char = ['a'..'z','A'..'Z'];
```

Єдиним значенням типізованої константи – *вказівника* може бути тільки `NIL`, наприклад:

```
const pR: ^Real = NIL;
```

Зауваження. У Object Pascal початкові значення можна задавати і **глобальним** змінним у розділі опису змінних!

4.9. Тип Variant у Object Pascal

Тип `Variant` використовують у тих випадках, коли на етапі компіляції не можна заздалегідь визначити, якого типу дані будуть використовуватися у виразі або у параметрі підпрограми. Тип `Variant` дає змогу компілятору створити код, який буде здійснювати необхідне перетворення типів на етапі виконання програми. У змінну типу `Variant` варіант можна помістити ціле або дійсне значення, логічне значення, рядок, час і/або дату, OLE-об'єкт, масив довільної розмірності і довжини, який містить елементи одного з перелічених типів. На прикладі покажемо використання варіантів:

```
Var V1, V2, V3, V4, V5:Variant;
  I:Integer; D:Double; S:String;
Begin V1:=1; V2:=1234.5678; V3:='Рядок символів';
 V4:='1000'; V5:= V1+V2+V4; I:=V1; // I=1
 D:=V2; {D=1234.5678} S:=V3; // S='Рядок символів'
 I:=V4; {I=1000} S:=V5;  // S='2235.5678'
…
```

Будь-яка змінна типу `Variant` займає у пам'яті 16 байтів, де у перших двох байтах зберігається інформація про тип поточного значення змінної (поле `Vtype`). У полі `Vtype` у момент створення варіантної змінної компілятор записує ознаку відсутності даних (`varEmpty`). Під час роботи програми значення цього поля змінюються відповідно до поточного типу даних, розміщених у варіантній частині. Отримати поточний тип варіантних даних можна за допомогою спеціальної функції `VarType(x)`, яка повертає код типу варіантної змінної `x`. Якщо отриманий код порівняти за допомо-

гою логічної операції **and** зі стандартною константою `varTypeMask`, як-от:

```
VarType(x) and varTypeMask
```

– результатом буде одне зі значень: `VarEmpty` (немає даних); `VarNull` (невідомий тип); `VarSmallInt` (цілий тип `SmallInt`); `VarInteger` (цілий тип `Integer`); `VarSingle` (тип `Single`); `VarDouble` (тип `Double`); …; `VarString` (тип `String`) і т.д.

• ? Запитання для самоперевірки

1. Що таке масив?
2. Як описують тип масиву?
3. Як описують одновимірний масив?
4. Як описують багатовимірний масив?
5. Що таке елемент масиву?
6. Що таке лінійний пошук?
7. Що таке пошук з бар'єром?
8. Що таке бінарний пошук?
9. Охарактеризуйте спосіб сортування вектора вибором.
10. Охарактеризуйте спосіб сортування вектора обміном.
11. Охарактеризуйте спосіб сортування вектора включенням.
12. Напишіть фрагмент програми множення матриць.
13. Напишіть фрагмент програми транспонування матриці.
14. Що таке запис?
15. Як описують запис?
16. Для чого використовують оператор приєднання?
17. Що таке множина?
18. Як описують множину?
19. Перерахуйте й охарактеризуйте операції над множинами.
20. Що таке типізована константа?
21. Як описують типізовані константи?

• Завдання для програмування

Завдання 4.1. Дано натуральне число `n` і дійсні числа $a_1, a_2,..., a_n$. Скласти програми розв'язання таких задач (у задачах 1 – 7 передбачити опрацювання ситуації відсутності від'ємних елементів):

1. Знайти добуток елементів масиву, які розміщені після останнього від'ємного числа.

2. Знайти середнє арифметичне елементів масиву, які розміщені після останнього від'ємного числа.
3. Замінити всі елементи, які розміщені перед першим від'ємним числом, на число 2.
4. Замінити всі елементи, які розміщені після останнього від'ємного числа, на число 10.
5. Знайти середнє арифметичне елементів масиву, які розміщені між першим та останнім від'ємним числом.
6. Замінити всі від'ємні числа на їхні квадрати і впорядкувати після цього масив за незростанням.
7. Замінити всі від'ємні числа на їхні модулі і впорядкувати після цього масив за незростанням.
8. Знайти добуток елементів масиву, які розміщені після останнього входження найменшого числа.
9. Знайти суму елементів масиву, які розміщені перед останнім входженням найменшого числа.
10. Знайти середнє арифметичне елементів масиву, які розміщені після останнього входження найбільшого числа.
11. Знайти добуток елементів масиву, які розміщені перед останнім входженням найбільшого числа.
12. Знайти середнє арифметичне елементів масиву, які розміщені між останніми входженнями найбільшого та найменшого числа.
13. Знайти суму елементів масиву, які розміщені між першими входженнями найбільшого та найменшого числа.
14. Замінити всі додатні числа на їхні квадрати і впорядкувати після цього масив за незростанням.
15. Замінити всі нульові числа на число 3 і впорядкувати після цього масив за незростанням.

Завдання 4.2. Дано натуральне число `n` і цілі числа a_1, a_2,..., a_n. Скласти програми розв'язання таких задач:

1. Впорядкувати масив за незростанням методом обміну.
2. Впорядкувати масив за незростанням методом вибору.
3. Впорядкувати масив за незростанням методом включення.
4. Стиснути масив, видаливши з нього числа, модуль яких менший за 3.
5. Стиснути масив, видаливши з нього всі нулі.
6. Стиснути масив, видаливши з нього всі від'ємні числа.
7. Після кожного нуля вставити у масив число 100.

8. Після кожного від'ємного числа вставити у масив два нулі.
9. Кожне від'ємне число у масиві оточити з обох боків нулями.
10. Впорядкувати масив за неспаданням модулів чисел методом обміну.
11. Впорядкувати масив за неспаданням модулів чисел методом вибору.
12. Впорядкувати масив за неспаданням модулів чисел методом включення.
13. Впорядкувати масив за незростанням модулів чисел методом обміну.
14. Впорядкувати масив за незростанням модулів чисел методом вибору.
15. Впорядкувати масив за незростанням модулів чисел методом включення.

Завдання 4.3. Дано цілочисельну прямокутну матрицю. Скласти програми розв'язання таких задач:

1. Визначити кількість рядків, які не містять нулів.
2. Визначити кількість стовпців, які не містять нулів.
3. Визначити кількість стовпців, які містять хоча б один нуль.
4. Визначити кількість рядків, які містять хоча б один нуль.
5. Визначити добуток елементів у кожному рядку, що не містить нуля.
6. Визначити добуток елементів у кожному стовпці, що не містить нуля.
7. Визначити суму елементів у кожному рядку, що не містить від'ємних чисел.
8. Визначити суму елементів у кожному стовпці, що не містить від'ємних чисел.
9. Визначити середнє арифметичне елементів у кожному рядку, який містить хоча б один нуль.
10. Визначити середнє арифметичне елементів у кожному стовпці, який містить хоча б один нуль.
11. Визначити суму елементів у кожному рядку, який містить хоча б одне від'ємне число.
12. Визначити суму елементів у кожному стовпці, який містить хоча б одне від'ємне число.
13. Знайти номер першого з рядків, що не містить від'ємних чисел.
14. Знайти номер останнього з рядків, що не містить від'ємних чисел.
15. Знайти номер останнього з стовпців, що не містить нуля.

5. ПРОЦЕДУРИ І ФУНКЦІЇ

План викладу матеріалу

1. Загальні положення
2. Опис процедури. Оператор процедури
3. Опис функції. Виклик функції
4. Механізм взаємозв'язку між параметрами й аргументами
5. Відкриті параметри-масиви
6. Параметри без типів. Процедурні типи
7. Рекурсія
8. Параметри за домовленістю
9. Перевантаження підпрограм

Ключові терміни розділу

- *Допоміжні алгоритми*
- *Оператор процедури*
- *Виклик функції*
- *Параметри-змінні*
- *Параметри-константи*
- *Процедура* `exit`
- *Параметри без типів*
- *Поняття рекурсії*
- *Перевантаження підпрограм*
- *Процедура. Опис процедури*
- *Опис функції*
- *Параметри-значення*
- *Локальні та глобальні змінні*
- *Вихідні параметри*
- *Відкриті параметри-масиви*
- *Процедурні типи*
- *Параметри за домовленістю*
- *Ключове слово* **`overload`**

5.1. Загальні положення

Паскаль дає можливість виділяти фрагменти програми у допоміжні алгоритми, за допомогою яких можна писати добре структуровані програми. Мови програмування, в яких передбачені допоміжні алгоритми, називають *процедурно-орієнтованими*. Структуровані програми зазвичай простіші для розуміння та налагодження.

Наявність допоміжних алгоритмів у мові програмування – це можливість застосовувати спеціальні методи проектування і розроблення складних програмних комплексів. Перший *називають методом низхідного програмування*, або розробкою програми "зверху – вниз", при якому спочатку створюється головна програма, у якій передбачається наявність деяких допоміжних алгоритмів, що вирішують певні задачі. Потім переходять до детального розроблення згаданих попередньо допоміжних алгоритмів.

Іншим підходом проектування і розроблення програм є *метод висхідного програмування*, або проектування “знизу – вгору”. У цьому випадку все починається зі створення невеликих допоміжних алгоритмів, з яких потім створюють більш складні допоміжні алгоритми і, нарешті, основну програму.

У Паскалі допоміжні алгоритми оформляють у вигляді *процедур* або *функцій*. Процедури і функції за своєю структурою подібні до звичайних програм і мають загальну назву – *підпрограми*.

У програмі опис підпрограм зазвичай розміщують між розділами змінних і операторів, хоча Object Pascal допускає довільну послідовність опису констант, змінних, типів, позначок і підпрограм. Але необхідно пам’ятати **основне правило** Object Pascal: будь-яка назва у програмі має бути описана перед тим, як вона з’явиться серед операторів, що виконуються. Кожна процедура або функція визначається лише один раз, але може використовуватися багатократно.

У процедурах і функціях, як і у звичайних програмах, можуть бути описані власні мітки, константи, типи, власні змінні і навіть підпрограми. Кожна підпрограма має власну назву, яка використовується в разі виклику підпрограми на виконання (процедура викликається окремим оператором, а виклик функції є частиною виразу).

5.2. Опис процедури. Оператор процедури

Опис процедури має таку структуру:

```
Procedure назва(список_формальних_параметрів);
label
const                    // Опис локальних позначок,
type                     // констант, типів і змінних
var
procedure                // Опис внутрішніх процедур
function                 // і функцій
begin
      // Оператори тіла процедури
end;
```

Опис кожної процедури починається зі **заголовка**, в якому задають назву процедури і список формальних параметрів. Проце-

дура може бути і без параметрів, тоді у заголовку вказують лише її назву. За допомогою параметрів передають вихідні дані у процедуру, а також результати роботи процедури назад у головну програму.

Список формальних параметрів може складатися з *параметрів-значень*, *параметрів-змінних* (перед ними стоїть зарезервоване слово **var**), й деяких інших категорій формальних параметрів. Заголовок процедури з параметрами-значеннями, наприклад, має вигляд:

```
Procedure MyProc1(pr1, pr2: type1; pr3: type2);
```

Заголовок процедури з параметрами-змінними, наприклад, має вигляд:

```
Procedure MyProc2(var p1, p2: type1; var p3, p4: type2);
```

У заголовку процедури можуть бути і параметри-змінні й параметри-значення, наприклад:

```
Procedure MyProc3(var p1, p2: type1; p3, p4, p5: type2);
```

Виклик процедури на виконання здійснюється за допомогою оператора процедури:

```
Назва_процедури(список_фактичних_параметрів);
```

Між формальними і фактичними параметрами має бути відповідність, тобто однакова кількість формальних і фактичних параметрів, порядок фактичних і формальних параметрів має бути одним і тим самим; тип кожного фактичного параметра має збігатися з типом відповідного йому формального параметра. Фактичні параметри іноді ще називають *аргументами*, а формальні параметри – просто *параметрами*.

5.3. Опис функції. Виклик функції

Опис функції має таку структуру:

```
Function назва(список_формальних_параметрів):тип_результату;
label
const                    // Опис локальних позначок,
type                     // констант, типів і змінних
var
procedure                // Опис внутрішніх процедур
function                 // і функцій
```

```
begin
    // Оператори тіла функції, серед яких є хоча б один,
    // що присвоює назві функції значення результату
end;
```

Опис кожної функції починається зі **заголовка**, в якому задано назву функції, список формальних параметрів і тип результату. Типом результату функції може бути будь-який стандартний тип Паскаля, крім файлових типів.

Список формальних параметрів може включати в себе *параметри-значення*, *параметри-змінні* (перед ними стоїть слово **var**), й деякі інші категорії формальних параметрів. Заголовок функції з параметрами-значеннями, наприклад, має вигляд:

```
Function MyFunc1(pr1, pr2: type1; pr3: type2): type;
```

Заголовок функції з параметрами-змінними, наприклад, має вигляд:

```
Function MyFunc2(var p1, p2: type1; var p3: type2): type;
```

У заголовку функції можуть бути і параметри-змінні й параметри-значення, наприклад:

```
Function MyFunc3(var p1, p2: type1; p3, p5: type2): type;
```

Виклик функції на виконання знаходиться у деякому виразі і має вигляд: `назва_функції(список_фактичних_параметрів)`. Між формальними і фактичними параметрами має бути відповідність за кількістю і типами.

5.4. Механізм взаємозв'язку між параметрами і аргументами

При виклику підпрограми (процедури або функції) формальним *параметрам-значенням* виділяється нове місце у пам'яті і туди заносять значення фактичних параметрів. У таких випадках на місці фактичних параметрів можуть бути вирази. Сумісність типів параметрів і аргументів визначається можливостями присвоєння. Після виконання підпрограми місце розміщення у пам'яті формальних параметрів звільняється. Зміна формальних параметрів у підпрограмі не позначається на значенні фактичних параметрів.

Отже, *параметри-значення* містять копії даних, які передаються. Після завершення роботи підпрограми, ці копії знищуються. Як бачимо, параметри-значення можна використовувати лише для передачі у підпрограму вихідних даних.

При виклику підпрограми формальні *параметри-змінні* займають те ж саме місце у пам'яті, що і відповідні їм фактичні параметри. Таким чином, додаткове місце у пам'яті не виділяється і зміни формального параметра змінюють фактичний параметр. Параметри-змінні зазвичай використовуються для передачі результатів з підпрограми у програму чи іншу підпрограму.

Такий механізм передачі даних потребує, щоб фактичні параметри були **змінними**, причому того ж самого типу, що і формальні параметри. Описуючи підпрограму перед параметрами-змінними ставлять слово `var`.

Присвоєння значень параметру-змінній всередині підпрограми одночасно виконується і для відповідного аргументу (фактичного параметра). Це пов'язано з тим, що параметр-змінна містить не саме значення, що передається, а адресу, за якою воно розміщується в оперативній пам'яті комп'ютера (передача за посиланням). Маючи у своєму розпорядженні адресу, підпрограма може довільно змінювати дані, що знаходяться за цією адресою. Отже, параметри, в які записуються результати роботи процедури, мають передаватися лише за посиланням (тобто бути параметрами-змінними). Параметри-змінні зазвичай використовують і для передачі у підпрограму структурованих даних (масивів, записів тощо), оскільки під час цього не виділяється додаткова пам'ять.

Існують поняття **локальних** і **глобальних** позначок, констант, типів і змінних. Пояснимо їх на прикладі змінних. Змінні, описані в основній програмі, є *глобальними* щодо процедур і функцій, які описані пізніше від цих змінних. Аналогічно, змінні, описані в процедурах і функціях, є глобальними щодо внутрішніх процедур і функцій, які описані пізніше. Інші змінні називаються *локальними*. Їхня область дії є **локалізованою**, тобто вони обмежені тією підпрограмою, де описані. Зауважимо, що **глобальним** змінним у Object Pascal можна присвоювати початкові значення у розділі опису змінних (розділі `var`).

Початкові дані для роботи підпрограми можна передавати як через параметри, так і через глобальні змінні (автор не рекомендує використовувати цей спосіб через можливі помилки при виконанні програми, які важко діагностувати).

Формальні параметри підпрограми також відносять до локальних змінних. Локальні дані створюються компілятором під час виклику підпрограми, а звільнення цієї пам'яті відбувається після завершення роботи підпрограми. У тому випадку, коли локальна змінна має такий самий ідентифікатор, що і глобальна змінна, підпрограма працює з локальною змінною. Водночас значення глобальної змінної зберігається у стекові.

Вище вже були описані дві категорії формальних параметрів: параметри-значення і параметри-змінні. Крім них, у Паскалі є ще *параметри-константи*, а у Object Pascal ще й *вихідні параметри*. Розглянемо особливості застосування нових категорій формальних параметрів.

При використанні параметра-константи перед ним у заголовку процедури чи функції має стояти зарезервоване слово **const**, наприклад:

```
procedure sum(const x:real; y:integer; var z:real);
```

Параметр-константа передається подібно до параметра-змінної (за посиланням), але між ними є суттєва відмінність*: значення параметра-константи не можна змінювати* всередині підпрограми. Якщо ж спробувати змінити значення параметра-константи, то компілятор виведе повідомлення про помилку.

Отже, параметри-константи слід використовувати в тому випадку, коли ми хочемо бути впевненими, що фактичний параметр не буде змінений у разі виклику підпрограми. Використовувати параметри-константи рекомендується при передачі даних великого об'єму з гарантією збереження їхніх значень.

Описуючи вихідний параметр, перед ним, у заголовку процедури, слід ставити зарезервоване слово **out**, наприклад:

```
procedure abc(x,y:real; out z:real);
```

Вихідний параметр подібний до параметра-змінної. Але, якщо параметр-змінну можна використовувати як для передачі у процедуру вхідних даних, так і для повернення результату, то вихідний параметр слід використовувати *тільки для повернення результату*,

оскільки пам'ять, що зайнята відповідним фактичним параметром, у момент виклику процедури очищається.

Приклад 5.1. Обчислити $u = \dfrac{\max(x,\ x \cdot y) + \max^3(y, y + x)}{1 + \max^2(x + y^2, xy)}$. Для ліпшого розуміння різниці між процедурами і функціями – обчислення величини $\max(a, b)$ оформимо: а) у вигляді процедури; б) у вигляді функції.

а) Розв'язок:

```
program Project1;
{$APPTYPE CONSOLE}
uses SysUtils;
var x, y, t, u:real;
procedure max(a,b:real; var c:real);
  begin if a>b then c:=a else c:=b end;
begin  writeln('Enter x, y'); readln(x, y);
 max(x,x*y,u); max(y,y+x,t); u:=u+sqr(t)*t;
 max(x+y*y,x*y,t); u:=u/(1+t*t);
 writeln('u=', u:7:3); readln;
end.
```

б) Розв'язок:

```
…
var x, y, t, u:real;
function max(a,b:real):real;
  begin if a>b then max:=a else max:=b end;
begin  writeln('Enter x,y'); readln(x,y);
 u:=max(x, x*y); t:=max(y, y+x); u:=u+sqr(t)*t;
 t:=max(x+y*y,x*y); u:=u/(1+t*t);
 writeln('u=', u:7:3); readln;
end.
```

У Object Pascal для присвоєння результату може використовуватися не тільки назва функції, але й стандартна змінна `result`, наприклад:

```
function max(a, b:real):real;
 begin if a>b then result:=a else result:=b end;
```

Результат роботи функції в обох випадках буде однаковий. Але змінна `result` має одну принципову відмінність від назви функції. Назва функції – це не змінна! Вона може фігурувати тільки у лівій частині оператора присвоювання і не може входити у будь-які вирази. А `result` – це звичайна змінна, а тому вона може входити у будь-який вираз, наприклад:

```
function fab(a, b:real):real;
 begin result:=a+b; result:=sqr(result)*result end;
```

У Паскалі існує стандартна процедура `exit`, яка забезпечує вихід з підпрограми у будь-якому місці підпрограми, наприклад:

```
function fex(a, b:real):real;
  begin result:=a+b; if result<0 then exit;
        result:=sqr(result)*result     end;
```

Приклад 5.2. Запрограмувати і протестувати процедуру сортування одновимірного масиву методом вибору.

```
program ProcSort;
 {$APPTYPE CONSOLE}
const nmax = 30;
Type masyv=array [1..nmax] of real;
var a: masyv; i, n: byte;
procedure  SortMasyv(var mas:masyv; n: byte);
  var i, m, k: byte; x: real;
begin
 for k:=n downto 2 do
  begin m:=1; { m - місце max }
   for i:=2 to k do if mas[i]>mas[m] then m:=i;
   x:=mas[m]; mas[m]:=mas[k]; mas[k]:=x
  end
end;
begin write('Kilkist chysel= ');  read(n);
 writeln('Chysla: '); for i:=1 to n do read(a[i]);
 SortMasyv(a,n); for i:=1 to n do write(a[i]:6:2,' ');
 readln(i)
end.
```

Зауваження. Для більшої універсальності у процедуру `SortMasyv` передається параметр `n` – поточна верхня межа масиву `mas`. Ни-

жню межу зафіксовано в 1. Якщо нижня межа масиву-аргумента матиме інше значення, то процедуру `SortMasyv` треба переробляти (втрачається універсальність). Часом ситуацію вдається виправити за допомогою *відкритих* параметрів-масивів.

5.5. Відкриті параметри-масиви

Відкриті параметри-масиви можуть бути параметрами-значеннями, параметрами-змінними і параметрами-константами. Вони використовуються для передачі масивів довільної розмірності. Заголовок процедури з відкритими параметрами-масивами може мати, наприклад, такий вигляд:

```
Procedure OpenArray(Vector: array of Type);
```

Формальний параметр `Vector` тут є масивом елементів деякого типу `Type` з нульовою базою, тобто **`Array`** `[0..N]` **`of`** `Type`; де `N` – верхня межа масиву, яку можна визначити за допомогою стандартної функції `High`. Фактичний параметр-масив може мати довільні межі; перетворення до меж `0..N` у рамках роботи підпрограми здійснюється **автоматично**.

Приклад 5.3. Збільшення удвічі елементів масивів з різними межами.

```
program Project3;
{$APPTYPE CONSOLE}
const       n=5; m=2;
type  T1 = array[1..n] of integer;
T2 = array[-m..m] of integer;
var A: T1; y: T2; k: integer;
Procedure Double(var X: array of integer);
var i: byte;
begin for i:=0 to High(X) do X[i]:=X[i]*2; end;
begin  writeln('Enter 2*5 elementiv ');
 for k:=1 to n do read(A[k]);
 for k:=- m to m do read(y[k]);
 Double(a); {Збільшення у 2 рази елементів масиву A}
  Double(y); {Збільшення у 2 рази елементів масиву y}
  Double(k); {Збільшення у 2 рази k,тобто k:=k*2}
  writeln(' k=', k);
  for k:=1 to n do write(a[k],' '); writeln;
```

```
  for k:=- m to m do write(y[k],' '); readln(k)
 end.
```

5.6. Параметри без типів. Процедурні типи

У Паскалі є можливість створення процедур і функцій з параметрами, що не мають типу. Параметри без типів можуть бути параметрами-змінними і параметрами-константами, оскільки передаються тільки за адресою. Заголовок процедури з параметрами, що не мають типу, може виглядати, наприклад, таким чином:

```
Procedure MyProc(var par1, par2; const par3, par4);
```

Перед використанням формальних параметрів необхідно їх привести до якого-небудь типу. Використання параметрів без типів дає велику гнучкість програмі, але відповідальність за їхнє коректне застосування покладається на програміста.

Приклад 5.4. Додавання перших N байтів, з того ж місця, що і X.

```
program without_type;
var N:word; s:string;
{$R-} (*відключення контролю за межами діапазонів *)
function Sum(var X; N:byte): word;
type A=array[1..1] of byte;
var i:byte; s:word;
begin s:=0; for i:=1 to n do S:=S+A(X)[i]; Sum:=s end;
begin write('enter string: '); readln(s);
 writeln(Sum(s[1],1)); //код першого символа рядка s
 writeln(Sum(s[1], length(s)));
 {сума кодів усіх символів рядка s}
 read(N); writeln(Sum(N, 2)); {сума двох байтів,
   з яких складається N типу word} readln(n)
end.
```

У програмі можуть бути наявні змінні процедурних типів. Змінним процедурних типів як значення присвоюються назви відповідних підпрограм. Цей підхід широко використовується у Object Pascal, особливо при використанні класів.

У разі створення консольних програм процедурний тип використовують для передачі назв процедур і функцій як параметри з основної програми у підпрограму.

Є два процедурних типи: тип-процедура і тип-функція. Опис процедурного типу має вигляд:

```
type назва_процедурного_типу= заголовок_підпрограми_без_назви;
```

Наприклад:

```
 type
    func=function(x,:real):real;
    p1=procedure(x, y:integer; var z:real);
    …
 function test1(x,:real):real;
    begin test1:=x*x end;
  …
```

Застосування процедурного типу найліпше пояснити на прикладі функції, що обчислює інтеграл $\int_a^b f(x)dx$. Заголовок такої функції може виглядати так:

```
Function Integr(a, b:real; f:func):real;
```

а виклик функції `Integr` можна здійснити таким чином:

```
Square:= Integr(1, 5, test1);
```

5.7. Рекурсія

Рекурсія – це спосіб організації обчислювального процесу, при якому підпрограма у ході виконання звертається сама до себе. Рекурсія аналогічна методу математичної індукції: базі індукції відповідає база рекурсії; припущенню індукції відповідає припущення про те, що потрібна підпрограма вже написана; кроку індукції відповідає виклик рекурсивної підпрограми, що створюється. У будь-якій рекурсії необхідно передбачити умову завершення процесу, тобто коли виклику більше не відбувається.

Приклад 5.5. Обчислити `n`-е число Фіббоначчі (див. тему *цикли*):

```
program Fib;
{$APPTYPE CONSOLE}
var n:byte;
function F(k: byte): word;
 begin if k<2 then F:=1 else F:=F(k-1)+F(k-2) end;
begin write('Enter n '); readln(n);
```

```
    writeln(n,' -e chyslo Fibonachi=', F(n)); readln
  end.
```

Рекурсивний виклик може бути непрямим (або неявним). Це відбувається у випадку, коли одна підпрограма викликає іншу, а та у свою чергу – першу. При використанні такої програмної конструкції необхідний випереджальний опис заголовка підпрограми з директивою **Forward**. Реалізацію підпрограми наводимо нижче. Разом з тим у ній можна писати у заголовку або тільки назву підпрограми, або повністю повторювати заголовок, наприклад:

```
Procedure B(x: byte); forward;
Procedure A(y: byte); begin … B(y);  … end;
Procedure B; … begin … A(x); … end;
```

5.8. Параметри за домовленістю

У Object Pascal є параметри, значення яких можна не вказувати при виклику підпрограми (*параметри за домовленістю*), які описують так: `назва_параметра:тип=значення`

Параметри за домовленістю мусять знаходитись наприкінці списку формальних параметрів. Якщо при виклику підпрограми аргумент, що відповідає параметру за домовленістю, не вказаний, то у підпрограму передається визначене у заголовку підпрограми `значення`. Якщо у підпрограмі є декілька параметрів за домовленістю і для частини з них задані фактичні значення, а для частини – ні, то останні мають розміщуватись у кінці списку формальних параметрів. Тобто при виклику підпрограми фактичні параметри мають знаходитись підряд, без пропусків.

Приклад 5.6. Скласти програму для знаходження максимального значення для додатних чисел, кількість яких не перевищує чотири. У процедурі для пошуку максимального значення опишемо чотири нульові значення за домовленістю:

```
program Project1;
{$APPTYPE CONSOLE}
var x,y,z,w,m:real;
procedure p(var max:real; a:real=0; b:real=0;
            c:real=0; d:real=0);
begin  max:=a; if max<b then max:=b;
```

```
        if max<c then max:=c; if max<d then max:=d
end;
begin writeln('Enter x,y,z,w'); readln(x,y,z,w);
 p(m,x,y); writeln('max=', m:3:0);
 p(m,x,y,z);  writeln('max=', m:3:0);
 p(m,x,y,z,w);  writeln('max=', m:3:0); readln
end.
```

5.9. Перевантаження підпрограм

У мові Паскаль є можливість використовувати підпрограми з однаковими назвами, але з різною кількістю і типом параметрів. У заголовках таких підпрограм потрібно вказувати зарезервоване слово **overload**. Під час виклику таких підпрограм компілятор аналізує кількість і тип фактичних параметрів і викликає відповідну підпрограму.

Приклад 5.7. Скласти програму для знаходження мінімального з двох чисел і мінімального порядкового номера для двох символів. У програмі будемо використовувати дві функції з назвою min. Першу для знаходження мінімального числа, другу – для визначення мінімального порядкового номера для двох символів:

```
program Project1;
{$APPTYPE CONSOLE}
uses SysUtils;
var x,y:real;  s,v:char;
function min(a,b:real):real; overload;
 begin if a>b then Result:=b else Result:=a end;
function min(a,b:char):char;overload;
 begin Result:=a; if a>b then Result:=b end;
begin  writeln('Enter two numbers'); readln(x, y);
 writeln('min=', min(x, y):3:0);
 writeln('Enter two chars'); readln(s, v);
 writeln('min=', min(s, v)); readln;
end.
```

? Запитання для самоперевірки

1. Для чого використовують допоміжні алгоритми?
2. Як описують процедуру?

3. Як описують функцію?
4. Що таке оператор процедури?
5. Як реалізовується виклик функції?
6. Що таке параметри-значення?
7. Що таке параметри-змінні?
8. Що таке параметри-константи?
9. Для чого використовують процедуру `exit`?
10. Для чого використовують процедурні типи?
11. Для чого використовують відкриті параметри-масиви?

💻 Завдання для програмування

Завдання 5.1. Виконати *завдання 2.2* (стор.39), оформивши обчислення відповідної функції у вигляді *підпрограми-процедури*.

Завдання 5.2. Виконати *завдання 2.2* (стор.39), оформивши обчислення відповідної функції у вигляді *підпрограми-функції*.

Завдання 5.3. Виконати *завдання 2.4* (стор.40), оформивши обчислення функції за допомогою розкладу у ряд Тейлора у вигляді *підпрограми-функції*.

Завдання 5.4. Виконати *завдання 3.1* (стор.53), оформивши розв'язання основної функціональної задачі у вигляді *підпрограми-процедури*.

Завдання 5.5. Виконати *завдання 4.1* (стор.86), оформивши розв'язання основної функціональної задачі у вигляді *підпрограми-функції*.

Завдання 5.6. Виконати *завдання 4.2* (стор.87), оформивши розв'язання основної функціональної задачі у вигляді *підпрограми-процедури*.

Завдання 5.7. Виконати *завдання 4.3* (стор.88), оформивши розв'язання основної функціональної задачі у вигляді *підпрограми-функції*.

6. ПОКАЖЧИКИ. ДИНАМІЧНІ СТРУКТУРИ ДАНИХ

План викладу матеріалуч

1. Статичне і динамічне розміщення даних у пам'яті
2. Покажчики
3. Операція рознайменування
4. Динамічні змінні
5. Нетипізовані покажчики
6. Динамічні масиви
7. Динамічні структури даних
8. Реалізація лінійних списків

Ключові терміни розділу

- *Сегмент даних*
- *Статичне розміщення даних*
- *Динамічні змінні*
- *Поняття покажчика*
- *Опис типізованого покажчика*
- *Порожній покажчик* (`nil`)
- *Операція рознайменування*
- *Процедури* `New` *і* `Dispose`
- *Динамічні масиви*
- *Списки*
- *Поняття стека*
- *Динамічне розміщення даних*
- *Поняття купи*
- *Опис типу покажчика*
- *Операція @*
- *Присвоєння значень покажчикам*
- *Нетипізований покажчик*
- *Процедури* `GetMem` *і* `FreeMem`
- *Динамічні структури даних*
- *Лінійні списки, стеки і черги*

6.1. Статичне і динамічне розміщення даних у пам'яті

Як відомо, усі дані (прості та структуровані), які трапляються у програмі, мають бути описані. Перед початком виконання програми компілятор кожній змінній (як простій, так і структурованій), виділяє місце в оперативній пам'яті комп'ютера для розміщення її поточних значень.

Домовимося, що надалі у цьому розділі для спрощення викладу матерілу термін "*змінна*" у багатьох випадках буде використовуватися для позначення як власне змінних, так і структурованих даних (масивів, записів, множин і т.д.). Вдумливий читач за контекстом зможе легко розібратися, про що саме йде мова.

Отже, змінним ставиться у відповідність певна *адреса* ділянки пам'яті комп'ютера, у якій зберігатиметься її поточне значення (для структурованих даних – деяка множина поточних значень).

Звертання у програмі до значення, розміщеного за цією адресою, відбувається за допомогою відповідної змінної. Відповідність між змінною і адресою пам'яті зберігається протягом усього часу виконання програми для **глобальних** змінних або протягом усього часу виконання підпрограми для **локальних** змінних.

Розмір необхідної ділянки пам'яті для розміщення **глобальних** змінних компілятор може розрахувати під час етапу компілювання програми і на цьому ж етапі пам'ять виділити. Глобальні змінні розміщуються компілятором в одній неперервній області оперативної пам'яті, яка називається *сегментом даних* (або *областю глобальних даних*). У процесі виконання програми місце розміщення та розмір глобальних змінних не змінюються. Розмір пам'яті глобальних даних конкретного проекту можна побачити, вибравши команду ➢*Project* ➢ *Information* після компіляції програми – це вже значення, показане у вікні як розмір даних (*data size*).

Локальні змінні розміщуються компілятором у сегменті **стека**. Стек (*stack*) – це динамічна область пам'яті, яка виділяється і звільняється у порядку `LIFO` (*last in, first out* – останнім прийшов, першим вийшов). Це означає, що останій, розміщений у пам'яті, об'єкт буде першим у черзі на знищення. Стек використовується підпрограмами для передавання аргументів і результуючих значень, а також для локальних змінних підпрограми. Після завершення виклику підпрограми її область пам'яті у стекові звільняється. Зауважимо, що за домовленістю, прийнятою у *Delphi*, аргументи зазвичай передаються у регістрах процесора.

Windows-програма може зарезервувати під стек достатньо велику кількість пам'яті. У Delphi це значення настроюється на сторінці *Linker* (компонувальник) діалогового вікна *Project Options*. Однак значення за домовленістю цілком достатньо. Якщо програміст отримує повідомлення про переповнення стека, то це переважно відбувається через нескінченні рекурсивні виклики, а не через те, що розмір стека занадто малий. Початковий розмір стека можна побачити у разі вибору команди ➢*Project* ➢*Information.*

Отже, усі глобальні змінні розміщуються в оперативній пам'яті до початку виконання програми (**статично**) у процесі компілювання програми. Розміщення локальних змінних також є статич-

ним, оскільки компілятор до початку виконання програми резервує під стек ділянку оперативної пам'яті (зазвичай 64 Кб).

Крім цього, частину оперативної пам'яті займає код самої програми та інших резидентних програм. Вся інша **вільна** оперативна пам'ять комп'ютера, що надається програмі при її роботі, називається **динамічною пам'яттю**.

Через те, що розмір сегмента даних обмежений, часто виникають проблеми під час опрацювання великих масивів даних. Бувають також випадки, коли розміри пам'яті для розміщення деяких даних у ході виконання програми можуть змінюватися. Такого роду проблеми вирішують, використовуючи змінні, які розміщують у динамічній пам'яті (називатимемо такі змінні **динамічними**). Появою і часом існуваня динамічних змінних керує програміст.

Нагадаємо, що термін "змінна" використовується нами для позначення як власне змінних, так і структурованих даних (масивів, записів тощо).

Доступ до статичних змінних реалізують за допомого відповідних ідентифікаторів. Але такий підхід неможливий для динамічних змінних, оскільки заздалегідь є невідомою їхня кількість та місце у пам'яті. Доступ до динамічних змінних реалізується за допомогою особливих змінних – **покажчиків** (або **вказівників**), значеннями яких є адреси розміщення у пам'яті змінних (динамічних і статичних). Сам покажчик є *статичною* змінною.

Динамічне розміщення даних означає використання динамічної пам'яті безпосередньо під час виконання програми. При динамічному розміщенні попередньо не є відомий ні тип, ні кількість даних. Динамічний розподіл пам'яті широко використовується для економії обчислювальних ресурсів: ті змінні чи об'єкти, які стають непотрібними, програма знищує, а звільнене місце використовує для нових змінних чи об'єктів.

Всю динамічну пам'ять у Паскалі розглядають як суцільний масив байтів, який називають *купою*. Купа (*heap*) – це область, в якій розміщення і звільнення пам'яті відбувається випадково. Фізично купа розміщується у старших адресах відразу за областю пам'яті, яку займає тіло програми. Адреса початку купи зберігається в

стандартній змінній `heaporg`, кінець – у змінній `heapend`, поточна сторінка незайнятої динамічної пам'яті – `heapptr`.

6.2. Покажчики

Паскаль надає у розпорядження програміста гнучкий засіб керування динамічною пам'яттю – *покажчики.* **Покажчик** – це змінна, яка містить адресу байта пам'яті. За допомогою покажчиків можна розміщувати у динамічній пам'яті змінні будь-яких типів даних. Лише деякі з них (`Byte, Char, ShortInt, Boolean`) займають у внутрішньому поданні один байт, решта – декілька суміжних. Ось чому насправді покажчик адресує лише перший байт даних. Для визначення **типу** покажчика у програму включають такий опис:

```
type p= ^t;
```

Опис покажчиків має вигляд:

```
var v,w: p;
```

Змінні `v` і `w` є об'єктами типу `p` (**покажчики** на тип `t`), які містять адресу першого байта пам'яті, виділеного для об'єктів типу `t`. Покажчики, описані подібним чином, називають *типізованими покажчиками*. Типізовані покажчики оголошують символом "^", який ставиться перед відповідним типом. Попередні описи можна задати і так:

```
var v,w: ^t;
```

Тип `t` називають **базовим типом** покажчика. У ролі `t` може бути будь-який тип, окрім файлового. Крім адреси, значенням покажчика може бути `nil` (порожний покажчик-константа, що не вказує на жодне місце у пам'яті). Зауважимо, що типи покажчиків, які утворені від *різних* базових типів, вважаються *різними*.

Ще одне, самі покажчики `v` і `w` є **статичними** змінними, пам'ять для яких компілятор відводить у сегменті даних (покажчик займає **чотири байти** пам'яті).

Приклад 6.1. Опис **типів** покажчиків:

```
type pb=^byte; ps=^string; pd=^double;
```

`pb` визначає тип покажчика на змінні базового типу `byte`; `ps` – на змінні-рядки; `pd` – на змінні раціонального типу (`double`).

Приклад 6.2. Опис **покажчиків**:

```
var b1,b2:pb; s1:ps; d1: pd; vi: ^integer;
```

`b1,b2` – покажчики на змінні типу `byte`; `s1` – покажчик на змінні-рядки; `d1` – покажчик на змінні типу `double`, `vi` – покажчик на змінні цілого типу.

Покажчик може отримати *значення* (адресу байта пам'яті) одним з чотирьох способів:

- за допомогою унарної операції @ (взяття адреси), яка має вигляд @ `ідентифікатор`;
- за допомогою оператора присвоювання;
- за допомогою процедури `New`;
- за допомогою процедури `GetMem`.

Приклад 6.3. Отримання значень покажчиками:

```
type pb=^byte; pd=^double;
var  b1,b2:pb; d1, d1: pd; vi: ^ShortInt;
     i:byte; a: array[1..5] of byte;
     c: double; k: ShortInt;
begin
...
 b1:=@i; // значенням b1 є адреса змінної i
 b2:=@a[2]; // значенням b2 є адреса елемента масиву
 d1:=@c; // значенням d1 є адреса змінної c
 d2:= d1; // значенням d2 є адреса змінної c
 vi:=nil; // значенням vi є порожний покажчик
...
end.
```

Як уже зазначалося, значеннями покажчиків є адреси змінних у пам'яті, тому слід очікувати, що значення одного вказівника можна передавати (присвоювати) іншому. Насправді це не зовсім так. У Паскалі можна передавати значення тільки між вказівниками, які пов'язані з **одним і тим самим базовим типом даних** (у попередньому прикладі це присвоєння `d2:= d1;`). Присвоєння вигляду:

```
vi:=b1;
```

веде до помилки. Виходом з цієї ситуації може бути явне перетворення типу покажчика, як-от:

```
pb(vi):=b1;
```

Увага! Використовувати перетворення типу покажчика треба тільки у крайніх випадках. Адже базові типи покажчиків можуть мати різну довжину, а компілятор не контролює розмір ділянки пам'яті перетвореного покажчика.

Над значеннями покажчиків допускаються тільки операції порівняння на рівність/нерівність (перевіряють, чи вказують покажчики на одне і те ж саме місце у пам'яті). У порівнянні можуть брати участь лише покажчики одного типу. Порівнюючи покажчики різних типів, використовують явне перетворення типу.

6.3. Операція рознайменування

Важливою операцією при роботі з покажчиками є операція *рознайменування*, яка дає змогу отримати значення, що знаходиться у ділянці пам'яті, на яку вказує покажчик. Повернемося до прикладу 6.3. Тут, після присвоєння

```
b1:=@i;
```

для отримананння значення статичної змінної `i` є дві можливості:

- за допомогою ідентифікатора `i`;
- за допомогою адреси, яка є значенням покажчика `b1`.

Щоб отримати доступ до змінної через покажчик, треба використати операцію *рознайменування* (після покажчика записати символ "^"). У нашому випадку: `b1^` – **значення**, на яке вказує покажчик `b1`. Значення, яке отримують унаслідок операції рознайменування, має тип, що збігається з базовим типом покажчика (для `b1^` тип `byte`). Тепер, щоб збільшити, наприклад, значення змінної `i` на 5, можна використати один з трьох операторів:

```
i:=i+5; // доступ до i через ідентифікатор
b1^:= b1^+5; // доступ до i через покажчик
i:= b1^+5; // змішаний доступ
```

Іноді кажуть, що `b1^` – це змінна, на яку вказує покажчик `b1`. Цим самим підкреслюють той факт, що `b1^` може застосовуватися у програмі так само, як і звичайна змінна. Головне – це чітко розуміти,

що `b1^` повертає **значення**, яке зберігається за **адресою** в `b1`. Отримане значення має тип, який збігається з базовим типом `b1`.

Заборонено використовувати операцію рознайменування для покажчика, що має значення `nil`.

Використання операції рознайменування покажчиків для отримання значень статичних змінних є своєрідною *екзотикою*, адже набагато простіше скористатися для цього ідентифікаторами. Зазвичай операцію рознайменування покажчиків використовують для отримання значень *динамічних змінних*.

6.4. Динамічні змінні

Пам'ять під деяку динамічно розміщену змінну можна виділити за допомогою процедури `New(x)`, де `x` – типізований покажчик. У результаті звертання до цієї процедури покажчик набуває значення, що відповідає адресі, починаючи з якої розміщуються значення відповідної динамічної змінної. Таким чином, кожній динамічній змінній відповідає деякий покажчик і тоді можна сказати і так: `x^` – це динамічна змінна, на яку вказує покажчик `x`.

Динамічну пам'ять можна не тільки забирати з купи, але і повертати у купу. Для цього використовують процедуру `Dispose(x)`, де параметр `x` – покажчик, якому динамічна пам'ять була виділена. Процедура `Dispose(x)` не змінює значення покажчика `x`, а лише повертає в купу пам'ять, раніше виділену цьому покажчику. Цю процедуру потрібно використовувати щодо фіксованого покажчика лише раз, бо повторне використання процедури призведе до помилки періоду виконання.

Приклад 6.4. Виділення і звільнення динамічної пам'яті:

```
type v_int: ^integer; v_real: ^real;
var int1, int2: v_int; real1: v_real; c:integer;
begin new(int1); new(int2); new(real1);
        …
dispose(int1); dispose(int2); dispose(real1);
end.
```

Використання покажчиків, яким не присвоєно значення процедурою `New(x)` або іншим способом, не контролюється системою і може стати причиною непередбачуваних результатів.

Після того, як покажчик набув деякого значення, тобто став показувати на конкретний фізичний байт пам'яті, то за цією адресою можна розмістити будь-яке значення відповідного типу за допомогою операції рознайменування:

```
int1^:= 25; real1^:= sqrt(int1^);
```

Якщо за покажчиком немає символу "^", то цей покажчик надає адресу, за якою розміщені дані. Динамічно розміщені дані можна використовувати в будь-якому місці програми, де це допустимо для констант і змінних відповідного типу. Звичайно, **недопустимі** такі оператори:

```
int1:= c; real1^:=sin(int1)+int2;
```

У першому випадку покажчикові `int1` присвоюється *значення* змінної `c`, а в другому – значенню `real1^`, на яке вказує покажчик `real1`, присвоюється вираз, який містить функцію від покажчика `int1`.

При використанні великої кількості динамічних змінних може виникнути проблема нестачі динамічної пам'яті. У цьому випадку значення покажчика не зміниться і ніяких повідомлень не буде виведено. Така ситуація може призвести до непердбачуваних наслідків. Для підвищення надійності програм перед створенням нових динамічних змінних треба перевіряти наявність вільної динамічної пам'яті. Це можна робити за допомогою перехоплення і опрацювання виняткової ситуації `EOutOfMemory`.

У Delphi 1 і Паскалі для цієї мети використовують функції:

- `MaxAvail` – повертає максимальний розмір (у байтах) найбільшої вільної ділянки динамічної пам'яті;
- `MemAvail` – повертає сумарний розмір (у байтах) вільних ділянок динамічної пам'яті.

Ці дві функції загалом дають різні результати, оскільки динамічна пам'ять унаслідок довільного виділення і знищення є зазвичай поділена на зайняті та вільні ділянки.

Виділення/звільнення пам'яті для *динамічної змінної* можна ще реалізувати за допомогою процедур:

`GetMem(p, Size)` і `FreeMem(p, Size)`,

де p –покажчик (типізований або нетипізований); Size –константа цілого типу, що задає розмір пам'яті (у байтах), яку необхідно виділити чи звільнити; процедури GetMem і FreeMem – відповідно виділення пам'яті і звільнення пам'яті. Максимальне значення Size – 64 Кб. Зауважимо, що не можна поєднувати GetMem і Dispose чи New і FreeMem.

Для визначення розміру ділянки пам'яті, яка необхідна для розміщення динамічної змінної певного типу, зручно користуватися функцією SizeOf(тип). Ця функція повертає кількість байтів, необхідних для розміщення значень вказаного типу (стандартного чи визначеного користувачем).

Вивчаючи масиви (див. розділ 4), ми користуватилися математичним формулюванням задачі типу: "Дано натуральне число n і дійсні числа a_1, a_2,..., a_n. Знайти …". Число n стає відомим тільки під час виконання програми (вводиться з клавіатури).

У Паскалі при розміщенні масиву в статичній пам'яті *кількість елементів масиву, їхню упорядкованість і тип треба задавати явно до початку виконання програми*. Тому, якщо границі масиву заздегідь точно невідомі, то їх вибирають "із запасом" (це стосується переважно верхньої межі діапазону індексу). У наших програмах для цього використовувалася константа nmax.

Якщо кількість елементів масиву n набагато менша за nmax, то певна частина пам'яті "*простоює*". Для малих значень nmax і n це не страшно, але ситуація змінюється, якщо ці значення є достатньо великими, а елементи масиву теж мають великі розміри. Можна зменшити неефективні витрати пам'яті, якщо розміщувати такі масиви у динамічній пам'яті.

Для наочності за допомогою різних способів використання покажчиків (приклади 6.4 – 6.6) будемо розв'язувати одну і ту ж саму задачу прикладу 4.7 (стор. 60). Спочатку для опису масиву дійсних чисел використаємо *масив покажчиків.*

Приклад 6.4. Використання *масиву покажчиків*:

```
program Project6_4;
 {$APPTYPE CONSOLE}
const nmax = 30;
```

```
type vect=array [1..nmax] of ^real;
var a: vect; x: real; i, n, kmin, kmax: byte;
begin
 repeat // Введення кількості чисел
     write('Kilkist chysel<=30:');read(n);
 until (n<=nmax) and (n>=1);
 for i:=1 to n do New(a[i]); // Виділення пам'яті для масиву
 writeln(' Chysla: ');
 for i:=1 to n do read(a[i]^); // Введення чисел
 kmin:=1; kmax:=1;
 for i:=2 to n do // Пошук мінімуму та максимуму
  begin
   if a[kmin]^>a[i]^ then kmin:=i;
   if a[kmax]^<=a[i]^ then kmax:=i
  end;
 x:= a[kmin]^; a[kmin]^:= a[kmax]^; a[kmax]^:=x;
 for i:=1 to n do write(a[i]^:6:2,'  '); // Виведення
 readln(n);
 for i:=1 to n do Dispose(a[i]); // Звільнення пам'яті
end.
```

Масив покажчиків a на дійсні числа розміщується статично та займає фіксоване число байтів (nmax·4=30·4=120). Масив дійсних чисел розміщується у динамічній пам’яті та займає n·8 байтів. Якщо масив дійсних чисел розміщується у статичній пам’яті (див. приклад 4.7), то для нього резервується фіксоване число байтів (nmax·8). Якщо порівнювати витрати пам’яті при статичному і динамічному розміщенні масиву дійсних чисел (через використання масиву вказівників), то отримаємо таке:

$$nmax \cdot 8 \vee nmax \cdot 4 + n \cdot 8; \quad nmax \vee n \cdot 2;$$

- якщо $n < \frac{nmax}{2}$, то більшими є витрати пам’яті при статичному розміщенні масиву дійсних чисел;
- якщо $n > \frac{nmax}{2}$, то більшими є витрати пам’яті при динамічному розміщенні масиву дійсних чисел;

- якщо $n=\frac{nmax}{2}$, то витрати пам'яті є однаковими.

Використання *масиву покажчиків* не дає суттєвого поліпшення в економії пам'яті для довільних значень n (для $n>\frac{nmax}{2}$ є ліпшим статичне розміщення масиву). Приклад 6.4 має чисто пізнавальне значення і є корисним для відпрацювання техніки роботи з покажчиками.

Більшого ефекту в економії пам'яті можна досягнути, якщо використати типізований покажчик на масив дійсних чисел, який розмістимо в динамічній пам'яті за допомогою процедури GetMem.

Приклад 6.5. Використання *типізованого покажчика* на масив:

```
program Project6_5;
 {$APPTYPE CONSOLE}
const nmax = 30;
type vect=array [1..nmax] of real;
var p: ^vect; x: real; i, n, kmin, kmax: byte;
begin
 // Введення n
 GetMem(p,n*SizeOf(real)); // Виділення пам'яті
 writeln(' Chysla: ');
 for i:=1 to n do read(p^[i]);
  kmin:=1; kmax:=1;
 for i:=2 to n do
  begin
   if p^[kmin]>p^[i] then kmin:=i;
   if p^[kmax]<=p^[i] then kmax:=i
  end;
 x:= p^[kmin]; p^[kmin]:= p^[kmax]; p^[kmax]:=x;
 for i:=1 to n do write(p^[i]:6:2,'  ');
 FreeMem(p,n*SizeOf(real)); readln(n)
end.
```

Для розміщення масиву дійсних чисел у цьому прикладі задіяно $4+n\cdot 8$ байтів. Іншим способом створення динамічних змінних є використання *нетипізованих покажчиків*, які будуть розглянуті у наступному параграфі.

6.5. Нетипізовані покажчики

У Паскалі можна оголосити покажчик і разом з тим не зв'язувати його з яким-небудь конкретним типом даних. Для цього служить стандартний тип `Pointer`, наприклад:

```
var p: Pointer;
```

Покажчик такого роду називають *нетипізованим.* Оскільки нетипізовані покажчики не пов'язані з конкретним типом, то за їхньою допомогою дуже зручно розміщувати дані, структура і тип яких змінюються у ході виконання програми. У Паскалі можна передавати значення тільки між покажчиками, які зв'язані з одним і тим самим базовим типом даних. Якщо, наприклад,

```
var v1, v2: ^integer; vr: ^Real; p: Pointer;
```

то присвоєння `v1:=v2;` цілком допустиме, тоді як присвоєння `v1:=vr;` заборонене, оскільки `v1` і `vr` вказують на різні типи даних. Це обмеження, однак, не накладається на нетипізовані покажчики, тому для подолання вищевказаної проблеми можна записати: `p:=vr; v1:=p;` У цьому випадку присвоєння будуть виконані успішно і компілятор не видасть повідомлення про помилку.

Виділення/звільнення пам'яті для *динамічної змінної*, яка *зв'язується з нетипізованим покажчиком*, реалізують процедури:

`GetMem(p, Size)` і `FreeMem(p, Size)`,

де `p` – нетипізований покажчик; `Size` – константа цілого типу, що задає розмір пам'яті (у байтах), яку необхідно виділити чи звільнити; процедури `GetMem` і `FreeMem` – відповідно – виділення пам'яті і звільнення пам'яті. Максимальне значення `Size` – 64 Кб.

Використання цих процедур, як і всі операції з динамічною пам'яттю, потрібно робити обережно, дотримуючись такого правила: *звільняти потрібно рівно стільки пам'яті, скільки її було виділено, і саме з тієї адреси, з якої вона була виділена*. Наприклад, виклик `GetMem(p, 20)` виділяє для динамічної змінної ділянку пам'яті розміром 20 байтів і записує у `p` адресу першого байта цієї ділянки. Доступ до значень цієї змінної є можливим через операцію рознайменування з дальшим перетворенням отриманого значення до конкретного типу. Щоб звільнити цю ділянку пам'яті треба викликати `FreeMem(p, 20)`.

Приклад 6.6. Використання *нетипізованого покажчика* на масив дійсних чисел:

```
program Project6_6;
 {$APPTYPE CONSOLE}
const nmax = 30;
type vect=array [1..nmax] of real;
var p: pointer;
    min, max, x: real;
    i, n, kmin, kmax: byte;
begin
 repeat write('Kilkist chysel<=30:');read(n);
 until n<=nmax;
 GetMem(p,n*SizeOf(real));
 writeln(' Chysla: ');
 for i:=1 to n do read(vect(p^)[i]);
 min:=vect(p^)[1]; kmin:=1;
 max:=vect(p^)[1]; kmax:=1;
 for i:=2 to n do
  begin
   if min>vect(p^)[i] then
    begin min:=vect(p^)[i]; kmin:=i end;
   if max<=vect(p^)[i] then
    begin max:=vect(p^)[i]; kmax:=i end
  end;
 x:= vect(p^)[kmin];
 vect(p^)[kmin]:= vect(p^)[kmax];
 vect(p^)[kmax]:=x;
 for i:=1 to n do write(vect(p^)[i]:6:2,'  ');
 FreeMem(p,n*SizeOf(real));
 readln(n)
 end.
```

Для розміщення масиву дійсних чисел у цьому прикладі задіяно $4+n\cdot 8$ байтів пам’яті.

Створення матриці зі змінними розмірами є набагато складнішим процесом, адже посилаючись на рядки матриці, треба створити *масив* нетипізованих покажчиків, *який розміщують* у дина-

мічній пам’яті. Для більшої наочності розв’яжемо задачу прикладу 4.22 (стор. 73).

Приклад 6.7. Використання масиву нетипізованих покажчиків при роботі з матрицями:

```
program Project 6_7;
{$APPTYPE CONSOLE}
const nmax = 10;
type vect=array[1..nmax] of real;
     matr=array[1..nmax] of pointer;
var p: pointer; i, j, n: 1..nmax; max, s: real;
begin
 repeat write('Rozmir n<=10:');read(n);
 until n<=nmax;
 GetMem(p,n*SizeOf(real)); //Виділення пам'яті для
// масиву покажчиків на рядки – доступ через p
 for i:=1 to n do GetMem(matr(p^)[i],n*SizeOf(real));
 for i:=1 to n do // Заповнення матриці випад.числами
  for j:=1 to n do vect(matr(p^)[i]^)[j]:=random*10;
 max:=vect(matr(p^)[2]^)[1];
 for i:=3 to n do
  for j:=1 to i-1 do
   if vect(matr(p^)[i]^)[j]>max then
                    max:= vect(matr(p^)[i]^)[j];
 s:=0;
 for i:=1 to n do
  for j:=i to n do
   if vect(matr(p^)[i]^)[j]>max then
                    s:=s+vect(matr(p^)[i]^)[j];
  Write('s=', s:8:3);  readln(n)
end.
```

Отже, за допомогою покажчиків можна створювати масиви у динамічній пам’яті, виділяючи для них тільки потрібну кількість байтів. Однак необхідність рознайменування покажчиків і накладання типу робить програми важкими для читання і налагодження.

Виходом з цього може бути використання спеціальних *покажчиків на динамічні масиви*, які вперше були введені у Delphi 4.

6.6. Динамічні масиви

Одновимірний динамічний масив описують так:

```
var назва_масиву: array of тип;
```

назва_масиву – це *спеціальний покажчик*, який містить адресу байта, починаючи з якого розміщуються елементи масиву. Якщо місце під масив ще не виділене, то він має значення `nil`. До цього покажчика не можна застосовувати операцію рознайменування і процедури виділення/звільнення пам'яті. Наприклад:

```
var A: array of integer;
```

оголошує змінну `A` як динамічний масив цілих чисел.

При описі динамічного масиву місце у пам'яті для нього ще не виділяється. Розміщення динамічного масиву в пам'яті реалізовується процедурою `SetLength`, якій передаються два аргументи – назва масиву та кількість елементів, наприклад:

```
SetLength(A, 15);
```

виділяє для `A` місце у динамічній пам'яті під 15 цілих чисел і задає *нульові значення* усім його елементам.

Індекси динамічного масиву – це завжди цілі числа, що починаються з **нуля**. Отже, у наведеному прикладі масив містить елементи `A[0], A[1], … , A[14]`.

Повторне застосування `SetLength` до розміщеного у пам'яті динамічного масиву змінює його розміри. Якщо нове значення розміру є більшим за попереднє, то всі значення елементів зберігаються і до кінця масиву додаються нові нульові елементи. Якщо нове значення розміру є меншим за попереднє, то масив зберігає значення тільки перших елементів згідно з новим розміром, а решта видаляються.

Якщо динамічний масив уже розміщений у пам'яті, то до нього можна застосовувати стандартні функції `Length` – довжина масиву, `High` – найбільше значення індексу (`Length=High-1`) і `Low` – найменше значення індексу (завжди 0).

Видалити з пам'яті динамічний масив можна одним з цих способів: присвоїти йому значення `nil`, використати функцію `Finalize`, або встановити нульову довжину.

Тому ці оператори є еквівалентними:

`A:=nil;` або `Finalize(a);` або `SetLength(A, 0);`

Приклад 6.8. Використання динамічного масиву при розв'язанні задачі прикладу 4.7 (стор. 60):

```
program Project6_8;
 {$APPTYPE CONSOLE}
const nmax = 30;
type vect=array of real;
var a: vect;
    min, max, x: real;
    i, n, kmin, kmax: byte;
begin
 repeat
     write('Kilkist chysel<=30:'); read(n);
 until n<=nmax;
 SetLength(a, n);
 writeln(' Chysla: ');
 for i:=0 to n-1 do read(a[i]);
  kmin:=0; kmax:=0;
 for i:=1 to n-1 do
  begin
   if a[kmin]>a[i] then kmin:=i;
   if a[kmax]<=a[i] then kmax:=i
  end;
 x:= a[kmin]; a[kmin]:= a[kmax];
 a[kmax]:=x;
 for i:=0 to n-1 do write(a[i]:6:2,'  ');
 SetLength(a, 0);
 readln(n)
 end.
```

Багатовимірний динамічний масив визначають як динамічний масив динамічних масивів від динамічних масивів і т.д., як-от:

```
var A2: array of array of integer;
```

описано двовимірний динамічний масив.

Розмістити багатовимірний динамічний масив у пам'яті можна за допомогою процедури `SetLength`, якій передаються аргументи: назва масиву та декілька розмірів. Наприклад, оператор

```
SetLength(A, 3, 4);
```

задає матрицю з трьох рядків та чотирьох стовпців.

Приклад 6.9. Використання динамічної матриці при розв'язанні задачі прикладу 4.22 (стор. 73):

```
program Matrix6_9;
{$APPTYPE CONSOLE}
const nmax = 10;
type matr=array of array of real;
var a: matr; i, j, n: 0..nmax-1; max, s: real;
begin
 repeat write('Rozmir n<=10:');read(n);
 until n<=nmax;
 SetLength(a,n,n);
 for i:=0 to n-1 do
  for j:=0 to n-1 do a[i,j]:=random*10;
 max:=a[1,0];
 for i:=2 to n-1 do
  for j:=0 to i-1 do
   if a[i,j]>max then max:=a[i,j];
 s:=0;
 for i:=0 to n-1 do
  for j:=i to n-1 do
   if a[i,j]>max then s:=s+a[i,j];
 Write('s=', s:8:3);
 readln(n)
end.
```

Можна створювати і непрямокутні масиви, у яких другий розмір змінюється залежно від номера рядка.

Приклад 6.10. Створення і заповнення нижньої трикутної матриці:

```
program Project1;
{$APPTYPE CONSOLE}
uses SysUtils;
```

```
var A: array of array of string; I, J : Integer;
begin
 SetLength(A, 10);
 for I:=Low(A) to High(A) do
 begin
  SetLength(A[I], I);
  for J := Low(A[I]) to High(A[I]) do
   begin A[I,J]:=IntToStr(I) + ',' + IntToStr(J)+' ';
         write(A[I,J],' ')
   end; writeln
 end;
 readln
end.
```

6.7. Динамічні структури даних

У попередніх розділах і параграфах було розглянуто задачі, у яких необхідний для зберігання даних обсяг пам’яті був відомий або до компілювання програми, або до початку введення даних. У першому випадку пам’ять резервується за допомогою описів, у другому – за допомогою функцій виділення пам’яті.

Якщо до початку роботи з даними неможливо визначити, скільки пам’яті потрібно для їхнього зберігання, то пам’ять виділяється за потребою окремими *блоками*, які зв’язуються між собою покажчиками. Такі, взаємопов’язані покажчиками, блоки називають *динамічними структурами даних*. Слово “*динамічні*” у визначенні несе подвійне навантаження: такі структури даних розміщуються у динамічній пам’яті та їхній розмір змінюється під час виконання програми.

З динамічних структур даних у програмах найчастіше використовують різноманітні *лінійні списки*, *стеки*, *черги* і *бінарні дерева*. Вони відрізняються способами зв’язування окремих блоків та допустимими операціями над ними. Блоки динамічних структур даних часто називають **ланками**. У процесі роботи програми ланки можуть додаватися чи видалятися.

Список – це динамічна структура даних, кожна ланка якої складається з двох частин: *тіла* (інформаційної частини) і *поля зв’язку*. *Тіло* містить одне або декілька полів даних або покажчик на

ці поля. У *полі зв'язку* містяться адреси ланок, які визначають способи проходження списку. За кількістю цих адрес списки ділять на **одноелементні** (**лінійні** або **ланцюгові**) та **багатоелементні**. Лінійний список характеризує наявність єдиного зв'язку – з наступною ланкою (див. рис. 6.1).

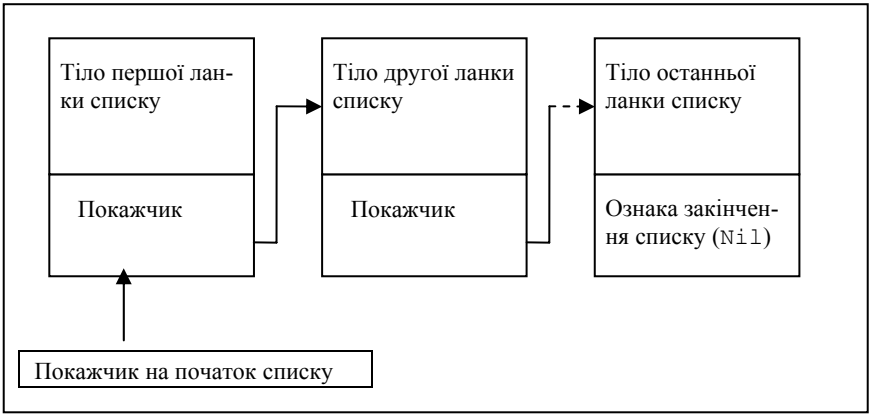


Рис. 6.1. Схематичне зображення лінійного списку

Кожна ланка списку, крім останньої, містить покажчик на наступну за нею ланку. Ознакою того, що ланка – остання у списку, є значення `nil` у полі зв'язку цієї ланки.

Для кожного списку обов'язково існує покажчик, що містить адресу першої ланки списку (*покажчик на початок списку*). Якщо список не має жодної ланки (*порожній* список), то значення цього покажчика дорівнює `nil`.

Зауважимо, що зображений на рис. 6.1 лінійний список є *однонапрямленим*, тоді як існують ще й *двонапрямлені* списки, тобто ті, у яких кожна наступна, крім початкової, ланка містить додатково покажчик і на попередню ланку списку. Під терміном "*лінійний список*" зазвичай розуміють саме "*однонапрямлений лінійний список*". Термін "*двонапрямлений лінійний список*", очевидно, трактується однозначно. Притримуючись традиційного розуміння, слово "*однонапрямлений*" будемо надалі опускати.

Для реалізації лінійних списків можна використати такі оголошення типів:

```
type TypTila= … // тип тіла ланки
  List= ^Listrecord;
  Listrecord = record Tilo:TypTila; next: List end;
```

Зауважимо, що тип тіла ланки можна і не виписувати окремо, а задавати безпосередньо при описі полів даних, як у прикладі 6.11.

Приклад 6.11. Опис типу лінійного списку цілих чисел:

```
type List= ^Listrecord;
     Listrecord = record a:integer; next: List; end;
```

У цьому прикладі тип `List` оголошується покажчиком на тип `Listrecordt` (запис), який попередньо у програмі не був оголошений. Навіть якщо поміняти місцями визначення типів `Listrecord` і `List`, то ситуація все одно не зміниться: у визначенні запису `Listrecord` міститься поле типу `List`.

Тому покажчики – це **єдиний виняток**, для якого допустиме звертання на ще не оголошений тип даних. Записи, в яких є посилання на аналогічні записи, називають *самоадресованими*. Отже, для формування списку визначено **спеціальний шаблон** використання покажчиків та самоадресованих записів.

Місця вставляння/вилучення ланок списку чи отримання значень тіл ланок жорстко не фіксуються і залежать від задачі, що розв'язується. Робота зі списками потребує доброго розуміння матеріалу та великої уваги. У наступних програмах цього розділу, в яких працюватимемо зі списками, розглянемо підпрограми (процедури і функції), які матимуть *універсальне* значення. Ці підпрограми досить просто буде пристосувати до розв'язку різноманітних задач, змінюючи тільки тіло ланки списку. Для зручності розгляду матеріалу **зафіксуємо** назви підпрограм і назви параметрів: `head` – параметр, який задає покажчик на початок списку типу `List`; `a_p:integer` – параметр, який задає/отримує тіло ланки списку:

- **`function`** `IsEmpty(head:List):boolean;` – повертає `true`, якщо список порожній;
- **`procedure`** `InBegin(`**`var`**` head:List; a_p:integer);` – вставляє нову ланку на початок списку;

- **procedure** `OutBegin(`**var** `head:List;` **var** `a_p:integer);` – звертання до ланки на початку списку;
- **procedure** `InEnd(`**var** `head:List; a_p:integer);` – вставляє нову ланку на кінець списку;
- **procedure** `OutEnd(`**var** `head:List;` **var** `a_p:integer);` – звертання до ланки у кінці списку;
- **procedure** `output(`**var** `head: List);` – виведення усього списку на екран;
- **procedure** `delall(`**var** `head:List);` – видалення усього списку з пам'яті.

Стек і *черга* – це динамічні структури даних, у яких жорстко фіксуються місця вставляння і вилучення ланок. Особливістю цих структур даних є те, що ланка *перебуває* у них **тільки** *до* моменту *звертання*, яке полягає у тому, що:

- тіло ланки використовують для потреб реалізації алгоритму (зазвичай присвоюють певним змінним);
- ланку видаляють.

Занесення даних у *стек* чи звертання до ланки стека відбувається **тільки з одного боку** динамічної структури даних. Занесення даних у *чергу* відбувається **з одного боку** динамічної структури даних, а звертання до ланки черги – з **іншого боку**.

Динамічні структури даних можна зберігати у пам'яті за допомогою вектора або списку (часом кажуть, що динамічна структура даних *моделюється вектором або списком*). У посібнику ці структури моделюються за допомогою лінійних списків.

Обсяг книги не дає змоги детально зупинитися на розгляді усіх динамічних структур даних. Тому розглянемо тільки роботу з однонапрямленим списком, як універсальної структури даних, і його частковим випадком – **стеком**.

У процедурах `OutBegin/OutEnd` тіло ланки (ціле число) передається у точку виклику через параметр-змінну `a_p`, а сама ланка видаляється, тому ці процедури можуть використовуватися для звертання до стека. Аналогічно, процедури `InBegin/InEnd` можуть використовуватися для вставляння нової ланки стека.

Занесення даних у стек чи звертання до ланки стека може відбуватися тільки на/у:

- *початку списку* (дозволено використання тільки підпрограм `InBegin`, `OutBegin` та `IsEmpty`);
- *кінці списку* (дозволено використання тільки підпрограм `InEnd`, `OutEnd` та `IsEmpty`).

Лінійні списки дають можливість поєднувати різні підпрограми, що дозволяє звернутися до будь-якої ланки списку чи переглянути весь список без його знищення тощо.

Зауважимо, що для спрощення роботи з *чергою* її часто моделюють за допомогою лінійного списку з двома покажчиками (на початок і на кінець списку, див. [7, 10, 26]).

6.8. Реалізація лінійних списків

У цьому параграфі розглянемо приклади, які дадуть можливість на належому рівні вивчити техніку роботи зі списками. Щоб не відволікатися на зайві деталі, спочатку за допомогою списків моделюватимемо послідовність цілих чисел a_1, a_2,..., a_n, де `n` – натуральне число. Для цієї послідовності використаємо список типу `List` (див. приклад 6.11).

Приклад 6.12. З клавіатури вводять послідовність цілих чисел a_1, a_2,..., a_n, де `n` – натуральне число. Необхідно:

1) вивести цю послідовність на екран у зворотному порядку;
2) видалити з послідовності два числа, які введені останніми, з відповідним повідомленням;
3) видалити послідовність з пам’яті.

Попередні міркування. Найлегше буде виконати перші два завдання, якщо у процесі введення чисел буде формуватися *стек* із занесеням чисел чи звертанням до ланок на початку списку. При виведенні послідовності чисел на екран весь список переглядатиметься без знищення його ланок (тут список використовується у звичному для цієї структури значенні). Послідовно переглядається весь список і при видаленні його з пам’яті (знищення списку), оскільки кожна ланка видаляється окремо. Отже, на основі одного і того ж списку у програмі для різних цілей використовуються різні

структури даних: стек і лінійний список. Вважатимемо, що у програмі на початок цього списку посилається змінна `Chain` типу `List`, а відповідний формальний параметр, який отримуватиме адресу змінної `Chain`, матиме назву `head`.

Зауважимо, що про стек можна було б тут взагалі й не згадувати. Однак просте і ефективне (ефектне!) розв'язання тих чи інших задач великою мірою залежить від структури даних (див. [7]). А завдання 1-2 і асоціюються у більшості читачів саме зі стеком, воістину: "*як ти яхту назвеш, так вона і попливе*". Всі інші необхідні пояснення наведені у коментарях програми.

```
program Project6_12;
{$APPTYPE CONSOLE}
uses SysUtils;
const nmax=30;
type List = ^ListRecord;
 ListRecord = record a:integer; next:List; end;
function IsEmpty(head:List):boolean;
 begin   IsEmpty:= head=nil   end;
procedure InBegin(var head:List; a_p:integer);
 var node: List;  // нова ланка списку
begin new(node); // розміщення нової ланки
    node^.a:=a_p; node^.next:=head;
    head:=node  // додавання нової ланки у список
end;
procedure OutBegin(var head:List; var a_p:integer);
 var node: List;  // допоміжна ланка
begin
  if IsEmpty(head) then // список порожний
        begin write ('Chain is empty!'); exit end;
  node:=head; a_p:=head^.a;  head:=head^.next;
  Dispose(node) // допоміжна ланка
end;
procedure output(var head: List);
var curr: List;  // поточна ланка списку
    st:string; // представлення списку рядком
begin
 if IsEmpty(head) then // список порожний
        begin write ('Chain is empty!'); exit end;
```

```
 st:=''; curr:=head;
 while curr <> NIL do
  begin st:=st+IntToStr(curr^.a)+' ';
        curr:=curr^.next;
  end;
 writeln(st);
end;
procedure delall(var head:List);
 var curr:List; // поточна ланка списку
begin
 if IsEmpty(head) then // список порожний
        begin write ('Chain is empty!'); exit end;
 // послідовно знищуємо ланки
 curr:=head; head:=head^.next;
 while (head<>nil) do
 begin dispose(curr); curr:=head; head:=head^.next end
end;
// реалізація задачі
var Chain:List; a_new:integer; i,n:byte; st:string;
begin Chain:=nil;
 repeat write('Kilk.chysel:'); readln(n)
 until (n>=1) and (n<=nmax);   // введення n
 writeln('Chysla:');
 // введення послідовності чисел з формуванням стека
 for i:=1 to n do
  begin read(a_new); InBegin(Chain,a_new) end;
 // завдання 1
  output(Chain); readln;
 // завдання 2
 st:='Vylucheno:';
 for i:=1 to 2 do
  begin OutBegin(Chain, a_new);
        st:=st+IntToStr(a_new) + '  '  end;
  writeln(st);
 // для контролю виводимо ланки, що залишилися
  output(Chain); readln;
 // завдання 3
 delall(Chain)
end.
```

Приклад 6.13. З клавіатури вводять послідовність цілих чисел a_1, a_2,..., a_n, де n – натуральне число. Необхідно:

1) вивести цю послідовність на екран у порядку введення чисел;
2) видалити з послідовності два числа, які введені останніми, з відповідним повідомленням;
3) послідовно вивести на екран: a_1, a_1+a_2, ... , $a_1+a_2+...+a_{n-2}$.

Попередні міркування. Найлегше буде виконати перші два завдання, якщо у процесі введення чисел буде формуватися *стек* із занесенням чисел чи звертанням до ланок у кінці списку. При виконанні третього завдання треба організувати звертання до ланок на початку списку.

```
program Project6_13;
{$APPTYPE CONSOLE}
uses SysUtils;
const nmax=30;
type List = ^ListRecord;
 ListRecord = record a:integer; next:List; end;
function IsEmpty(head:List):boolean;
 begin IsEmpty:= head=nil end;
procedure InEnd(var head:List; a_p:integer);
var node, curr: List; // нова і поточна ланка списку
begin new(node); // розміщення нової ланки
  node^.a:=a_p; node^.next:=nil;
  // вставляння ланки у кінець списку
  if head=nil then head:=node
   else  // пошук останьої ланки списку
    begin curr:=head;
      while curr^.next <> NIL do curr:=curr^.next;
      // додавання нової ланки у кінець списку
      curr^.next:=node
    end
end;
procedure OutEnd(var head:List; var a_p:integer);
var curr: List;  // поточна ланка списку
begin
  if IsEmpty(head) then // список порожний
        begin write ('Chain is empty!'); exit end;
  curr:=head;
```

```
  // пошук передостанньої ланки списку
  while curr^.next^.next <> NIL do curr:=curr^.next;
  a_p:=curr^.next^.a;
  Dispose(curr^.next) // знищення останньої ланки
  // передостання ланка списку стає останньою
  curr^.next:=NIL;
end;
procedure OutBegin(var head:List; var a_p:integer);
var node: List;
begin  { див. проект Project6_12}  end;
procedure output(var head: List);
var curr: List; st:string;
begin  { див. проект Project6_12}  end;
// реалізація задачі
var Chain:List; s,a_new:integer; i,n:byte;st:string;
begin Chain:=nil;
 // введення кількості чисел
 repeat write('Kilk.chysel:'); readln(n)
 until (n>=1) and (n<=nmax);
 writeln('Chysla:');
 // введення послідовності чисел з формуванням стека
 for i:=1 to n do
  begin read(a_new); InEnd(Chain,a_new) end;
 output(Chain); readln;  // завдання 1
 // завдання 2
 st:='Vylucheno:';
 for i:=1 to 2 do
  begin OutEnd(Chain, a_new);
        st:=st+IntToStr(a_new) + '  '
  end;
 writeln(st); readln;
 // завдання 3
 s:=0;
 while Chain<> nil do
  begin OutBegin(Chain, a_new);
        s:=s+a_new; write(s, ' ')
  end;
 readln
end.
```

Приклад 6.14. Використання списку при розв'язанні задачі прикладу 4.7 (стор. 60):

```
program Project6_14;
{$APPTYPE CONSOLE}
uses SysUtils;
const nmax=30;
type List = ^ListRecord;
 ListRecord = record a:integer; next:List; end;
function IsEmpty(head:List):boolean;
begin IsEmpty:= head=nil end;
procedure InEnd(var head:List; a_p:integer);
var node, curr: List;
begin  { дивись проект Project6_13 }  end;
procedure output(var head: List);
var curr: List;  st:string;
begin  { дивись проект Project6_12 }  end;
procedure delall(var head:List);
var curr:List;
begin {дивись проект Project6_12} end;
procedure PozMin(var head: List; var p_min: List);
var curr: List;  // поточна ланка списку
begin
 if IsEmpty(head) then // порожний список
       begin write ('Chain is empty!'); exit end;
 p_min:=head; curr:=head^.next;
 while curr <> NIL do // шукаємо 1-шу позицію мінімуму
  begin if p_min^.a>curr^.a then p_min:=curr;
        curr:=curr^.next;   end
end;
procedure PozMax(var head: List; var p_max: List);
var curr: List;  // поточна ланка списку
begin
 if IsEmpty(head) then // порожний список
        begin write ('Chain is empty!'); exit end;
 p_max:=head; curr:=head^.next;
 while curr <> NIL do // шукаємо останню позицію максимуму
  begin if p_max^.a<=curr^.a then p_max:=curr;
        curr:=curr^.next;    end
end;
```

```
var kmin,kmax, Chain:List; a_new:integer; i,n:byte;
    st:string;
begin  Chain:=nil;
 // введення кількості чисел
 repeat write('Kilk.chysel:'); readln(n)
 until (n>=1) and (n<=nmax);
 // введення послідовності чисел з формуванням списку
 writeln('Chysla:');
 for i:=1 to n do
  begin read(a_new); InEnd(Chain,a_new) end;
 output (Chain); readln;
 // знаходження позицій мінімуму та максимуму
 PozMin(Chain,kmin); PozMax(Chain,kmax);
 // обмін значень мінімуму та максимуму
 a_new:=kmin^.a; kmin^.a:=kmax^.a; kmax^.a:= a_new;
 output(Chain); readln;
 delall(Chain)
end.
```

Для ліпшого розуміння динамічних структур даних опишемо ще один тип лінійного списку студентів:

Приклад 6.15. Опис типу лінійного списку студентів:

```
type
 TPStudent=^TStudent;
 TStudent =
 record
   f_name:string; l_name: string; next: TPStudent;
 end;
```

Об'єктами типу `TStudent` є записи, одне поле в яких (поле `next`) є покажчиком на конкретне місце у пам'яті, відведене для чергового об'єкта типу `TStudent`, або ж цей покажчик містить `nil`. У `TStudent` кожна ланка списку містить два інформаційні поля (`f_name` – ім'я студента та `l_name` – прізвище студента).

Приклад 6.16. З клавіатури у довільному порядку вводять імена та прізвища студентів. Необхідно посортувати список студентів відповідно до *алфавітного впорядкування прізвищ* студентів та вивести цей список на екран.

Попередні міркування. Для формування списку студентів використаємо тип `TStudent`, який описано у прикладі 6.15. При **додаванні** нової ланки будемо застосовувати процедуру `InEnd`. Для поповнення набору стандартних процедур опрацювання списку створимо процедуру `Sortlist`, яка сортуватиме список прізвищ студентів методом *включення* за принципом алфавітного впорядкування. Процедуру `Sortlist` досить просто буде пристосувати для сортування інших списків, змінюючи тільки тіло ланки списку.

```
• program Project6_16;
• {$APPTYPE CONSOLE}
• uses SysUtils;
• const nmax=30;
• type TPStudent=^TStudent;
•  TStudent = record
•   f_name:string; l_name: string; next:
  TPStudent end;
• function IsEmpty(head:TPStudent):boolean;
 begin IsEmpty:= head=nil end;
procedure InEnd(var head:TPStudent; f_n,l_n: string);
• var node, curr: TPStudent;
• begin new(node); node^.f_name:=f_n;
  node^.l_name:=l_n;
•  node^.next:=nil;
 if head=nil
  then head:=node
•    else begin curr:=head;
•          while curr^.next <> NIL do
  curr:=curr^.next;
•          curr^.next:=node
  end;
end;
• procedure output(var head: TPStudent);
• var curr: TPStudent; st:string;
begin if IsEmpty(head) then
    begin write ('Chain is empty!'); exit end;
•  st:=''; curr:=head;
•  while curr <> NIL do
```

```
  begin st:=st+curr^.f_name+' '+curr^.l_name+#13+#10;
      •          curr:=curr^.next;
  end;
      •   write(st)
      •  end;
      •  procedure delall(var head:TPStudent);
      •  var curr:TPStudent;
      •  begin if IsEmpty(head) then
      •          begin write ('Chain is empty!');
         exit end;
      •   curr:=head; head:=head^.next;
      •   while head<>nil do
      •    begin dispose(curr);
         curr:=head;head:=head^.next end
      •  end;
      •  procedure Sortlist(var head:TPStudent);
var node,curr,rob:TPStudent;
      •  begin new(node); // тимчасова ланка
      •   node^.next:=head; head:=head^.next;
 // head показує на невпорядковану частину списку
      •   node^.next^.next:=nil; // розмістили першу
         ланку у
 // новому впорядкованому списку, на який показує node
      •   while head<>nil do
      •    begin curr:=node; // пошук місця вставки у
         списку node
      •     while (curr^.next<>nil) and
      •
         (curr^.next^.f_name<head^.f_name)
      •      do curr:=curr^.next; // до наступної
         ланки
      •     if  curr^.next<>nil then // вставка у
         середину node
      •        begin rob:=head;head:=head^.next;
      •          rob^.next:=curr^.next;
         curr^.next:=rob  end
      •     else // вставка у кінець списку node
```

```
•        begin curr^.next:=head;
  head:=head^.next;
•              curr^.next^.next:=nil   end
•  end; // while
•  head:= node^.next; // показує на
  впорядкований список
•  Dispose(node) // вилучення тимчасової ланки
•  end;
•  var Chain:TPStudent; f_new,l_new: string;
  i,n:byte;
•  begin Chain:=nil; repeat
  write('Kilk.studentiv:');
•                    read(n) until (n>=1) and
  (n<=nmax);
•  write('Prizv. + <enter>; Im. + <enter>:');
  readln;
for i:=1 to n do
•   begin readln(f_new); readln(l_new);
•         InEnd(Chain,f_new,l_new)   end;
•  output (Chain); readln;  Sortlist(Chain);
•  output (Chain); readln; delall(Chain)
•  end.
```

- **? Запитання для самоперевірки**

1. Що розуміють під статичним розміщенням даних?
2. Що розуміють під динамічним розміщенням даних?
3. Що таке покажчик? Як описують покажчик?
4. Вкажіть на особливості присвоєння значень покажчикам.
5. Для чого використовують операцію рознайменування?
6. Для чого використовують процедури `New` і `Dispose`?
7. Для чого використовують процедури `GetMem` і `FreeMem`?
8. Як виділяють і звільняють пам’ять для динамічного масиву?
9. Що таке самоадресований запис? Що таке список?
10. Дайте визначення стека як динамічної структури даних.
11. Дайте визначення черги як динамічної структури даних.

- **Завдання для програмування**

Завдання 6.1. Виконати *завдання 4.1* (стор. 86), використавши *типізований покажчик* для створення масиву в динамічній пам'яті (аналог прикладу 6.5).

Завдання 6.2. Виконати *завдання 4.2* (стор. 86), утворивши список.

Завдання 6.3. Виконати *завдання 4.3* (стор. 88), уворивши динамічну матрицю (аналог прикладу 6.9).

Завдання 6.4. Розв'язати одну з наведених нижче задач: а) студенти з номерами 1 – 5; б) студенти з номерами 6 – 10; в) студенти з номерами 11–15. Додатково кожний студент має запрограмувати ще одне завдання (номер додаткового завдання вибирати відповідно до номера студента у списку студентів підгрупи).

а) Скласти програму, яка містить поточну інформацію про **замовлення** на авіаквитки (`пункт призначення, номер рейсу, прізвище та ініціали пасажира, дата вильоту`). Програма має забезпечувати: збереження усіх замовлень у вигляді списку; додавання та вилучення замовлень; виведення усіх замовлень на екран.

Додаткове завдання:

1) за номером рейсу і датою вильоту виводити список замовлень з вилученням їх зі списку;
2) упорядкувати список за зростанням дат, а кожну дату додатково впорядкувати за спаданням номерів рейсів;
3) упорядкувати список за спаданням кількості заявок на окремі рейси (для рейсу враховувати усі дати вильоту);
4) для конкретного рейсу виводити список заявок за зростанням дат без видалення їх зі списку;
5) упорядкувати список за зростанням номерів рейсів, а для кожного номера рейсу додатково впорядкувати за спаданням дат.

б) Скласти програму, яка містить інформацію про **анкетування** населення. Анкета містить дані про респондента (`вік: до 30/від 30 до 40/після 40, стать, освіта: початкова/середня/вища`) та відповідь на питання (`так/ні`). Програма має забезпечувати: збереження усіх анкет у вигляді списку; додавання та видалення анкет; виведення усіх анкет на екран.

Додаткове завдання:

6) встановити кількість мужчин, які мають більше 40-ка років і вищу освіту, що відповіли "*так*";
7) встановити кількість жінок, які мають менше 40-ка років і середню освіту, що відповіли "*ні*";

7. СИСТЕМА ВІЗУАЛЬНОГО ПРОГРАМУВАННЯ DELPHI 6

План викладу матеріалу

1. Інтегроване середовище Delphi 6
2. Побудова проекту віконного застосування
3. Структура проекту віконного застосування

Ключові терміни розділу

- *Інтегроване середовище*
- *Етапи побудови проекту*
- *Інспектор об'єктів*
- *Текстові редактори*
- *Методи опрацювання подій*
- *Модуль форми*
- *Файл параметрів проекту*
- *Основні вікна середовища*
- *Компонент* (*властивості, події*)
- *Текстовий підпис* (*позначка*)
- *Кнопка*
- *Головний модуль проекту*
- *Текстовий файл форми*
- *Файл ресурсів проекту*

7.1. Інтегроване середовище Delphi 6

Інтегроване середовище Delphi 6 (надалі *середовище*) – це програмний продукт, у якому є все необхідне для проектування, запуску і тестування застосувань (прикладних програм) під Windows мовою Object Pascal. Програму Delphi 6 запускають за правилами операційної системи Windows (вибираючи піктограму чи через меню **Пуск**). Насамперед ознайомимося з основними вікнами (див. рис. 7.1):

1. *Головне вікно* розміщене зверху і є центральним елементом керування під час роботи Delphi 6. Воно складається з *рядка меню*, *палітри компонентів* і *панелі швидкого доступу*.

2. *Вікно інспектора об'єктів* розміщене ліворуч знизу екрана. Інспектор об'єктів складається з двох сторінок: перша сторінка – це *список властивостей* (Properties), а друга – *список подій* (Events). Інформація на цих сторінках змінюється залежно від компонента, виділеного у вікні форми. Важливо зрозуміти, що кожний компонент є об'єктом і можна змінювати його вигляд і поведінку за допомогою інспектора об'єктів або миші.

3. *Вікно форми* знаходиться праворуч під головним вікном, воно є робочою поверхнею для розміщення елементів керування програмою під час її майбутнього виконання (створення *візуального інтерфейсу* користувача). Вікно форми спочатку скла-

дається з одного порожнього вікна, яке у ході проектування застосування заповнюється всілякими об’єктами (компонентами), обраними на палітрі компонентів.

4. *Вікно редактора коду* дає змогу вводити та редагувати тести програм мовою Object Pascal. Воно розміщене праворуч внизу (може бути прихованим під вікном форми) і є багатосторінковим (на кожній сторінці можна редагувати інший текстовий файл програми). Зауважимо, що сторінка *подій* (`Events`) інспектора об’єктів зв’язана з редактором коду. Якщо двічі клацнути мишкою справа від деякої події, то відповідний цій події код автоматично запишеться у редакторі коду, а сам редактор негайно одержить фокус. Це дає можливість відразу сформувати код процедури опрацювання події.

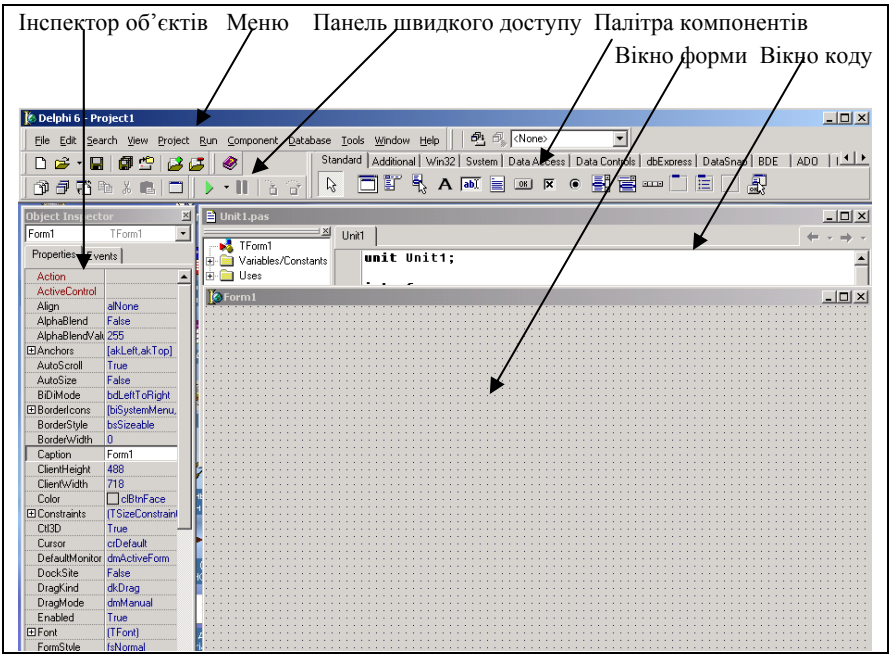


Рис. 7.1. Інтегроване середовище розробки Delphi 6

Рядок меню надає швидкий і гнучкий інтерфейс до середовища Delphi 6, тому що може керуватися набором “гарячих клавіш”. Це зручно ще і тому, що у меню використовуються слова або

короткі фрази, які є більш точними і зрозумілими, ніж іконки панелі швидкого доступу.

Палітра компонентів дає можливість вибирати потрібні об'єкти (компоненти) для розміщення їх на формі. Для використання палітри просто треба перший раз клацнути мишкою на одному з об'єктів і потім другий раз – на формі. Обраний компонент з'явиться на проектованому вікні форми і ним далі можна маніпулювати за допомогою мишки. Розміщений на формі компонент можна переміщати по формі чи змінювати його розміри, використовуючи маркери, які обрамляють виділений компонент.

Палітра компонентів групує об'єкти на сторінках. На палітрі знаходиться набір закладок: *Standard*, *Additional*, *Dialogs* і т.д. Якщо клацнути мишкою на одній із закладок, то можна перейти на відповідну сторінку палітри компонентів. Принцип розбивання на сторінки широко використовується у середовищі Delphi 6 і його можна легко використовувати у прикладній програмі (на сторінці *Additional* є компоненти для організації сторінок із закладками зверху або знизу).

Панель швидкого доступу знаходиться безпосередньо під меню, ліворуч від палітри компонентів, і містить набір кнопок у вигляді іконок (піктограм). Кнопки дають швидкий доступ до окремих команд меню. Якщо затримати мишку над іконкою на панелі, то можна побачити підказку, що пояснює призначення цієї кнопки. На панелі можна розміщувати кнопки команд меню за власними потребами або вилучити маловикористовувані кнопки. Для прикладу доповнимо панель швидкого доступу кнопкою виклику команди меню **Copy** (копіювати) з категорії **Edit**:

❑**ПКМ** //над панеллю швидкого доступу – отримуємо контекстне меню

➢**Customize … 🗀Commands ⇩Categories ⇗ Edit ⇩Commands ⇗ Copy**

// Зафіксувавши кнопку Copy перетягуємо її на панель швидкого доступу

❑**Close**

Для вилучення кнопки з панелі швидкого доступу необхідно (при відкритому вікні **Customize**) перетягнути її за межі панелі.

Важливою частиною середовища Delphi 6 є *довідник*. Для доступу до нього потрібно вибрати у рядку меню пункт *Help*. Довід-

ник є контекстозалежним; натискаючи клавішу F1, можна отримати підказку, яка відповідає поточній ситуації.

Вікна форми та редактора коду під час запуску Delphi 6 можуть перекриватися, а тому перед початком роботи їх треба налаштувати під власні потреби. Після запуску зазвичай обирають один з двох варіантів:

1. Створення нової програми (вірніше, нового *проекту*); виконавши команди ➢**File** ➢**New Application**, отримаємо нове порожне вікно форми.
2. Робота з наявним проектом; виконавши команди ➢**File** ➢**Open Projects…**, отримаємо стандартне вікно відкриття файлів (файли проектів мають розширення .**bpr**). Після відкриття файлу проекту отримаємо розроблену раніше форму та відповідний програмний код. Усі застосування Delphi 6 мають інтерфейс користувача на основі вікон. На початку роботи над новим проектом вікно форми, як уже зазначалося, є порожним, однак воно уже володіє всіма загальними властивостями вікон Windows щодо розміщення та розміру, а також має кнопки максимізації, мінімізації та відновлення, закривання вікна.

7.2. Побудова проекту віконного застосування

Процес побудови будь-якого *проекту* (синоніми: *застосування*, *прикладна програма*) у Delphi 6 складається з таких основних етапів:

1. Проектування інтерфейсу користувача (етап візуального програмування).
2. Програмування процедур опрацювання подій та інших необхідних підпрограм (етап кодування).
3. Налагодження і запуск проекту на виконання.

Зрозуміло, що це дуже загальна схема, яка буде уточнюватися (зокрема, у наступному параграфі детальніше розглянемо виконання третього етапу). Після першого запуску може виникнути необхідність внесенння певних уточнень і змін, а тому треба буде знову виконати перші два етапи побудови проекту. Загалом побудова проекту у Delphi 6 є ітераційним процесом виконання

етапів 1–3. Головне, що Delphi 6 надає для побудови проекту прості та зручні засоби.

У термінології візуального програмування **об'єктами** проекту (**компонентами**) є діалогові вікна та елемети керування у них – командні кнопки, позначки, поля введення текстової інформації, перемикачі, меню, панелі тощо. **Власивостями** об'єкта є його розміри (довжина і висота), координати розміщення на формі, назва у програмному коді, підпис, шрифт тощо. **Подія** – це те, що з об'єктом відбувається під час роботи програми. Кожна подія має зафіксовану назву: `onClick`– клацання на об'єкті кнопкою миші; `onKeyDown` – натиснення клавіші тощо. Реакцією на подію є виконання певної дії (закриття вікна, обчислення за формулами, виведення повідомлення тощо). Реакція на подію у Delphi 6 реалізується через **процедуру опрацювання події**.

Для створення найпростіших віконних проектів будемо використовувати компоненти введення/виведення даних (*позначку* і *текстові редактори*) і компонент ініціалізації дій (*командну кнопку*).

Текстовий підпис або позначка є компонентом класу **`TLabel`**, який використовують для розміщення на формах та інших контейнерах тексту, що не змінюється користувачем. Основна властивість – **`Caption`** типу **`String`**, яка відображає текст позначки. Зазвичай текст є поясненням (розгорнутою назвою) іншого компонента. Дуже часто позначку використовують і для відображення результатів роботи програми. Детальніше про позначки див. 12.1.2.

Компонент класу **`TEdit`** (однорядковий текстовий редактор або поле введення) – відображає на формі вікно для введення і редагування одиночного рядка тексту. Введений рядок є значенням властивості **`Text`** типу **`String`**. Детальніше про поле введення див. 12.1.4.

Компонент класу **`TMemo`** є вікном редагування багаторядкових текстів. Зміст області редагування визначає масив рядків, який є значенням властивості **`Lines`**, що має тип **`TStrings`**. Детальніше про багаторядкові текстові редактори див. 12.1.5.

Компонент класу **`TButton`** є стандартною командною кнопкою Windows, яка ініціалізує якусь дію. Напис на кнопці задається

властивістю **`Caption`**. Основна подія кнопки – **`OnClick`**, що виникає при натисканні кнопки. Детальніше про кнопку див. 12.2.1.

При розміщенні на формі компонент отримує стандартну назву, як-от: `Label1`, `Label2`, ..., `Edit1`, `Edit2`, ..., `Button1` і т.п. Ця назва зберігається у властивості `Name` компонента. У великих проектах (див. 12.6) назви компонентів зазвичай змінюють на осмисленіші, які відповідають призначенню компонента у проекті. Для економії часу в більшості наших навчальних проектів цього робити не будемо. У програмах саме за назвою компонента здійснюється посилання на нього. Для звертання до властивості у програмі використовують складну назву вигляду **`назва_компонента.властивість`**, наприклад:

```
Label1.Caption:= 'Це повідомлення!';
```

Тепер дослідимо глибше інспектор об'єктів (Object Inspector). Основне, у чому полягає розуміння інспектора об'єктів, – це те, що він використовується для зміни властивостей будь-якого компонента, розміщеного на формі під час проектування інтерфейсу користувача. Крім цього, інспектор об'єктів використовується і для зміни властивостей самої форми. Найліпший шлях для вивчення інспектора об'єктів – попрацювати з ним.

Побудуємо **проект** віконного застосування (див. рис. 7.2) розв'язання прикладу 3.11 на сторінці 45 (вияснити, чи уведений рядок є паліндромом). На формі розміщено дві позначки, дві командні кнопки і поле введення. Після натиснення кнопки з написом "*Аналіз*" перевірятиметься рядок, введений користувачем у поле введення, чи він є паліндромом. Кнопка "*Вихід*" закриває вікно форми.

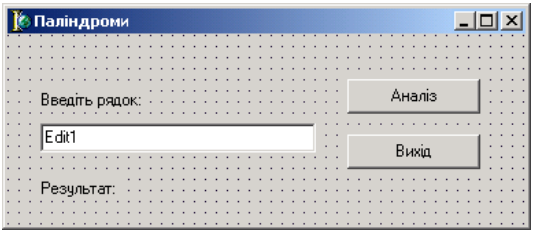


Рис. 7.2. Форма для розв'язання прикладу 3.11

Перший етап – *проектування інтерфейсу користувача*.

Оскільки у кінцевому підсумку проект – це множина різних спеціальних файлів (див. наступний параграф), то передусім не-

обхідно **створити каталог** (папку) для зберігання цих файлів. Автор на цьому питанні, з поваги до читачів, зупинятися не буде.

Виконавши команди ➢**File** ➢**New Application**, отримаємо нове порожне вікно форми (`Form1`). Далі послідовно виконаємо такі дії:

- змінимо за допомогою інспектора об'єктів стандартний заголовок форми "*Form1*" (значення властивості `Caption`) на заголовок "*Паліндроми*";
- на формі розміщуємо позначку `Label1` і змінимо значення її властивості `Caption` на "*Введіть рядок*:". Для цього виділимо позначку, клацнувши на ній мишкою. Після цього на сторінці *Properties* інспектора об'єктів будуть відображатися властивості цього компонента. У полі введення справа від властивості **Caption** уведемо текст "*Введіть рядок*:" (без лапок);
- на формі розміщуємо позначку `Label2` і змінимо значення її властивості `Caption` на "*Результат*:";
- на формі розміщуємо поле введення `Edit1`;
- на формі розміщуємо командну кнопку `Button1` і змінимо значення її властивості `Caption` на "*Аналіз*";
- на формі розміщуємо командну кнопку `Button2` і змінимо значення її властивості `Caption` на "*Вихід*";
- за допомогою мишки встановимо розміри форми та компонентів на ній приблизно такими, як це є на рис. 7.2.

Далі варто зберегти результати нашої роботи. Виконавши команди ➢**File** ➢ **Save Projects As…** отримаємо послідовно два діалогові вікна: **Save Unit1 As** (збережемо файл *Unit1.pas*) і **Save Project1 As** (збережемо файл *Project1.dpr*). На цьому перший етап побудови застосування завершено.

Зауваження. Опис усіх файлів проекту буде зроблено у наступному параграфі. У процесі проектування доцільно періодично зберігати результати роботи командами ➢**File** ➢ **Save All**.

Як бачимо, проект простий, а слів для опису етапу проектування інтерфейсу цього проекту сказано немало (ціла сторінка). Описуючи наступні проекти, будемо користуватися скороченою

формою запису, опускаючи стандартні кроки щодо відкриття нової форми та збереження результатів роботи. Скорочено етап проектування інтерфейсу нашого першого проекту можна описати так:

```
Form1: Label1; Label2; Edit1 //розміщуємо компоненти
       Button1; Button2
 Form1.Caption := 'Паліндроми'
 Label1.Caption:= 'Введіть рядок: '
 Label2.Caption:= 'Результат: '
 Button1.Caption:= 'Аналіз'
 Button2.Caption:= 'Вихід'
```

Другий етап – *програмування методів опрацювання подій*.

Для обох кнопок необхідно запрограмувати процедуру опрацювання події `OnClick`. Клацнувши два рази на кнопці `Button1` перейдемо на сторінку редактора коду, на якій побачимо програмну заготовку:

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

  end;
```

Оформимо текст програми 3.11 у вигляді процедури `Button1Click`:

```
procedure TForm1.Button1Click(Sender: TObject);
var s,s1:string;i,n:byte;
begin
 s1:='рядок '; Label2.Caption:='Результат: ';
 s:=Edit1.Text;
 while pos(' ',s)<>0 do delete(s,pos(' ',s),1);
 i:=1; n:=length(s);
 while (i<=n div 2) and (s[i]=s[n-i+1]) do inc(i);
 if i<=n div 2 then s1:=s1+'- не паліндром'
  else s1:=s1+'- паліндром';
 Label2.Caption:=label2.Caption+s1
end;
```

Процедура опрацювання події `OnClick` для кнопки `Button2`:

```
procedure TForm1.Button2Click(Sender: TObject);
begin close end;
```

Третій етап – *налагодження і запуск проекту на виконання*.

Натиснемо на кнопку **Run** на панелі швидкого доступу. Після декількох секунд, які витрачаються на компілювання та компонування проекту, отримаємо вікно застосування. Увівши декілька різних рядків (паліндромів і не паліндромів) впевнюємося, що завдання виконане.

7.3. Структура проекту віконного застосування

Програма на Delphi 6 складається з множини *модулів* (див. наступний розділ). Принцип модульності є дуже важливим для побудови надійних і відносно легко змінюваних застосувань. Будь-які зміни в окремому модулі не зачіпають решти модулів.

Усі компоненти розміщують на **формах**. Для кожної форми середовище Delphi 6 створює окремий модуль (файл з розширенням *.pas*) і саме у ньому програмуються процедури опрацювання подій. Крім того, можуть бути модулі, які не зв'язані з формами (містять функції, які мають загальне призначення).

Після етапів проектування інтерфейсу користувача і кодування необхідно налагодити і запустити програму на виконання, тобто реалізувати **третій етап** побудови проекту. Під *налагодженням* програми розумітимемо процес виявлення синтаксичних помилок і помилок часу виконання програми. Синтаксичні помилки виявляють під час компіляції програми, а інші помилки − після запуску програми на виконання (виявляють їх у ході *тестування програми*, а це своєрідна наука).

Процес запуску програми (вірніше, процес створення *виконавчого файлу*) здійснюється поетапно. Спочатку працює *компілятор*, який переводить текст модулів у машинний код (файли з розширенням .**dcu**). На завершення у роботу вступає *компонувальник*, який об'єднує усі файли .**dcu** у єдиний *виконавчий файл* з розширенням .**exe.** Як бачимо, у кінцевому підсумку проект – це множина різноманітних файлів (файли модулів, машинні файли модулів, виконавчий файл та інші).

Компіляцію і компонування *усіх модулів* проекту реалізують команди ➢**Project** ➢ **Build Project1.** Компіляцію і компонування *модулів* проекту, *тексти яких змінювалися* з моменту останньої компіляції, можна задати командами ➢**Project** ➢ **Compile Project1**. В

обох випадках після успішної компіляції створюються машинні файли модулів і відбувається компонування програми. Якщо і воно пройшло успішно, то створюється виконавчий файл.

Запуск програми на виконання реалізують командами ➢**Run** ➢**Run** або відповідною кнопкою на панелі швидкого доступу. Якщо до цього моменту компіляція і компонування не проводилися, то команда **Run,** незалежно від форми активізації (меню чи кнопка), попередньо виконає компіляцію і компонування програми.

У процесі побудови проекту Delphi 6 автоматично створює *головний модуль* (або *файл проекту*). Щоб побачити код головного модуля (стандартна назва **Project1.dpr**, див. рис. 7.3), треба виконати команди ➢**Project** ➢ **View Source**.

```
program Project1;
uses Forms, Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Рис. 7.3. Головний модуль проекту

Код головного модуля проекту змінювати не рекомендується[1]. Немає особливої потреби і у перегляді цього модуля, який формується автоматично. Однак для повноти розуміння процесів, які відбуваються у ході компонування програми, розглянемо головний модуль детальніше.

Назва програми збігається з назвою проекту, яка задається під час зберігання файлу проекту (головного модуля). Рекомендується надавати проектам осмислені назви (для наших невеликих навчальних проектів ми цього робити не будемо). Оператор `uses` підключає стандартний модуль роботи з формами `Forms` і модуль нашої форми `Unit1`. Директива `{$R *.res}` вказує компіляторові,

[1] Досвідчені програмісти, звісно, можуть певним чином змінювати код головного модуля. Наприклад, можна задавати умови створення допоміжних форм чи змінювати параметри настроювання середовища тощо.

що треба використати файл *ресурсів* (піктограми, курсори й інші зображення), назва якого збігається з назвою файлу проекту.

Перший виконавчий оператор тіла головного модуля

```
Application.Initialize();
```

ініціалізує компоненти нашого застосування. Наступний оператор

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

створює форму `Form1`. Останній оператор

```
Application.Run;
```

починає виконання програми. Після цього оператора програма чекає на настання відповідних подій, які й керують її ходом.

Крім головного модуля кожна програма включає (як мінімум) один *модуль форми* (файл зі стандартною назвою **Unit1.pas**), який містить опис стартової форми застосування та процедури опрацювання подій, а також інші необхідні підпрограми. Для перемикання між формою і вікном редактора коду, що містить відповідний модуль форми (див. рис. 7.4), треба виконати команди **➢View ➢Toggle Form/Unit** або натиснути відповідну кнопку на панелі швидкого доступу.

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes,
     Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    Label2: TLabel;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var Form1: TForm1;
```

```
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
var s,s1:string;i,n:byte;
begin
 s1:='рядок ';
 // Див. на стор. 142
end;
procedure TForm1.Button2Click(Sender: TObject);
begin close end;
```

Рис. 7.4. Файл модуля форми

У розділі інтерфейсу модуля Delphi 6 підключає стандартні модулі, а також формує опис типу (*класу*) нашої форми. Детально про класи та об'єкти буде сказано у десятому розділі. Далі у *файлі модуля* форми розміщено оголошення покажчика на об'єкт `Form1`, а потім реалізовані процедури опрацювання подій для цієї форми.

Директива `{$R *.dfm}` вказує компіляторові на те, що форма використовує текстовий файл форми (розширення **.dfm** з) тією ж назвою, що і назва цього файлу (тобто файл **Unit1.dfm**). Текстовий файл форми зберігає інформацію про властивості самої форми та компонентів, які на ній розміщені. Для того, щоб у середовищі Delphi 6 побачити текстовий файл форми, необхідно виконати команди:

- **ПКМ** /*над формою – отримаємо контекстне меню*/
- ➢ **View As Text**

Щоб цей файл знову побачити у графічному вигляді форми, необхідно виконати такі команди:

- **ПКМ** /*над текстовим файлом форми*/
- ➢ **View As Form**

Висновки. Прикладна програма (застосування) в інтегрованому середовищі Delphi 6 називається *проектом*. Проект – це множина різноманітних файлів. Розширення файлу визначає його зміст.

Основні файли проекту:

Головний модуль проекту (`Project1.dpr`)	Файл головного модуля, який ініціалізує і запускає застосування на виконання
Файл параметрів проекту (`Project1.dof`)	Текстовий файл, що містить значення параметрів проекту, які впливають на процеси компілювання та компонування
Файл ресурсів проекту (`Project1.res`)	Двійковий файл, що містить ресурси проекту: піктограми, курсори й інші зображення
Файл конфігурації системи Delphi 6 (`Project1.cfg`)	Текстовий файл зберігає інформацію про останній сеанс роботи з проектом: відкриті вікна, їхній розмір і положення тощо
`Project1.exe`	Виконавчий файл застосування
Файл модуля форми (`Unit1.pas`)	Кожній формі відповідає текстовий файл модуля, який зберігає процедури опрацювання подій форми та реалізацію інших підпрограм
Файл форми (`Unit1.dfm`)	Текстовий файл зберігає інформацію про властивості самої форми та властивості її компонентів
Файли резервних копій (`.~dfm`, `.~ ddp`, `.~ pas`)	За допомогою резервних копій можна повернутися до попередніх версій відповідних файлів
Файли компіляції `Unit1.dcu`, `Unit1.ddp`	Файли, які створює компілятор

Зі всіх перелічених файлів найважливішими є файли **.dpr**, **.res**, **`.pas`** і **.dfm**. Це ті файли, які треба переносити на інший комп'ютер, якщо там треба демонструвати чи завершувати проект. Решту файлів можна витирати з диска. Назви `Project1` і `Unit1` при першому збереженні проекту можна поміняти на більш осмислені.

Зауважимо, що виконавчий файл (файл `Project1.exe`) буде працювати на іншому комп'ютері, якщо там є необхідні динамічні бібліотеки, що входять до складу Delphi 6 (наприклад, бібліотека `vcl40.bpl` обсягом 1 888 4136 байтів).

Для повторного запуску інтегрованого середовища Delphi 6 з одночасним викликом необхідних частин проекту достатньо два рази клацнути мишею над файлом головного модуля проекту.

На завершення звернемо увагу на такий факт: між змістом вікна форми і вікном редактора коду існує неперервний зв'язок, який строго контролюється системою Delphi 6. Наприклад, розміщення на формі нового компонента змінює код програми. Автоматично створюються заготовки процедур опрацювання подій. Програміст заповнює ці заготовки конкретними операторами, але водночас він **не повинен видаляти з тексту ті рядки, які туди вставила система Delphi 6**.

• ? Запитання для самоперевірки

1. Перерахуйте основні вікна середовища Delphi 6.
2. З чого складається головне вікно?
3. Для чого використовується вікно інспектора об'єктів?
4. Для чого використовується вікно форми?
5. Для чого використовується вікно редактора коду?
6. Що таке панель швидкого доступу?
7. Що таке палітра компонентів?
8. Перерахуйте основні етапи побудови проекту віконного застосування.
9. Коротко охарактеризуйте етап візуального програмування.
10. Коротко охарактеризуйте етап кодування.
11. Для чого у проекті використовують позначки?
12. Для чого у проекті використовують поля введення?
13. Для чого у проекті використовують командні кнопки?
14. Коротко охарактеризуйте призначення головного модуля проекту.
15. Коротко охарактеризуйте призначення файлу модуля форми.
16. Коротко охарактеризуйте призначення текстового файлу форми.
17. Що містить файл параметрів проекту?
18. Що містить файл ресурсів проекту?
19. Що містить файл конфігурації системи?

• Завдання для програмування

Завдання 7.1. Створити віконне застосування для виконання *завдання 3.1* (стор. 53).

Завдання 7.2. Створити віконне застосування для виконання *завдання 3.2* (стор. 54).

8. МОДУЛІ

План викладу матеріалу

1. Означення модуля і його застосування у Паскалі
2. Модулі в Object Pascal
3. Приклад модуля Object Pascal (виділення підрядків)
4. Модуль глобальних оголошень

Ключові терміни розділу

- *Означення модуля*
- *Підключення модулів у проект*
- *Функція* `GetSubStr`
- *Виняткові ситуації*
- *Опрацювання винятків*
- *Структура модуля мови Паскаль*
- *Структура модуля Object Pascal*
- *Функція* `GetLine`
- *Захищені блоки*
- *Модуль глобальних оголошень*

8.1. Означення модуля і його застосування у Паскалі

Модуль (unit) у мові Паскаль – це особливим чином оформлена бібліотека підпрограм (процедур і функцій). Модуль не може бути запущений на виконання самостійно, він може тільки брати участь у побудові програм та інших модулів. Модуль – це окрема програмна одиниця, що незалежно компілюється і зберігається.

Отже, **модуль** – це сукупність програмних елементів (константи, типи, змінні, підпрограми), визначених для використання іншими програмами. Елементи модуля розбивають на три частини:

- **інтерфейсна частина** (або просто **інтерфейс**, *interface*) містить програмні елементи, визначені для використання іншими програмами або модулями. Ці елементи є видимими поза модулем;
- **реалізація** (*implementation*) містить програмні елементи, необхідні тільки для роботи самого модуля. Ці елементи є невидимими (або прихованими);
- **частина ініціалізації** модуля містить початкові значення деяких елементів модуля.

Модуль має таку загальну структуру:

```
unit назва_модуля; {заголовок модуля}
interface
{ опис видимих програмних елементів модуля }
```

```
implementation
{ опис прихованих програмних елементів модуля }
begin
{ оператори ініціалізації елементів модуля }
end.
```

У окремих випадках модуль може не містити частин реалізації та ініціалізації, тоді структура модуля буде такою:

```
unit назва_модуля; {заголовок модуля}
interface
{ опис видимих програмних елементів модуля }
implementation
end.
```

Використання у модулях процедур і функцій має свої особливості. Заголовок підпрограми містить всі відомості, необхідні для її виклику: назву, перелік і тип параметрів, тип результату функції, ця інформація має бути доступна для інших програм і модулів. З іншого боку, текст підпрограми, що реалізовує її алгоритм, іншими програмами і модулями не може бути використаний.

Тому заголовок процедур і функцій вміщують в інтерфейсну частину модуля, а текст – у частину реалізації. Інтерфейсна частина модуля містить тільки видимі (доступні для інших програм і модулів) заголовки процедур і функцій. Повний текст процедури або функції вміщують у частину реалізації, причому у заголовку може не бути списку формальних параметрів.

Назву модуля беруть із заголовка модуля. Крім того, назва файлу, в якому зберігається модуль, має збігатися з назвою модуля. Для підключення модуля до програми необхідно вказати його назву в розділі опису модулів (розділ **uses**), наприклад:

```
uses CRT, Graph, myUnit;
```

У випадку, коли назви змінних в інтерфейсній частині модуля і в програмі, що використовує цей модуль, збігаються, звернення буде відбуватися до змінної, описаної в програмі. Для звернення до змінної, описаної в модулі, необхідно застосувати складну назву, до якої входить назва модуля і назва змінної, розділені крапкою.

Рекурсивне використання модулів заборонене. Якщо в модулі є розділ ініціалізації, то оператори з цього розділу будуть виконані перед початком виконання програми, в якій використовується цей модуль.

Приклад 8.1. Побудувати модуль, який включає опис комплексного типу даних `Complex` (числа вигляду $m+n\cdot i$, де m і n – дійсні числа, а i – уявна одиниця) і функції/процедури операцій над комплексними числами: додавання, віднімання, множення, ділення; присвоєння початкових значень через введення даних з клавіатури; виведення на екран комплексного числа. Перевірити роботу модуля на прикладі обчислення виразу $d=\dfrac{a+b-(2+3\cdot i)\cdot c}{a+2\cdot b}$, де a, b і c – комплексні числа.

```
program Project_81;
{$APPTYPE CONSOLE}
uses SysUtils, Cmplx in 'Cmplx.pas';
var a,b,c,d,e: Complex;
begin
 writeln('Enter coplex numbers:');
 readc(a);  readc(b);  readc(c);
 e.re:=2;  e.im:=3;  d:=AddC(a,b);
 d:=DivC(SubC(d,MulC(e, c)), AddC(d,b));
 writec(d);  readln;
end.
```

Файл модуля:

```
//-------------------------------------
unit Cmplx; // Збережено у файлі Cmplx.pas
//-------------------------------------
interface
//-------------------------------------
 Type Complex=record re,im:real end;
 Function AddC(x,y:Complex):Complex;
 Function SubC(x,y:Complex):Complex;
 Function MulC(x,y:Complex):Complex;
 Function DivC(x,y:Complex):Complex;
```

```
  Procedure ReadC(var x:Complex);
  Procedure WriteC(x:Complex);
 //-------------------------------------
 implementation
 //---------------------------------------
   Function AddC(x,y:Complex):Complex;
   begin   AddC.re:=x.re+y.re;
           AddC.im:=x.im+y.im;
   end;
   Function SubC(x,y:Complex):Complex;
   begin    SubC.re:=x.re-y.re;
            SubC.im:=x.im-y.im;
   end;
   Function MulC(x,y:Complex):Complex;
   begin    MulC.re:=x.re*y.re-x.im*y.im;
            MulC.im:=x.re*y.im+x.im*y.re;
   end;
   Function DivC(x,y:Complex):Complex;
   var z:real;
   begin z:=sqr(y.re)+sqr(y.im);
    if z=0 then
      begin writeln('Divide by 0');
            readln; halt end;
    DivC.re:=(x.re*y.re+x.im*y.im)/z;
    DivC.im:=(x.re*y.im-x.im*y.re)/z;
   end;
   Procedure ReadC(var x:Complex);
   begin  write('Enter Re: '); read(x.re);
          write('Enter Im: '); read(x.im);
   end;
   Procedure WriteC(x:Complex);
   begin  write('Result=', x.re:8:2);
          if x.im>0 then write('+');
          write(x.im:8:2,' I'); readln;
   end;
 end.
```

Приклад 8.2. Побудувати модуль, який включає опис раціонального типу даних `Rat` (числа вигляду $\frac{m}{n}$, де m і n – цілі числа, $n \neq 0$) і функції/процедури операцій над раціональними числами: додавання, віднімання, множення, ділення; присвоєння початкових значень через уведення даних з клавіатури; виведення на екран раціонального числа; спрощення раціонального числа (скорочення спільних множників у чисельнику та знаменнику дробу; присвоєння чисельнику знака числа). Перевірити роботу модуля на прикладі обчислення виразу $d = \frac{a + b - \frac{2}{3} \cdot c}{a + 2 \cdot b}$, де a, b і c – раціональні числа, знаменники яких не дорівнюють нулю.

Зауваження. Контроль за відсутністю нулів у знаменниках дробів, значення яких уводяться з клавіатури, здійснюється у головній програмі `Project_82`. Однак при виконанні операцій додавання/віднімання у чисельнику дробу може бути нуль. Якщо дріб з нулем у чисельнику братиме участь, як дільник, в операції ділення, то виникне помилка виконання програми. Для виявлення такої ситуації у функції `DivR` відбувається відповідна перевірка з виведенням повідомлення і завершенням роботи програми.

```
program Project_82;
{$APPTYPE CONSOLE}
uses SysUtils, Ratio in 'Ratio.pas';
var a,b,c,d,e:Rat;
begin
  writeln('Enter 6 integer numbers<>0:');
  readR(a); readR(b); readR(c);
  if (a.zn=0) or (b.zn=0) or (c.zn=0)
   then
    begin
         writeln('We have 0! Enter number');
         read(a.zn);  halt
    end;
  e.ch:=2; e.zn:=3;
  d:=AddR(a,b);
```

```
  d:= DivR(SubR(d,mulR(e,c)), AddR(d,b));
  WriteR(d); // Виводимо не спрощений дріб
  Simpl(d);
  WriteR(d); // Виводимо спрощений дріб
  readln;
end.
```

Файл модуля:

```
unit Ratio; // Збережено у файлі Ratio.pas
//-------------------------------------
interface
//-------------------------------------
 Type Rat=record ch,zn:integer end;
 Function AddR(x,y:Rat):Rat;
 Function SubR(x,y:Rat):Rat;
 Function MulR(x,y:Rat):Rat;
 Function DivR(x,y:Rat):Rat;
 Procedure ReadR(var x:Rat);
 Procedure WriteR(x:Rat);
 Procedure Simpl(var x:Rat);
//-------------------------------------
implementation
//---------------------------------------
  Function AddR(x,y:Rat):Rat;
  begin AddR.ch:=x.ch*y.zn+y.ch*x.zn;
        AddR.zn:=x.zn*y.zn;      end;
  Function SubR(x,y:Rat):Rat;
  begin SubR.ch:=x.ch*y.zn-y.ch*x.zn;
        SubR.zn:=x.zn*y.zn;      end;
  Function MulR(x,y:Rat):Rat;
  begin MulR.ch:=x.ch*y.ch;
        MulR.zn:=x.zn*y.zn;      end;
  Function DivR(x,y:Rat):Rat;
  var z:integer;
  begin DivR.ch:=x.ch*y.zn; z:=x.zn*y.ch;
   if z=0 then
    begin writeln('Divide by 0! Enter number');
```

```
            read(z); halt
      end;
      DivR.zn:=z;
    end;
   Procedure ReadR(var x:Rat);
   begin  write('Enter ch: '); read(x.ch);
          write('Enter zn<>0: '); read(x.zn);
   end;
   Procedure WriteR(x:Rat);
   begin write('Result='); write(x.ch:8);
     write('/ ('); writeln(x.zn:6,' )'); readln;
   end;
   Procedure Simpl(var x:Rat);
   var i,n:integer;
   begin
    if abs(x.ch)<abs(x.zn)
      then n:=abs(x.ch) else n:=abs(x.zn);
    for i:=n downto 2 do
     if (x.ch mod i=0)and(x.zn mod i=0) then
      begin x.ch:=x.ch div i; x.zn:=x.zn div i end;
    if x.ch*x.zn>0 then
      begin x.ch:=abs(x.ch); x.zn:=abs(x.zn) end
    else
      begin x.ch:=-abs(x.ch); x.zn:=abs(x.zn) end
   end;
 end.
```

8.2. Модулі в Object Pascal

З виходом пакетів **Delphi** змінилася технологія створення прикладних застосувань, унаслідок чого суттєво зросло значення мовної конструкції “**модуль**” (**`Unit`**), у синтаксисі якої були зроблені деякі зміни. Все це було відображено так:

- у середовищі Delphi кожній формі відповідає свій модуль і невізуальні алгоритмічні дії, також, як правило, оформляють у вигляді окремих модулів. Отже, середовище Delphi має вбудовану підтримку концепції модульного програмування мови Object Pascal;
- у Object Pascal введено новий розділ модуля – розділ завер-

шення (*finalization*), якого не було у Borland (Turbo) Pascal. Цей розділ використовується переважно для звільнення ресурсів, виділених додатку в розділі ініціалізації, і тим самим він гарантує коректне “чисте” завершення додатка;

- у Object Pascal для позначення розділу ініціалізації модуля використовується зарезервоване слово *initialization*, на відміну від версії Borland (Turbo) Pascal, в якій використовувалося слово *begin*. У принципі, використання слова *begin* також допускається (для сумісності з попередніми версіями);
- машинні файли модулів у Object Pascal мають розширення **.dcu** (у Borland (Turbo) Pascal ці файли мають розширення **.tpu**).

Загалом модуль Object Pascal має таку структуру:

```
unit назва_модуля; {заголовок модуля}
interface
{ опис видимих програмних елементів модуля }
implementation
{ опис прихованих програмних елементів модуля }
initialization
{ оператори ініціалізації елементів модуля }
finalization
{ оператори, які виконуються при закритті модуля}
end.
```

Якщо у модулі немає потреби застосовувати оператори ініціалізації, то ключове слово `initialization` можна не вказувати. Розділ завершення є необов’язковим і використовується лише разом з розділом `initialization`. Розділи модулів `finalization` виконують у порядку, протилежному до порядку виконання розділів `initialization` цих модулів.

8.3. Приклад модуля Object Pascal (*виділення підрядків*)

8.3.1. Постановка завдання

Досить типовою є ситуація отримання від користувача чи з файлу під час роботи застосування значень елементів масивів (найчастіше – це вектори чи матриці). На формі ці значення найзручніше розміщувати у вигляді окремого **рядка** (для вектора) чи **табли-**

ці (для матриці) – *послідовності рядків*. У рядку окремі числа відокремлюють пропусками. Для моделювання вектора використовуватимемо компонент **`Edit1`**, а матриці – компонент **`Memo1`**. Тоді вектор, як **окремий рядок символів** (в цілому), буде значенням властивості **`Edit1.Text`**, а матриця, як окремий рядок символів, буде значенням властивості **`Memo1.Text`**. Ознакою закінчення рядка у компонентові **`Memo1.Text`** є символ `#13` (за ним у парі завжди йде символ `#10`).

Для отримання значення елемента вектора за його індексом (нехай *i*) необхідно мати функцію (назвемо її `GetSubStr`), яка б давала змогу отримати з рядка символів *i*-тий **підрядок** (ознака закінчення підрядка – символ пропуску), який відповідає потрібному елементу. Далі за допомогою відповідних функцій здійснюється перетворення цього **підрядка символів** до типу елементів вектора.

Для отримання значення елемента матриці за його індексами (нехай *i* – індекс рядка, а *j* – індекс стовпця) необхідно мати функцію (назвемо її `GetLine`), яка б давала можливість отримати з рядка символів *i*-тий **підрядок** (ознака закінчення підрядка – символ `#13`), який відповідає *i*-тому **рядку матриці**. З цього підрядка за допомогою функції `GetSubStr` отримаємо *j*-тий **підрядок**, який відповідає потрібному елементу матриці. За допомогою відповідних функцій відбувається перетворення **підрядка символів** до типу елементів матриці.

Для того, щоб з отриманого рядка виділити *i*-тий підрядок, функція `GetSubStr` спочатку видаляє попередні *i*-1 підрядки; потім знаходить пропуск, який позначає *i*-тий підрядок (фактично це **перший підрядок** у рядку-залишку); виділяє цей підрядок і повертає його як значення функції.

Функція `GetSubStr` при такому підході даватиме **правильні результати**, якщо перед першим підрядком не стоятимуть пропуски, а між підрядками стоятиме тільки один пропуск. На практиці цього досягнути важко, адже користувач не повинен рахувати пропуски!

Для забезпечення універсальності функція `GetSubStr` автоматично знищує пропуски на початку рядка, а також на початку рядка-залишку після видалення чергового підрядка.

Якщо індекс заданого підрядка перевищує кількість усіх підрядків рядка-арґументу, то функція `GetSubStr` повертає значення **порожного рядка**. Це можна використати у викликаючому програмному коді для виведення відповідного повідомлення.

Функція `GetLine` працює аналогічно, тільки для пошуку підрядка використовується символ `#13`, а при копіюванні враховується наявність пари символів `#13` і `#10`.

Будемо вважати, що функція `GetLine` може використовуватися і для отримання значень вектора-стовпця (з **`Memo1.Text`**). Тому передбачимо можливість автоматичного знищення пропусків на початку рядка, а також на початку рядка-залишку після видалення чергового підрядка.

8.3.2. Реалізація модуля виділення підрядків

Наведемо модуль `CStrToArray`, що містить усі ці функції:

```
unit CStrToArray;
//-------------------------------------
interface
//-------------------------------------
Function GetSubStr(st:string; n:integer):string;
// Виділяє з рядка st підрядок з вказаним номером n.
// Підрядок - послідовність символів між пропусками.
// Результат - підрядок або '' (порожний рядок).
//-------------------------------------
Function GetLine(st:string; n:integer):string;
// Функція повертає підрядок з вказаним номером n.
// Підрядок – послідовність символів з #13 на кінці.
// Результат - підрядок або ''
//-------------------------------------
implementation
//-------------------------------------
Function GetSubStr(st:string; n:integer):string;
var p:integer;
```

```
begin
 // Видалення початкових пропусків у st
 while (pos(' ',st)=1) do delete(st,1,1);
 if n>1 then
  repeat p:=pos(' ',st);
   if p <> 0 then
    begin
     // Видалення підрядка через часткове копіювання
     st:=copy(st,p+1,Length(st)-p); //
     // Видалення початкових пропусків у частині
     // рядка, що залишається
     while (pos(' ',st)=1) do delete(st,1,1);
     n:=n-1;
    end;
  until (n=1) or (p=0);
 // Формування результату
 if n>1
  then result:=''
  else
   begin p:=pos(' ',st);
      if p <> 0
        then result:=copy(st,1,p-1)
        else result:=st;
   end
end;
//-------------------------------------
Function GetLine(st:string; n:integer):string;
var p:integer;
begin  // Видалення початкових пропусків у st
          while (pos(' ',st)=1)do delete(st,1,1);
 if n>1 then
  repeat p:=pos(#13,st);
   if p <> 0 then
    begin st:=copy(st,p+2,Length(st)-p);
     // Роздільник - 2 символи з кодом 13 і 10
  // Видалення початкових пропусків у рядку-залишку
        while (pos(' ',st)=1) do delete(st,1,1);
     n:=n-1;
```

```
    end;
  until (n=1)or(p=0);
 // Формування результату
  if n>1
   then result:=''
   else
    begin p:=pos(#13,st);
      if p <> 0 then result:=copy(st,1,p-1)
                else result:=st;
    end;
end;
end.
```

8.3.3.Тестування модуля виділення підрядків

Перевіримо роботу модуля на прикладі знаходження добутку вектора *p* з п'яти елементів на матрицю *q* розміром 5×3 (тобто *p*×*q*). Значення вектора *p* отримуватимемо з властивості **Edit1.Text**, а матриці – з властивості **Memo1.Text**. Форма для такого множення зображена на рис. 8.1. Результати множення отримуємо у стандартному вікні функції `ShowMessage` (вікно **Project1** на рис. 8.1). Хоча результат (для **спрощення** програмного коду) оформлено у вигляді стовпця, але насправді цей результат є вектором-рядком.

Етап проектування інтерфейсу користувача (запишемо у стилі вмістимого файлу .**dfm**):

```
object Form1: TForm1
 object Label1: TLabel ⇒ Caption = 'Введіть вектор'
 object Label2: TLabel ⇒ Caption = 'Введіть матрицю'
 object Edit1: TEdit ⇒ Text = ''{порожне значення}
 object Memo1: TMemo ⇒ Lines.Strings = ''
                            {порожне значення}
 object Button1: TButton ⇒ Caption = 'Обчислити добуток'
                          OnClick = Button1Click
 object Button2: TButton ⇒ Caption = 'Ввести вектор'
                           OnClick = Button2Click
 object Button3: TButton ⇒ Caption = 'Ввести матрицю'
                          OnClick = Button3Click
```

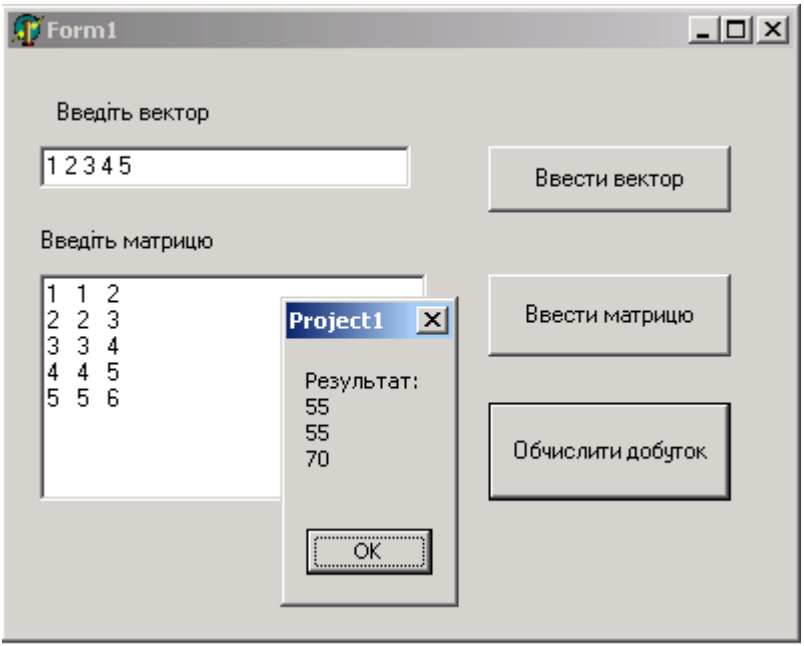


Рис. 8.1. Форма для множення вектора на матрицю

Вмістиме **модуля форми:**

```
unit Unit1;
//-------------------------------------
interface
//-------------------------------------
uses Windows, Messages, SysUtils, Variants, Classes,
Graphics, Controls, Forms, Dialogs, StdCtrls;
//-------------------------------------
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Memo1: TMemo;
    Label1: TLabel;
    Label2: TLabel;
    Label4: TLabel;
```

```
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var Form1: TForm1;
//-------------------------------------
implementation
//-------------------------------------
uses CStrToArray; // підключення модуля
const SIZER=5; //Розмір вектора, кількість рядків матриці
      SIZEC=3; //Кількість стовпців матриці
var   p:array[1..SIZER] of real;
   q:array[1..SIZER,1..SIZEC] of real;
{$R *.dfm}
//-------------------------------------
procedure TForm1.Button1Click(Sender: TObject);
var i,j:integer; st:string; a:real;
begin
  st:='Результат:'+#13;
  for j:=1 to SIZEC do
   begin a:=0;
     for i:=1 to SIZER do a:=a+p[i]*q[i,j];
     st:=st+FloatToStr(a)+#13;
   end;
  ShowMessage(st);
end;
//-------------------------------------
```

```
// Введення елементів вектора
procedure TForm1.Button2Click(Sender: TObject);
var i:integer; buf:string;
begin
 i:=1;
 repeat buf:=GetSubstr(Edit1.text,i);
  if Length(buf) <> 0
  then try p[i]:=StrToFloat(buf)
        except on EConvertError do
          begin ShowMessage('Треба ввести число');
                   Edit1.SetFocus; Exit; end
        end
  else begin ShowMessage('Треба ввести'+IntToStr(SIZER)+
             'чисел.'); Edit1.SetFocus; Exit; end;
  i:=i+1
 until (i>SIZER) or (Length(buf)=0);
 buf:='Вектор:'+#13;
 for i:=1 to SIZER do buf:=buf+FloatToStr(p[i])+#13;
 ShowMessage(buf);
end;
//-------------------------------------
// Введення елементів матриці
procedure TForm1.Button3Click(Sender: TObject);
var i,j:integer; buf, st:string;
begin
 i:=1;
 repeat st:=Getline(Memo1.text,i);
  if Length(st) <> 0
  then
   begin j:=1;
    repeat buf:=GetSubstr(st,j);
     if Length(buf) <> 0
     then try q[i,j]:=StrToFloat(buf)
            except on EConvertError do
             begin ShowMessage('Треба ввести число');
                   Memo1.SetFocus; Exit; end
             end
     else begin ShowMessage('Треба ввести'
```

```
                  +IntToStr(SIZEC)+' чисел.');
                  Memo1.SetFocus; Exit; end;
     j:=j+1
    until (j>SIZEC) or (Length(buf)=0)
   end
  else begin ShowMessage('Треба ввести'+IntToStr(SIZER)
         +' рядків.'); Memo1.SetFocus; Exit;
       end;
  i:=i+1
 until (i>SIZER) or (Length(st)=0);
end;
end.
```

8.3.4. Опрацювання виняткових ситуацій

Як уже зазначалося, під час виконання функцій перетворення рядків у числа при виникненні помилок генерується **виключення** (або виняткова ситуація) `EConvertError`.

У *Object Pascal* для опрацювання виняткових ситуацій передбачений механізм захищеного блоку, який починається зарезервованим словом **`try`** (зробити спробу виконати) і закінчується зарезервованим словом **`end`**. Є два типи захищених блоків – **`except`** (вилучити) і **`finally`** (на закінчення), які відрізняються способом опрацювання виняткових ситуацій. Розглянемо тип **`except`**, який використовується у програмі тестування.

Порядок виконання операторів у блоці **`except`** такий: спочатку виконуються оператори секції **`try`**…**`except`**; якщо після виконання операторів виняткової ситуації не виникло, то робота захищеного блоку на цьому припиняється й управління переходить до оператора, який стоїть після **`end`**; якщо при виконанні частини **`try`** виникла виняткова ситуація, то керування отримує відповідний оператор опрацювання винятку в секції **`except`**.

Оператор опрацювання винятку в блоці **`except`** має такий синтаксис:

```
on клас_виняткової_ситуації do оператор;
```

У програмі тестування класом_виняткової_ситуації виступає клас EConvertError, а оператором є блок, у якому:

- виводиться повідомлення про помилку за допомогою стандартної функції ShowMessage;
- фокус уведення повертається на поле, в якому виникла помилка;
- реалізується вихід з процедури опрацювання події.

8.4. Модуль глобальних оголошень

При роботі з модулями треба мати на увазі, що традиційні правила області дії глобальних і локальних змінних для модулів не мають сили. Конструкція **unit** була спеціально розроблена так, щоб виключити вплив глобальних змінних головної програми на внутрішні оголошення модуля.

Якщо ж все-таки виникає потреба ввести деякі оголошення (типів, змінних тощо), які були б доступними для всіх блоків програми (тобто глобальні оголошення), то для модулів це можна зробити єдиним чином – створити спеціальний **модуль глобальних оголошень** і включити його у список імпорту (оператор uses) усіх модулів, які потребують цих оголошень.

Наведемо приклад модуля роботи зі стеком UStack, який містить процедури занесення елемента у стек (Push), процедуру отримання елемента зі стека (Pop) і виведення значень елементів стека у рядок символів (GetStack). Елементи стека мають тип Char. Однак доцільно так запрограмувати модуль UStack, щоб зі зміною типу елементів стека (наприклад, на **integer**) мінімізувати кількість виправлень. Для цього створимо модуль глобальних оголошень UGlob, у якому введемо глобальний тип TVal.

Модуль глобальних оголошень:

```
unit UGlob;
interface
 type TVal=Char;
implementation
end.
```

Модуль роботи зі стеком:

```
unit UStack;
//-------------------------------------
interface
//-------------------------------------
 uses UGlob;
 procedure Push(Val:TVal);
 procedure Pop(var Val:TVal);
 function GetStack:string;
//-------------------------------------
implementation
//-------------------------------------
 type
  TPtr=^TElem; // Тип покажчика на елемент стека
  TElem=record // Тип елемента стека
          Inf:TVal; Link: TPtr  end;
 Var Top:TPtr; // Покажчик на вершину стека
     Val: TVal; // Домоміжна змінна для очищення стека
//-------------------------------------
 procedure Push(Val:TVal);
 var p:TPtr;
 begin
   New(p); p^.Inf:=Val;
   p.Link:=Top; Top:=p
 end;
//-------------------------------------
 procedure Pop(var Val:TVal);
  var p:TPtr;
 begin
  if Top<>nil then
   begin Val:=Top^.Inf; p:=Top;
         Top:=p^.Link; Dispose(p)
   end
 end;
//-------------------------------------
 function GetStack:string;
  var p:TPtr; s:string;
```

```
 begin p:=Top;  s:='';
  While p<>nil do begin s:=s+p^.inf; p:=p^.Link; end;
  GetStack:=s
 end;
//-------------------------------------
initialization
 Top:=nil; // Початкове значення вершини стека
finalization
 while Top<>nil do Pop(Val); // Очищення стека
end.
```

Перевіримо роботу модуля на прикладі **перевертання слів**. Як відомо, цю задачу найліпше запрограмувати за допомогою стека. Форма такого перевертання слів наведена на рисунку 8.2. Крім того реалізуємо задачу видалення з уведеного слова останнього символу (першого символу у стеку).

Етап проектування інтерфейсу користувача:

```
object Form1: TForm1  ⇒
                      Caption = 'Перевертання слів'
 object Label1: TLabel  ⇒
                      Caption = 'Введіть слово'
 object Label2: TLabel  ⇒
                       Caption = 'Перевернуте слово'
 object Edit1: TEdit ⇒ Text= ''  {порожне значення}
 object Edit2: TEdit ⇒ Text= '' {порожне значення}
               ReadOnly = True
 object Button1: TButton  ⇒  OnClick = Button1Click
        Caption = 'Отримати перевернуте слово'
 object Button2: TButton  ⇒  OnClick = Button2Click
        Caption  = 'Видалити останній символ'
```

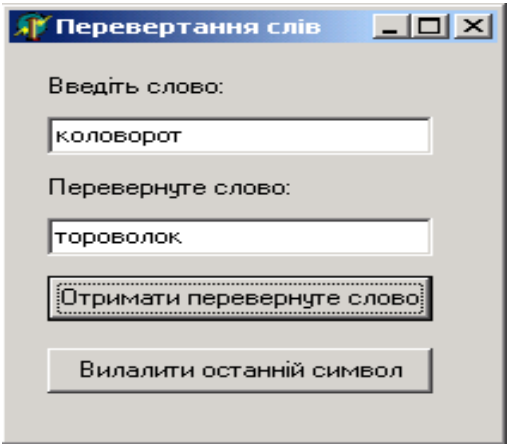


Рис.8.2. Перевертання слів

Вмістиме **модуля форми:**

```
unit Unit1;
//-------------------------------------
interface
//-------------------------------------
 uses
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
//-------------------------------------
 type
  TForm1 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    Label2: TLabel;
    Edit2: TEdit;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
  end;
//-------------------------------------
 var  Form1: TForm1;
//-------------------------------------
implementation
//-------------------------------------
 uses UGlob, UStack; // підключення власних модулів
//-------------------------------------
{$R *.dfm}
//-------------------------------------
 procedure TForm1.Button1Click(Sender: TObject);
 var
     s:String; i,n:byte; c:TVal;
 begin
  s:=Edit1.Text; n:=Length(s);
  for i:=1 to n do
   begin c:=s[i]; Push(c) end;
  if Edit2.ReadOnly Then Edit2.ReadOnly:=false;
  Edit2.Text:=getStack;
  Edit2.ReadOnly:=true;
 end;
//-------------------------------------
 procedure TForm1.Button2Click(Sender: TObject);
 var
     s:String; i,n:byte; c:TVal;
 begin
  s:=Edit1.Text;  n:=Length(s);
  s:=Copy(s,1,n-1);
  Edit1.Text:=s;  Pop(c);
  if Edit2.ReadOnly Then    Edit2.ReadOnly:=false;
  Edit2.Text:=getStack;
  Edit2.ReadOnly:=true;
 end;
end.
```

? Запитання для самоперевірки

1. Що таке модуль?
2. Яка структура модуля у мові Паскаль?
3. Для чого використовується інтерфейсна частина модуля?
4. Для чого використовується частина реалізації модуля?
5. Для чого використовується частина ініціалізації модуля?
6. Які частини модуля можна не вживати?
7. Коли у модулі треба вживати складну назву змінної?
8. Вкажіть на особливості застосування підпрограм у модулях.
9. Чи допускається рекурсивне використання модулів?
10. Коротко охарактеризуйте нововведення у модулях мови Object Pascal.
11. Яка структура модуля у мові Object Pascal?
12. Для чого використовується розділ `finalization`?
13. Для чого використовується розділ `initialization`?
14. Як підключити власний модуль до програми/проекту?
15. Коротко опишіть алгоритм виділення підрядка з рядка.
16. Що таке виняткова ситуація?
17. Коли виникає виняткова ситуація `EConvertError`?
18. Що таке захищений блок?
19. Які є типи захищених блоків?
20. Який порядок виконання операторів у блоці типу `except`?
21. Який вигляд має оператор опрацювання виняткової ситуації?
22. Як виконується оператор опрацювання виняткової ситуації?
23. Для чого використовується модуль глобальних оголошень?

Завдання для програмування

Завдання 8.1. Створити віконне застосування для виконання *завдання 4.1* (стор. 86). Для виділення окремих елементів вектора у рядку поля введення скористатися можливостями, які надає модуль `CStrToArray` (стор. 158).

Завдання 8.2. Створити віконне застосування для виконання *завдання 4.2* (стор. 87). Для виділення окремих елементів вектора у рядку поля введення скористатися можливостями, які надає модуль `CStrToArray` (стор. 158).

Завдання 8.3. Створити віконне застосування для виконання *завдання 4.3* (стор. 88). Для виділення окремих елементів матриці у багаторядковому текстовому редакторі скористатися можливостями, які надає модуль `CStrToArray` (стор. 158).

Завдання 8.4. У режимі консолі створити і протестувати заданий модуль.[1]

1. Модуль, який включає опис типу вектора цілих чисел та процедури знаходження: найменшого елемента; добутку значень усіх елементів вектора; середнього арифметичного значення серед ненульових елементів.
2. Модуль, який включає опис типу вектора цілих чисел та процедури знаходження: найбільшого елемента; суми значень усіх елементів вектора; середнього арифметичного значення серед додатних елементів.
3. Модуль, який включає опис типу вектора дійсних чисел та процедури: заміни від'ємних значень вектора на нульові; зменшення додатних значень на задану величину; заміни знаків усіх елементів на протилежні.
4. Модуль, який включає опис двох типів векторів цілих чисел з різною кількістю елементів та процедури: об'єднання двох заданих векторів у третій сумарної довжини; додавання векторів; множення векторів.
5. Модуль, який включає опис типу "*день_тижня*" (*пн*, *вт*, ...), типу вектора з 365 елементів (дні невисокосного року) з елементами типу "*день_тижня*" та процедури: заповнення вектора назвами дня тижня; визначення найпершого понеділка року; обчислення кількості певних днів тижня (*понеділків*, *вівторків* і т.д.) у році.
6. Модуль, який включає опис типу вектора, елементами якого є символи та процедури: обчислення кількості входжень заданого символа у вектор; заміни заданого символа на інший символ у всьому векторі; перестановки місцями двох заданих символів, якщо вони стоять поряд.
7. Модуль, який включає опис типу вектора, елементами якого є символи та процедури: утворення вектора з окремих символів, що читаються з клавіатури, доповнення новими символами (без пропусків) у зворотному порядку у тому самому векторі; друкування такого вектора.
8. Модуль, який включає опис типу вектора дійсних чисел та процедури: формування (читання) елементів вектора з клавіатури; друкування значень вектора по п'ять у рядок; перевірити, чи утворюють елементи вектора зростаючу послідовність.

---

[1] Використано індивідуальні завдання, розроблені на кафедрі програмування ЛНУ імені Івана Франка

9. Модуль, який включає опис типу вектора цілих чисел та процедури: читання елементів вектора з клавіатури; друкування такого вектора; визначення кількості підряд розміщених нульових значень у векторі.
10. Модуль, який включає опис типу вектора, елементами якого є слова (групи символів) довжиною вісім букв та процедури: читання значень такого вектора з клавіатури; друкування по два слова у рядок; обчислення кількості слів, які починаються і закінчуються однією буквою.
11. Модуль, який включає опис типу "*місяць*" (*січень*, *лютий*, ...), типу "*дата*" (*число* і *місяць*) і типу "*пори_року*" та процедури: перевірки передування однієї дати іншій; утворення дати наступного дня після заданої (невисокосний рік); визначення пори року для заданої дати.
12. Модуль, який включає опис типу матриці дійсних чисел та процедури: читання значень матриці; перестановки місцями двох заданих рядків чи стовпців; заміни всіх значень заданого рядка або стовпця на однакове вказане значення.
13. Модуль, який включає опис типу матриці цілих чисел, типу матриці логічних значень такого самого розміру та процедури: заміни знаків усіх чисел дійсної матриці на протилежні; побудови логічної матриці на основі матриці дійсних чисел за формулою $B[i, j] := X[i, j] >= 1$; знаходження позиції найбільшого числа матриці.
14. Модуль, який включає опис типу множини з латинських літер та процедури: утворення множини з прочитаного тексту, що вводиться з клавіатури, до крапки; друкування у зворотному алфавітному порядку літер цієї множини; обчислення кількості входжень у множину заданої літери.
15. Модуль, який включає опис переліченого типу, – назви міст (до 10), типу множини таких міст та процедури: друкування всіх міст списку, яких немає у жодній з двох заданих множин; друкування всіх міст списку, які є одночасно у кожній з двох заданих множин; введення з клавіатури назв міст (як літерних рядків – типу `string`) і утворення множини прочитаних міст.

9. ФАЙЛИ

План викладу матеріалу

1. Загальні положення
2. Процедури і функції роботи з файлами
3. Текстові файли
4. Компонентні файли
5. Безтипові файли
6. Послідовний і прямий доступ
7. Приклади роботи з файлами (режим консолі)
8. Робота з файлами у Object Pascal

Ключові терміни розділу

- *Означення файлу*
- *Буфер уведення/виведення*
- *Процедура* `Assign`
- *Процедура* `Rewrite`
- *Функція* `EOF`
- *Процедура* `Erase`
- *Текстовий файл*
- *Безтиповий файл*
- *Прямий доступ до файлу*
- *Функція* `FilePoz`
- *Операції введення/виведення*
- *Файлові змінні*
- *Процедура* `Reset`
- *Процедура* `Close`
- *Процедура* `Rename`
- *Функція* `IOResult`
- *Компонентний файл*
- *Послідовний доступ до файлу*
- *Функція* `FileSize`
- *Процедура* `Seek`

9.1. Загальні положення

Уведення файлового типу в мову Паскаль спричинене необхідністю забезпечити можливість роботи з периферійними (зовнішніми) пристроями комп'ютера, призначеними для введення і виведення даних.

Файловий тип даних, або **файл,** визначає впорядковану сукупність довільного числа однотипних *компонент* (або *елементів*). Загальна властивість масиву, множини і записів полягає в тому, що кількість їхніх компонент визначено на етапі написання програми, тоді як кількість компонент файлу у тексті програми не визначається і може бути довільною.

Під час роботи з файлами виконують операції введення-вивеення. Операція **введення** (**читання** – `Read`) означає перепис даних зі зовнішнього пристрою (з вхідного файлу) в основну пам'ять

комп'ютера, операція **виведення** (**запису** – `Write`) – це пересилання даних з основної пам'яті на зовнішній пристрій (у вихідний файл).

Файли на зовнішніх пристроях часто називають фізичними файлами. Їхні назви визначають за операційною системою. У програмах на мові Паскаль назви файлів задають за допомогою рядків. Наприклад, назва файлу на диску може мати вигляд:

```
'A:LAB1.DAT'
'c:\ABC150\pr.pas'
'lab3.pas'
```

З файловою системою ПАСКАЛЬ пов'язане поняття **буфера** введення-виведення. Введення і виведення здійснюється через буфер. Буфер – це область у пам'яті, яку виділяють для кожного файлу. При записі у файл вся інформація спочатку спрямовується у буфер і там нагромаджується доти, доки весь об'єм буфера не буде заповнений. Тільки після цього або після спеціальної команди скидання відбувається передача даних на зовнішній пристрій. У разі зчитування з файлу дані спочатку прочитуються в буфер, причому даних прочитується не стільки, скільки потрібно, а скільки вміститься у буфер.

Для роботи з файлами у програмі необхідно визначити файлову змінну. Паскаль підтримує три файлових типи: *текстові* файли, *компонентні* файли, *безтипові* файли. Описують файлові змінні текстового типу за допомогою слова `Text`, наприклад:

```
var tStory: Text;
```

Опис компонентних файлів має вигляд: `var fComp: File of T;`

де `T` – тип компоненти файлу. Приклади опису файлів компонентного типу: `var f1: File of Real; f2: File of Char;`

Безтипові файли описують словом `File`: `var f: File;`

Файлові змінні, що їх описано в програмі, називають **логічними файлами**. Всі основні процедури і функції, що забезпечують введення/виведення, працюють тільки з логічними файлами. Фізичний файл має бути пов'язаний з логічним до виконання процедур відкриття файлів.

9.2. Процедури і функції роботи з файлами

ПАСКАЛЬ уводить низку процедур і функцій, застосовних для будь-яких типів файлів:

- процедура `Assign(var f; FileName: String)` зв'язує логічний файл `f` з фізичним файлом, назва якого задана у рядку `FileName`. Якщо файл знаходиться в одній папці з програмою, то достатньо задавати тільки назву файлу, а інакше треба додатково вказувати повний шлях до файлу;
- процедура `Reset(var f)` відкриває логічний файл `f` для наступного читання даних або, як кажуть, відкриває вхідний файл. Після успішного виконання процедури `Reset` файл готовий до **читання** з нього першого елемента для **текстових** файлів і **читання/запису** – для **компонентних і безтипових** файлів;
- процедура `Rewrite(var f)` створює і відкриває логічний файл `f` для запису даних (відкриває вихідний файл). Якщо відповідний фізичний файл уже є, то його вміст очищається. Після успішного виконання цієї процедури **текстовий файл** готовий до запису в нього першого елемента і **запису/читання** – для **компонентних і безтипових** файлів;
- процедура `Close(var f)` закриває відкритий до цього логічний файл. Виклик процедури `Close` необхідний при завершенні роботи з файлом. Якщо з певних причин процедура `Close` для вихідного файлу не буде виконана, файл все ж буде створений на зовнішньому пристрої, але вміст останнього буфера у нього не буде перенесений. Для вхідних файлів використання оператора закриття файлу необов'язкове;
- логічна функція `EOF(var f): Boolean` повертає `TRUE`, коли при читанні досягнуто кінця файлу. Це означає, що вже прочитаний останній елемент у файлі або файл після відкриття виявився порожний;
- процедура `Rename(var f; NewName: String)` дає змогу перейменувати фізичний файл на диску, пов'язаний з логічним файлом `f`. Перейменуванння можливе після закриття файлу;

- процедура `Erase(var f)` знищує фізичний файл на диску, який був пов'язаний з файловою змінною `f`. Файл до моменту виклику процедури `Erase` має бути закритий;
- функція `IOResult: Integer` повертає ціле число, відповідне коду останньої помилки введення-виведення. При нормальному завершенні операції функція поверне значення 0. Значення функції `IOResult` необхідно привласнювати деякій змінній, оскільки при кожному виклику функція обнулює своє значення. Функція `IOResult` працює тільки при вимкненому режимі перевіряння помилок введення-виведення або з ключем компіляції `{$I-}`.

9.3. Текстові файли

Особливе місце у мові Паскаль займають текстові файли, компоненти яких мають символьний тип. Для опису текстових файлів у мові визначений тип `Text`: `var TF1, TF2: Text;`

Текстові файли – це послідовність рядків, а рядки – послідовність символів. Рядки мають змінну довжину, кожний рядок завершується ознакою кінця рядка – `EOLN` (*end of line*), який зображає символ #13 (символ переведення каретки – `CR`), за яким, можливо, розміщений символ #10 (символ переведення рядка – `LF`). Закінчується файл символом #26 (символом закінчення файлу – `EOF`).

Існує функція `EOLn(var T:Text):Boolean`, де `T` – назва текстового файлу. Ця функція набуває значення `TRUE`, якщо досягнуто кінця рядка. Для операцій над текстовими файлами, крім перерахованих вище у параграфі 9.2, ще визначені такі оператори звернення до процедур:

- `ReadLn(T)` – пропускає рядок до початку наступного;
- `WriteLn(T)` – завершує рядок файлу, в який проводиться запис, ознакою кінця рядка і переходить до початку наступного.
- `Read(T, X1, X2,...XK)` – читання значень з текстового файлу `T` у змінні `X1, X2,...XK`, які можуть бути змінними цілого, дійсного, символьного типу або рядком. При читанні

значень змінних з файлу вони перетворюються з текстового зображення у машинне;

- `Write(T, X1, X2,...XK)` – запис значень змінних `X1, X2, ... XK` у текстовий файл `T`. Змінні `X1, X2,...XK` можуть бути цілого, дійсного, символьного, логічного типу або рядком. При записі значень змінних у файл вони перетворюються з внутрішнього зображення у текстовий.

До текстових файлів відносять стандартні файли `INPUT`, `OUTPUT`. Робота з цими файлами має такі особливості:

- назви цих файлів у списках введення-виведення не вказують;
- застосування процедур `Reset`, `Rewrite` і `Close` до стандартних файлів введення-виведення заборонено;
- для роботи з файлами `INPUT/OUTPUT` введено різновид функції `EOLn` без параметрів.

Паскаль уводить додаткові процедури і функції, **застосовні тільки** до текстових файлів:

- процедура `Append(var f: Text)` служить для спеціального відкриття вихідних файлів. Вона застосовна до вже наявних фізичних файлів і відкриває їх для дозаписування даних у кінець файлу;
- процедура `Flush(var f: Text)` застосовується до відкритих вихідних файлів. Вона примусово записує дані з буфера у файл незалежно від ступеня його заповнення;
- функція `SeekEOLn(var f: Text): Boolean` повертає значення `True`, якщо до кінця рядка залишилися тільки пропуски;
- функція `SeekEOF(var f: Text): Boolean` повертає значення `True`, якщо до кінця файлу залишилися рядки, заповнені пропусками.

9.4. Компонентні файли

Компонентний (або типізований файл) – це файл з оголошеним типом його компонент. Компонентні файли складаються з машинних зображень значень змінних, вони зберігають дані у тому ж

вигляді, що і пам'ять комп'ютера. Опис величин файлового типу має вигляд:

```
type M= File Of T;
```

де M – назва файлового типу, T – тип компоненти. Наприклад:

```
type FIO= String[20]; SPISOK=File of FIO;
var  STUD, PREP: SPISOK;
```

Тут STUD, PREP – назви файлів, компонентами яких є рядки.
Опис файлів можна задавати в розділі опису змінних:

```
var fsimv: File of Char; fr: File of Real;
```

Компонентами файлу можуть бути всі скалярні типи, а з структурованих – масиви, множини, записи. Практично у всіх конкретних реалізаціях мови Паскаль конструкція "файл файлів" – недопустима.

Всі операції над компонентними файлами проводять за допомогою стандартних процедур: Reset, Rewrite, Read, Write, Close. Для введення-виведення використовують процедури: Read(f,X); Write(f,X); де f – назва логічного файлу, X – змінна, масив, рядок, множина або запис з таким же типом, що й у компоненти файлу.

Виконання процедури Read(f, X) полягає у читанні зі зовнішнього пристрою однієї компоненти файлу і запис її у X. Повторне застосування процедури Read(f, X) забезпечить читання наступної компоненти файлу і запис її у X.

Виконання процедури Write(f, X) полягає у записі X на зовнішній пристрій як однієї компоненти. Повторне застосування цієї процедури забезпечить запис X як наступної компоненти файлу.

Для роботи з компонентними файлами введена розширена форма операторів введення і виведення:

```
Read(f, X1, X2,..., XK)
Write(f, X1, X2,..., XK)
```

9.5. Безтипові файли

Безтипові файли дають можливість записувати на диск довільні ділянки пам'яті комп'ютера і прочитувати їх з диска у пам'ять. Операції обміну з безтиповими файлами здійснюються за допомогою процедур BlokRead і BlockWrite. Крім того, вводить-

ся розширена форма процедур `Reset` і `Rewrite`. У іншому принципи роботи залишаються такими ж, як і з компонентними файлами. Перед використанням логічний файл **var** `f:` **File**; має бути зв'язаний з фізичним за допомогою процедури `Assign`. Далі файл має бути відкритий для читання або для запису процедурою `Reset` або `Rewrite`, а після закінчення роботи закритий процедурою `Close`.

Відкриваючи файл, довжину буфера встановлюють за домовленістю у 128 байтів. Паскаль дає змогу змінити розмір буфера введення-виведення, для чого потрібно відкривати файл розширеним записом процедур

```
Reset(var f: File; BufSize: Word);
```

або

```
Rewrite(var f: File; BufSize: Word);
```

Параметр `BufSize` задає число байтів, що прочитуються з файлу або, що записуються в нього за одне звертання. Мінімальне значення `BufSize` – 1 байт, максимальне – 64 До байт.

Читання даних з безтипового файлу відбувається за допомогою процедури

```
BlockRead(var f:File; var X; Count: Word;
                                var QuantBlock: Word );
```

Ця процедура здійснює за одне звертання читання в змінну `X` кількості блоків, задане параметром `Count`, водночас довжина блоку рівна довжині буфера. Значення `Count` не може бути меншим 1. За одне звертання не можна прочитати більше, ніж 64 байти. Необов'язковий параметр `QuantBlock` повертає число блоків (буферів), прочитаних поточною операцією `BlockRead`. У разі успішного завершення операції читання `QuantBlock` = `Count`,

а у разі аварійної ситуації параметр `QuantBlock` буде містити число вдало прочитаних блоків. Звідси слідує, що за допомогою параметра `QuantBlock` можна контролювати правильність виконання операції читання.

Записують дані у безтиповий файл за допомогою процедури

```
BlockWrite(var f: File; var X; Count: Word;
                                var QuantBlock: Word);
```

яка за одне звертання записує зі змінної `X` кількість блоків, заданих параметром `Count`, водночас довжина блоку рівна довжині буфера. Необов'язковий параметр `QuantBlock` повертає число блоків (буферів), записаних поточною операцією `BlockWrite`.

9.6. Послідовний і прямий доступ

Смисл послідовного доступу полягає у тому, що в кожний момент часу доступна лише одна компонента зі всієї послідовності. Для того, щоб звернутися (отримати доступ) до компоненти з номером *n*, необхідно переглянути від початку файлу *n*-1 попередню компоненту. Після звертання до компоненти з номером *n* можна звертатися до компоненти з номером *n*+1. Звідси слідує, що процеси формування (записи) компонент файлу і перегляду (читання) не можуть довільно чергуватися. Отже, файл спочатку будують за допомогою послідовного додавання компонент у кінець, а потім його можна послідовно переглядати від початку до кінця.

Усі розглянуті засоби роботи з файлами забезпечують послідовний доступ до його компонент. ПАСКАЛЬ дає змогу застосовувати до *компонентних* і *безтипових* файлів, записаних на диск, спосіб прямого доступу. Прямий доступ означає можливість заздалегідь визначити у файлі блок, до якого буде застосована операція введення/виведення. У разі безтипових файлів блок рівний розміру буфера, для компонентних файлів блок − одна компонента файлу.

Прямий доступ передбачає, що файл – це лінійна послідовність блоків. Якщо файл містить *n* блоків, то вони нумеруються від 1 через 1 до *n*. Крім того, вводиться поняття умовної межі між блоками, разом з тим умовна межа з номером 0 розміщена перед блоком з номером 1, межа з номером 1 розміщена перед блоком з номером 2 і, нарешті, умовна межа з номером *n* знаходиться після блоку з номером *n*.

Прямий доступ здійснюється за допомогою функцій і процедур:

- функція `FileSize(var f): Longint` повертає кількість блоків у відкритому файлі `f`;
- функція `FilePos(`**`var`**` f): Longint` повертає поточну позицію у файлі `f`. Позиція у файлі − це номер умовної межі. Для щойно відкритого файлу поточною позицією буде межа з номе-

ром 0. Це означає, що можна записати або прочитати блок з номером 1. Після читання або запису першого блоку поточна позиція переміститься на межу з номером 1, і можна буде звертатися до блоку з номером 2. Після прочитання останнього запису значення `FilePos` дорівнює значенню `FileSize`;

- процедура `Seek(var f; N: Longint)` забезпечує призначення поточної позиції у файлі (позиціонування). У параметрі `N` має бути заданий номер умовної межі блоку, до якого проводитиметься дальше звертання. Наприклад, щоб працювати з блоком 4, необхідно задати значення `N`, рівне 3. Процедура `Seek` працює з відкритими файлами;
- процедура `Truncate(var f)` встановлює у поточній позиції ознаку кінця файлу і видаляє (стирає) всі наступні блоки.

9.7. Приклади роботи з файлами (режим консолі)

Приклад 9.1. Нехай на диску є текстовий файл `ID.DAT`, який містить числові значення дійсного типу по два числа в кожному рядку – значення аргументу і функції відповідно. Кількість пар чисел не більше за 20. Скласти програму, яка читає файл, значення аргументу і функції записує в одновимірні масиви, підраховує їхню кількість, виводить на екран дисплея і записує у файл компонентного типу `RD.DAT`.

```
program Project1;
{$APPTYPE CONSOLE}
uses SysUtils;
var rArg, rF: Array[1..20] of Real; inf: Text;
    outf: File of Real; n, l: Integer;
begin
 Assign(inf,'ID.DAT');  Assign(outf,'RD.DAT');
 Reset(inf);  Rewrite(outf); n:=0;
 while not EOF(inf) do
   begin n:=n+1; ReadLn(inf, rArg[n], rF[n]) end;
 for l:=1 to n do
  begin  WriteLn(l:2, rArg[l]:8:2, rF[l]:8:2);
         write(outf, rArg[l], rF[l])  end;
```

```
  readln;
  close(outf)
 end.
```

Приклад 9.2. Використавши файл компонентного типу RD.DAT, створений у попередньому прикладі, обчислити суму значень функції.

```
 program Project2;
 {$APPTYPE CONSOLE}
 uses  SysUtils;
 var inf: File of Real; Arg, rF, S: Real;
 begin  {$I-}
  Assign(inf,'RD.DAT'); Reset(inf); {I+}
  If IOResult<>0 then
   begin Writeln ('Error file'); readln; Halt; end;
  S:=0;
  while not EOF(inf) do
   begin read(inf, Arg, rF);S:=S+rF end;
   Writeln ('S=',S:8:2); readln; close(inf)
 end.
```

Приклад 9.3. Програма сортування файлу за неспаданням значень методом обміну (метод “бульбашки”).

```
Project3;
 {$APPTYPE CONSOLE}
uses SysUtils;
var tf:text; f: File of Integer; x,y,i,j: Integer;
begin {$I-}
  Assign(tf,'StartFile.txt');  Reset(tf);
  Assign(f,'SortFile.dat');  Rewrite(f); {I+}
  If IOResult<>0 then
   begin Writeln ('Error file'); readln; Halt; end;
{-------------------------------------------------}
{Виведення початкових, невпорядкованих даних}
  Writeln ('Start file: ');
```

```
  while not EOF(tf) do
  begin read(tf, x); Write(x:8);
   {Створення компонентного файлу}
   write(f,x)
  end;
  Writeln; Write('Enter key'); readln;
  close(tf); close(f);
{-----------------------------------------------}
{При прямому доступі Reset використовують як для }
{читання, так і для запису даних у файл}
  Reset(f);
  for i:=FileSize(f)-1 downto 1 do
   {Виділення чергового максимального числа}
   for j:=0 to i-1 do
    begin Seek (f,j); Read(f,x,y);
     if x>y then
              begin Seek (f,j); Write(f,y,x); end;
    end;
  close(f);
{-----------------------------------------------}
  {Виведення результатів сортування}
  Reset(f); Writeln ('Sorted file: ');
   for i:=1 to FileSize(f) do
    begin Read(f,x); Write(x:8); end;
   Writeln; Write('Enter key'); readln;
  Close(f);
end.
```

9.8. Робота з файлами у Object Pascal

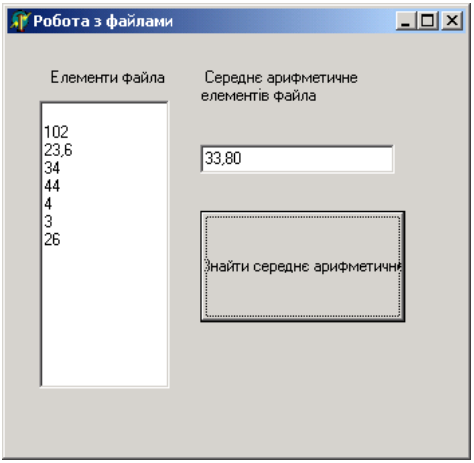
Працюючи з файлами у Object Pascal, треба знати таке:

- текстовий файл описують словом `TextFile`;
- процедури `Assign` і `Close` замінюються відповідно на процедури `AssignFile` і `CloseFile`. Функція `Erase` замінюється функцією `DeleteFile`.

Приклад 9.4. Створити віконне застосування для знаходження середнього арифметичного елементів файлу.

Форму для знаходження середнього арифметичного елементів файлу показано на рисунку 9.1. Етап **проектування інтерфейсу**:

```
object Form1: TForm1 ⇒  Caption = 'Робота з файлом'
 object Label1: TLabel ⇒  Caption = 'Елементи файлу'
```




```
object Label2: TLabel
⇒ Caption = 'Середнє
арифметичне елементів
файлу'
object Memo1: TMemo
  Lines.Strings = ''
  {порожне значення}
object Edit1: TEdit
Text=''{порожне зн-ня}
object Button1: TButton
 Caption = 'Знайти
середнє арифметичне'
 OnClick = Button1Click
```

Рис. 9.1. Форма для знаходження середнього арифметичного елементів файлу

Вміст **модуля форми**:

```
unit Unit1;
 interface
 uses
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms,
  Dialogs, StdCtrls;
 type
  TForm1 = class(TForm)
    Label1: TLabel;     Memo1: TMemo;
    Label2: TLabel;     Edit1: TEdit;
    Button1: TButton;
```

```
 procedure Button1Click(Sender: TObject);
  private { Private declarations }
  public  { Public declarations }
  end;
 var Form1: TForm1;
 Implementation
{$R *.dfm}
 procedure TForm1.Button1Click(Sender: TObject);
  var x, S: Real;  inf: TextFile; n:Integer;
    vStr: String;
 begin
  {$I-}
  AssignFile(inf,'ID.DAT'); Reset(inf);
  {$I+}
  If IOResult<>0 then
   begin ShowMessage('Помилка відкриття файлу');
         Halt;
   end;
  n:=0; S:=0;
  while not EOF(inf) do
   begin
     Read(inf, x);
     vStr:=FloatToStr(x);
     Memo1.Lines.Append(vStr);
     S:=S+x; n:=n+1;
   end;
   Edit1.Text:=FloatToStrF(S/n,fffixed,8,2);
   CloseFile(inf);
 end;
end.
```

- **? Запитання для самоперевірки**

1. Що таке файл?
2. Опишіть операції введення/виведення для файлів?
3. Для чого використовують буфер введення/виведення?

4. Для чого використовують файлові змінні?
5. Для чого використовують процедуру `Assign`?
6. Для чого використовують процедуру `Reset`?
7. Для чого використовують процедуру `Rewrite`?
8. Для чого використовують функцію `EOF`?
9. Для чого використовують процедуру `Close`?
10. Для чого використовують процедуру `Rename`?
11. Для чого використовують процедуру `Erase`?
12. Для чого використовують функцію `IOResult`?
13. Що таке текстовий файл?
14. Що таке компонентний файл?
15. Що таке безтиповий файл?
16. Як здійснюється послідовний доступ до файлу?
17. Як здійснюється прямий доступ до файлу?
18. Для чого використовують функцію `FileSize`?
19. Для чого використовують функцію `FilePoz`?
20. Для чого використовують процедуру `Seek`?

• Завдання для програмування

Завдання 9.1. *Текст* – довільна *послідовність символів*, що знаходиться у деякому файлі. *Слова* у тексті розділені пропусками. *Речення* – послідовність слів, що завершується крапкою (знаком оклику чи запитання). У режимі консолі написати програму, яка виконує такі дії:

1. Читає з файлу три послідовні речення (порядковий номер першого речення задається з клавіатури) і виводить їх на екран у зворотному порядку.
2. Читає текст з файлу і виводить на екран тількі ті речення, що містять задане слово.
3. Читає текст з файлу і виводить на екран тількі ті речення, які містять двозначні числа.
4. Читає текст з файлу і виводить на екран слова, що починаються з голосної букви.
5. Читає текст з файлу і виводить його на екран, міняючи місцями кожні два сусідні слова.

6. Читає текст з файлу і виводить на екран тільки ті речення, які не мають ком.
7. Читає текст з файлу і визначає, скільки у ньому слів, які мають не більше чотирьох букв.
8. Читає текст з файлу і виводить на екран тільки *цитати* – послідовність слів у лапках.
9. Читає текст з файлу і виводить на екран тільки ті речення, які мають визначену кількість слів.
10. Читає текст з файлу і виводить його на екран, замінюючи кожну першу букву слова на цю ж букву у верхньому регістрі (велику букву).
11. Читає текст з файлу, знаходить найкоротше слово і визначає, скільки разів воно трапляється у тексті.
12. Читає текст з файлу і виводить на екран спочатку запитання, а потім речення зі знаком оклику.
13. Читає текст з файлу і виводить на екран спочатку однобуквенні слова, а потім решту слів.
14. Читає текст з файлу, знаходить найдовше слово і визначає, скільки разів воно трапляється у тексті.
15. Читає текст з файлу і виводить на екран тільки ті речення, які мають максимальну кількість знаків пунктуації.

Завдання 9.2. Нехай маємо деякий *текстовий файл.* У режимі консолі скласти і протестувати підпрограму (процедуру або функцію), яка виконує таку дію:

1. Підраховує кількість порожних рядків.
2. Знаходить максимальну довжину рядка.
3. Знаходить мінімальну довжину рядка.
4. Підраховує кількість рядків, що починаються з голосної букви.
5. Підраховує кількість рядків, що закінчуються голосною буквою.
6. Підраховує кількість рядків, що починаються зі заданої букви.
7. Підраховує кількість рядків, що закінчуються заданою буквою.
8. Підраховує кількість рядків, що починаються і закінчуються заданою буквою.
9. У текстовому файлі записана непорожна послідовність цілих чисел, які відокремлені пропусками. Записати функцію для підрахунку суми всіх елементів файлу.

10. У текстовому файлі записана непорожна послідовність дійсних чисел, які відокремлені пропусками. Записати функцію для підрахунку добутку всіх елементів файлу.
11. Підраховує кількість рядків, що містять двозначні числа.
12. Підраховує кількість рядків, що містять однобуквенні слова.
13. Друкує на екрані окремими рядками вміст файлу.
14. Переписує вміст файлу у інший текстовий файл, зберігаючи ділення на рядки, але без порожних рядків.
15. Переписує вміст файлу у інший текстовий файл, зберігаючи ділення на рядки, але у новому файлі всі рядки мають мати фіксовану довжину (зайві символи видаляються, короткі рядки доповнюються пропусками).

Завдання 9.3. У текстовому файлі записана непорожна *послідовність дійсних чисел*, які відокремлені пропусками. Створити віконне застосування для розв'язання таких задач:

1. Знайти серед них найменше та номер його першого входження у файл.
2. Знайти серед них найбільше та номер його останнього входження у файл.
3. Обчислити суму всіх від'ємних значень.
4. Визначити кількість додатних, від'ємних і нульових значень.
5. Обчислити суму чисел, що належать проміжку [–3; 5].
6. Визначити відсоток додатних чисел.
7. Знайти середнє арифметичне чисел, розміщених за першим ненульовим. Вважати, що обов'язково є хоча б одне ненульове значення, причому не на останньому місці.
8. Знайти середнє арифметичне чисел, розміщених за першим ненульовим. Вважати, що ненульових значень може не бути взагалі.
9. Знайти різницю між найбільшим і найменшим числом.
10. Обчислити суму добутків сусідніх пар чисел (кожне число при обчисленні використовується тільки один раз).
11. Знайти позицію першого за порядком числа, що дорівнює нулю.
12. Знайти порядковий номер першого ненульового числа.
13. Знайти порядковий номер останнього ненульового числа.
14. Обчислити середнє арифметичне усіх додатних чисел.
15. Знайти суму модулів тих чисел, які передують першому нульовому, або усіх чисел групи, якщо нульових немає.

10. ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

План викладу матеріалу

1. Основні концепції об’єктно-орієнтованого програмування
2. Оголошення класу
3. Створення і використання об’єктів
4. Наслідування та поліформізм
5. Властивості та події

Ключові терміни розділу

- *Клас*
- *Методи класу*
- *Конструктори і деструктори*
- *Інкапсуляція*
- *Наслідування і поліморфізм*
- *Події*
- *Поля класу*
- *Розділи класу*
- *Створення об’єктів*
- *Кодифікатори методів*
- *Властивості*
- *Властивості-масиви*

10.1. Основні концепції об’єктно-орієнтованого програмування

У програмуванні широко використовують фундаментальний принцип керування складними системами – *devide et impera* (розділяй та владарюй, *лат.*), згідно з яким при проектуванні складної програмної системи здійснюють **алгоритмічну декомпозицію** задачі. Метою декомпозиції є подання системи у вигляді сукупності взаємодіючих підсистем, кожну з яких можна налагоджувати незалежно одна від одної.

Ідея розділення програм на самостійні частини, які реалізовують підпрограмами, знайшла відображення у **структурному підході** до створення програмних продуктів. У рамках цього підходу складають програми, структура яких нагадує “дерево”.

Дальший розвиток структурного підходу привів до **модульного програмування**, яке передбачає декомпозицію задачі у вигляді ієрархії взаємодіючих модулів (програм). Модуль, який містить дані та процедури їхнього опрацювання, є зручним для автономного проектування та налагодження. Розвитком ідеї модульного програмування є моделювання об’єктів предметної області за допомогою **класів** (об’єктних типів).

Об’єкт, який необхідно моделювати, уявляється людині цілісно, хоча він складається з частин та інших об’єктів. Цілісне уяв-

лення об’єкта у вигляді взаємозв’язаної сукупності його властивостей є базовим принципом об’єктного підходу. *Ієрархічна класифікація об’єктів і наслідування властивостей* стали початковими точками у розвитку з початку 80-х років минулого століття концепції **об’єктно-орієнтованого програмування** (**ООП**, скор.).

Основним поняттям ООП у Object Pascal є **клас**, який можна розглядати з двох позицій. По-перше, з позиції предметної області: *клас* відповідає визначеному *типовому* об’єктові цієї області. По-друге, з позиції технології програмування: *клас* – це визначена програмна структура, яка має три важливі властивості: *інкапсуляції*, *наслідування* та *поліморфізму*. Ці властивості використовуються програмістами, а забезпечуються об’єктно-орієнтованою мовою програмування (у нашому випадку Object Pascal) і відповідним компілятором мови. Вони дають змогу адекватно відображати структуру предметної області.

Інкапсуляція (дослівно – “вміст в оболонці”) − це об’єднання і локалізація у рамках класу, як єдиного цілого, даних і підпрограм (функцій і процедур), які опрацьовують ці дані.

Наслідування – властивість класу породжувати своїх нащадків − класи, які успадковують властивості батьківського класу. Клас-нащадок автоматично наслідує від батьківського класу його дані та підпрограми, а також містить нові дані та підпрограми чи перевизначає по-новому деякі батьківські підпрограми.

Поліформізм – властивість класів-родичів по-різному здійснювати однотипні (і навіть одинаково названі) дії, наприклад: дія “намалювати на екрані” має по-різному реалізовуватися для класів-родичів: “точка”, “пряма” і “прямокутник”.

10.2. Оголошення класу

Клас – це деякий абстрактний тип, який створюється на основі наявних типів. Окремий клас включає такі *елементи* (або члени), як **поля**, **методи**, **властивості** і **події**. **Поля** класу подібні до полів запису і служать для зберігання інформації про об’єкт. Поле описується як змінна, що може належати до будь-якого типу.

Методи – це процедури і функції, призначені для опрацювання полів. Опис методу схожий на опис звичайної підпрограми мо-

дуля. Заголовок методу розміщується в описі класу, а сам код методу знаходиться в розділі реалізації модуля. Назва методу в розділі реалізації є складовою і включає в себе назву класу.

Властивості займають проміжне положення між полями і методами. З одного боку, властивості об’єкта можна використовувати як поля, наприклад, їм можна надати значення за допомогою оператора присвоєння; а з іншого – всередині класу доступ до значень властивостей виконується методами класу. Кожній властивості відповідає поле, що містить її значення, і два методи, що забезпечують доступ до цього поля (читання/запис значень).

Подія – це спеціальна властивість типу `TNotifyEvent`.
Клас має таку *структуру*:

```
type НазваКласу= class (НазваКласуПредка)
      Відкриті / Опубліковані поля;
      Відкриті / Опубліковані методи;
      Відкриті / Опубліковані властивості;
      Відкриті / Опубліковані події;
  private   // Видно тільки у цьому модулі
      Закриті поля;
      Закриті методи;
      Закриті властивості;
      Закриті події;
  protected // Видно тільки у цьому модулі й нащадкам
      Захищені поля;
      Захищені методи;
      Захищені властивості;
      Захищені події;
  public      // Відкритий доступ
      Відкриті поля;
      Відкриті методи;
      Відкриті властивості;
      Відкриті події;
  published  // Відкритий доступ і видно в
             // Інспекторі об'єктів
      Опубліковані поля;
      Опубліковані властивості;
  end;
```

Клас `TObject` є базовим класом для всіх інших класів Object Pascal. Зауважимо, що об’єктів цього класу не існує. Класи,

які створюються в Object Pascal, за домовленістю успадковують усі методи класу `Tobject`.

В Object Pascal **усі об'єкти** розміщуються в динамічній пам'яті, тому для їхнього створення і знищення потрібно мати *конструктор* і *деструктор*. Конструктор `Create` класу `TObject` виконує початкові **стандартні** дії зі **створення об'єкта**:

- визначає у "*купі*" (**heap**) пам'ять для розміщення об'єкта;
- задає **нульові** значення для *числових* полів;
- задає значення **nil** для *покажчиків* і *полів-класів*;
- задає **порожні** значення для *полів-рядків*;
- повертає адресу розміщення у пам'яті створеного об'єкта.

Деструктор `Destroy` класу `TObject` звільняє динамічну пам'ять, виділену під об'єкт. Клас `TObject` має також процедуру `Free`, яка звільняє динамічну пам'ять, виділену під об'єкт, якщо об'єкт **дійсно** було створено (здійснення попередньої перевірки перед звільненням пам'яті). Конструктор `Destroy` **автоматично** викликається, якщо закінчується область дії і/або час існуваня відповідного об'єкта.

Якщо батьківським класом є **`TObject`**, то його назву після слова **`class`** можна не вказувати. Тоді перший рядок опису класу буде виглядати так:

`type` `НазваКласу =` **`class`**

`НазваКласу` може бути довільним ідентифікатором. Прийнято назву класу починати буквою "**`T`**" (від слова *Type*).

Для різних елементів класу можна встановлювати різні права доступу (видимість), для чого в описі класу використовуються окремі розділи, позначені спеціальними *специфікаторами видимості* (**`private, protected, public, published`**).

Специфікатор **`private`** використовують для задання **закритого** права доступу до полів класу, що дає змогу вирішити проблему захисту даних від непередбачуваних змін. Закриті поля є доступними тільки для методів свого класу і підпрограм, які знаходяться у тому ж модулі, що й цей клас.

Розділ **`protected`** містить **захищені** елементи, які доступні всередині модуля, в якому вони знаходяться. Елементи з цього роз-

ділу доступні також і для породжених класів за межами названого модуля. Статус захищений (**protected**) використовується в класах при застосуванні механізму успадкування класів. У разі відсутності успадкування специфікатор **protected** еквівалентний **private.**

Розділ **public** містить відкриті елементи, які видимі у будь-якому місці програми, де доступний сам клас. Специфікатор видимості **public** зазвичай використовують для задання відкритого доступу методам класу, які організують зв’язок об’єкта цього класу зі зовнішнім світом.

Розділ **published** складається з опублікованих елементів, область видимості яких така ж сама, як і відкритих елементів. Крім цього, опубліковані елементи породжують **динамічну** (під час виконання програми) **інформацію про тип** (Run-Time Type Information, RTTI). За цією інформацією при виконанні програми здійснюється перевірка на приналежність елементів об’єкта тому або іншому класу. Головним призначенням розділу **published** є забезпечення доступу до властивостей об’єктів під час конструювання проектів. У Інспекторові об’єктів видно тільки ті властивості, які є опублікованими.

Будь-які елементи, розміщені за рядком з вказівкою назви класу, вважаються загальнодоступними за домовленістю (водночас директива компілятора `$M` є виключеною, тобто `{$M-}`).

При включеній директиві компілятора `$M` (тобто `{$M+}`) будь-які елементи, розміщені за рядком з вказівкою назви класу, вважаються опублікованими.

Кількість і порядок розміщення розділів **private, protected, public, published** є довільними. Дія специфікатора доступу на члени класу починається з моменту його написання до нового специфікатора доступу або до кінця оголошення класу.

Оскільки класи найчастіше вводять для використання у багатьох програмах, то їх, як правило, описують у розділі **interface** деякого модуля. Реалізацію методів розміщують у розділі **implementation** цього модуля.

Методи, призначені для створення або видалення об’єктів, називаються **конструкторами** і **деструкторами**, відповідно. Описи

цих методів відрізняються від опису звичайних процедур тільки тим, що в їхніх заголовках стоять ключові слова **constructor** чи **destructor**. Як назви конструкторів/деструкторів у базовому класі `TObject` і багатьох інших класах використовують назви `Create` і `Destroy`**.** Конструктори/деструктори, введені програмістом, *доповнюють* конструктор/деструктор базового класу `Tobject` операторами ініціалізації полів і/або властивостей об'єкта під час його створення. Отже, якщо для створення об'єкта певного класу достатньо тільки стандартних дій, то в цьому класі конструктор можна не задавати.

Приклад 10.1. Визначити клас `TCircle` для опрацювання даних про коло. **Поля** – параметри кола: координати центра кола (`x`, `y`) і радіус `r`. **Методи** – зміна параметрів кола, отримання інформації про параметри кола, обчислення довжини кола і площі круга.

```
unit DefTCircle;
interface
 type TCircle=class
  private x,y,r:Real;
  public
   constructor create(x1,y1,r1:Real);
   function getX:Real; // отримання абсциси центра кола
   function getY:Real; // отримання ординати центра кола
   function getR:Real; // отримання радіуса
   function length:Real; // обчислення довжини кола
   function square:Real; // обчислення площі круга
   procedure setX(x1:Real); // задання абсциси центра
   procedure setY(y1:Real); // задання ординати центра
   procedure setR(r1:Real); // задання радіуса
 end;
implementation
  constructor TCircle.Create;
    begin x:=x1; y:=y1; r:=r1 end;
  function TCircle.getX;    begin getX:=x end;
  function TCircle.getY;    begin getY:=y end;
  function TCircle.getR;    begin getR:=r end;
  function TCircle.length;  begin length:=2*Pi*r end;
  function TCircle.square;  begin square:=Pi*r*r end;
  procedure TCircle.setX;   begin x:=x1 end;
```

```
  procedure TCircle.setY;   begin y:=y1 end;
  procedure TCircle.setR;   begin r:=r1 end;
end.
```

10.3. Створення і використання об’єктів

Оголошення класу ще не задає об’єкта, а тільки визначає тип того об’єкта, який буде створений у програмі пізніше. Отже, після оголошення класу `TCircle` можна визначити один або декілька об’єктів, як екземпляри цього класу. Для описаного вище класу, визначення об’єктів може виглядати так:

```
var c1,c2, c3: TCircle;
```

Тут визначені об’єкти `c1`, `c2` і `c3` типу (класу) `TCircle`. В Object Pascal усі об’єкти розміщуються в динамічній пам’яті, а тому вони фактично є **покажчиками** спеціального виду (операцію рознайменування виконують без задання символу `'^'`). Під час компілювання програми кожному об’єкту, незалежно від його класу, виділяється фіксована ділянка пам’яті (чотири байти) для зберігання адреси розміщення об’єкта в динамічній пам’яті.

Перш ніж звертатися до елементів об’єкта, його треба *створити* за допомогою **конструктора** і, при потребі, задати у ньому початкові значення полів об’єкта. Виклик конструктора може виконуватися двома способами: за допомогою задання *назви класу* або *назви* конкретного *об’єкта*. При створенні нового об’єкта виклик конструктора реалізується тільки через *назву класу*, наприклад:

```
c1:=TCircle.Create; // Покажчик (об'єкт) c1 отримує
                    // адресу розміщення у динамічній пам'яті
                    // нового об'єкта класу TCircle
```

Якщо конструктор `Create` доповнювався у відповідному класі, то під час виконання він має ініціалізувати поля і/або властивості об’єкта за допомогою значень, що передаються як аргументи конструктора, наприклад:

```
c2:=TCircle.Create(x2,y2,r2);
```

Конструктор класу під час створення об’єкта виконує такі дії:

- реалізує стандартні дії зі створення об’єкта (викликається конструктор `Create` класу `TObject`);
- виконує дії, які задані програмістом.

Якщо конструктор викликається як метод уже наявного об'єкта (за допомогою *назви* цього об'єкта), то виконуються тільки дії конструктора, які задані програмістом. У такий спосіб, як правило, викликають конструкторів-нащадків. Для доступу до відкритих полів, властивостей чи методів об'єкта використовують складені назви: `об'єкт.поле/властивість` або `об'єкт.метод[(аргументи)]`

Приклад 10.2. Використати клас `TCircle` для програмування задач:

1. На площині задати 15 кіл.
2. Надрукувати інформацію про кола, довжина яких перевищує задане значення `L`.
3. Кола, площею меншою за 2,75, пересунути на початок координат.
4. Для кіл, центр яких є у 3-му квадранті, зменшити радіус удвічі.

```
program Project2;
{$APPTYPE CONSOLE}
uses SysUtils, DefTCircle in 'DefTCircle.pas';
const NC=15;
var circles: array[1..NC] of TCircle;
    x,y,r:Real; i:byte;  C,L,S:Real;
begin  // Реалізація задачі 1
  writeln('Enter centr and radius');
  for i:=1 to NC do
   begin read(x,y,r); circles[i]:=TCircle.Create(x,y,r) end;
  // Реалізація задачі 2
  writeln('Enter limit of C'); read(L);
  for i:=1 to NC do
   begin C:= circles[i].length;
    If C>L then
     begin  r:=circles[i].getR; x:=circles[i].getX;
      y:=circles[i].getY; writeln('1: Centr: ',x:6:1,
                    ' ',y:6:1,' R=',r:6:1) end
   end;
  // Реалізація задачі 3
  for i:=1 to NC do
   begin S:=circles[i].square;
    If S<2.75 then
     begin r:=circles[i].getR;
      circles[i].setX(0);  circles[i].setY(0);
      x:=circles[i].getX;  y:=circles[i].getY;
      writeln('2: Centr: ', x:6:1,' ',
```

```
                    y:6:1,' R=',r:6:1)      end
   end;
  // Реалізація задачі 4
  for i:=1 to NC do
   begin x:=circles[i].getX; y:=circles[i].getY;
    If (x<0) and (y<0) then
      begin r:=circles[i].getR; r:=r/2;
       circles[i].setR(r); r:=circles[i].getR;
       writeln('3: Centr: ',x:6:1,' ', y:6:1,' R=',r:6:1)
      end
   end;   readln
end.
```

Приклад 10.3. Визначити клас **Tkafedra** для опрацювання загальних даних про кафедру. **Поля**: *назва, кількість співробітників* (не більше 15)*, кількість курсів* (не більше 15)*, списки викладачів і курсів*. Визначити методи, необхідні для реалізаії таких **задач**:

1. Сформувати дані про декілька кафедр.
2. Для заданої кафедри модифікувати список спецкурсів.
3. Вивести назви кафедр, викладачі яких читають певний курс.
4. Для кафедр, які мають найбільшу кількість викладачів, вивести списки викладачів.

```
program Project3;
{$APPTYPE CONSOLE}
uses SysUtils, Unit1; //Unit1 - див. після програми
const cKaf=3;
var kafs: array[1..cKaf] of TKafedra;
    i,j,nmax,nv,nk:byte; name,kurs:Str; c:char;
begin   // 1 task
 for i:=1 to cKaf do
  begin writeln('Enter kilkist vycl ',i,' kafedry');
   readln(nv); if nv>15 then nv:=15;
   writeln('Enter kilkist kursiv ',i,' kafedry');
   readln(nk); if nk>15 then nk:=15;
   writeln('Enter nazvu kafedry'); readln(name);
   kafs[i]:=TKafedra.Create(nv,nk,name);
   writeln('Enter name ',nv,' vycladachiv');
   for j:=1 to nv do
    begin  readln(name); kafs[i].modName(j,name) end;
   writeln('Enter ',nk,' kursiv');
```

```
   for j:=1 to nk do
   begin  readln(kurs); kafs[i].modKursy(j,kurs) end
  end;
    // 2 task
writeln('Enter name kafedry'); readln(name);
for i:=1 to cKaf do
 if kafs[i].getKafName=name then Break;
nk:=kafs[i].getcK;
for j:=1 to nk do
 begin kurs:=kafs[i].getKursy(j);
    write(' ',kurs,' Miniaty (y/n)?');readln(c);
    if c='y' then
     begin  write('Vvesty novu nazvu');
        readln(name); kafs[i].modKursy(j,name) end
 end;
repeat write('Vvesty novu nazvu(y/n)?');readln(c);
 if c='y' then begin  write('Vvesty novu nazvu:');
            readln(name); kafs[i].addKursy(name) end
until c='n';
 // 3 task
writeln('Vvesty nazvu kursu: ');readln(name);
for i:=1 to cKaf do
 begin nk:=kafs[i].getcK;
  for j:=1 to nk do
   begin  kurs:=kafs[i].getKursy(j);
    if kurs=name then
     begin writeln(kafs[i].getKafName);break end
   end
 end;
  // 4 task
nmax:=kafs[1].getcV;
for i:=2 to cKaf do
 begin nv:=kafs[i].getcV; if nv>nmax then nmax:=nv end;
writeln('Spysok vycladachiv naybilshyh kafedr:');
for i:=1 to cKaf do
 begin nv:=kafs[i].getcV;
  if nv=nmax then
   begin name:=kafs[i].getKafName;
    write('Kafedra ',name,': ');
    for j:=1 to nv do
     begin name:=kafs[i].getName(j); write(name,' ') end;
```

```
      writeln;
     end
  end;      readln
end.
```

Файл модуля Unit1:

```
unit Unit1;
interface
 type Str=String[25];
  TKafedra=class
   private
     cVycl,cKurs:byte;  kafname:Str;
     name:array[1..15] of Str;
     kursy:array[1..15] of Str;
   public
     constructor create(nV,nK:byte; nKafN:Str);
     function getcV:byte;   function getcK:byte;
     function getKafName:Str;
     function getName(i:byte):Str;
     function getKursy(i:byte):Str;
     procedure modName(i:byte; val:Str);
     procedure modKursy(i:byte; val:Str);
     procedure addName(val:Str);
     procedure addKursy(val:Str);
  end;
implementation
  constructor TKafedra.create;
   begin cVycl:=nV; cKurs:=nK; kafname:=nKafN end;
  function TKafedra.getcV; begin getcV:=cVycl  end;
  function TKafedra.getcK; begin getcK:=cKurs end;
  function TKafedra.getKafName;
           begin getKafName:=kafname  end;
  function TKafedra.getName;
           begin getName:=name[i] end;
  function TKafedra.getKursy;
           begin getKursy:=Kursy[i]   end;
  procedure TKafedra.modName; begin name[i]:=val end;
  procedure TKafedra.modKursy;begin Kursy[i]:=val  end;
  procedure TKafedra.addName;
   begin if cVycl<15
          then begin inc(cVycl);name[cVycl]:=val end
```

```
          else  writeln('Greater limit')
   end;
  procedure TKafedra.addKursy;
   begin if cKurs<15
          then begin inc(cKurs);Kursy[cKurs]:=val end
          else  writeln('Greater limit')
   end;
end.
```

Приклад 10.4. Визначити клас для роботи зі стеком. Стек має зберігати інформацію про календарні дати: *рік*, *місяць*, *число*. Реалізувати стек у вигляді однонапрямленого списку. **Методи**: занесення дати у стек (`put`); отримання дати зі стеку (`get`); ознака наявності даних у стекові (`empty`); очищення стека (`clear`). **Задачі**:

1. З клавіатури вводять деякі дати. Дати, що стосуються останнього року, записати в один стек, а решту дат – в інший стек.
2. Надрукувати всі дати останнього року, читаючи їх зі стека.
3. Зі стека попередніх років надрукувати лише три останні дати, а решту вилучити.

Файл модуля `Unit1`:

```
unit Unit1;
interface
 type Ty=1950..2003; Tm=1..12; Td=1..31;
  TMyDate=class
   private  y:Ty; m:Tm; d:Td; Link:TMyDate;
   public
    procedure put(y1:ty;m1:Tm;d1:Td);
    procedure get(var y1:ty;var m1:Tm; var d1:Td);
    function empty:boolean;  procedure clear;
  end;
implementation
  function TMyDate.empty;begin empty:=(Link=nil) end;
  procedure TMyDate.put;
  var p:TMyDate;
  begin
   if empty then
     begin Link:=TMyDate.Create; Link.y:=y1;
           Link.m:=m1; Link.d:=d1; Link.Link:=nil end
   else begin p:=TMyDate.Create();
```

```
          p.y:=y1; p.m:=m1; p.d:=d1;
          p.Link:=Link; Link:=p     end
  end;
  procedure TMyDate.get;
  var p:TMyDate;
  begin
   if not empty then
      begin y1:=Link.y; m1:=Link.m; d1:=Link.d;
            p:=Link; Link:=p.Link; p.Free      end;
  end;
  procedure TMyDate.clear;
  var p:TMyDate;
  begin
   if not empty then
    while Link.link<>nil do
        begin p:=Link; Link:=p.Link; p.Free end;
  end;
end.
```

Програма:

```
program Project4;
{$APPTYPE CONSOLE}
uses SysUtils, Unit1 in 'Unit1.pas';
const n=5;
var e,p:TMyDate; y:Ty; m:Tm; d:Td; i:byte;
begin e:=TMyDate.Create;  p:=TMyDate.Create;
 writeln('Enter ',n, ' date(yyyy,mm,dd):'); // Task 1
 for i:=1 to n do
  begin readln(y,m,d);
        if y=2003 then e.put(y,m,d) else p.put(y,m,d)
  end;
while not e.empty do // Task 2
  begin e.get(y,m,d); writeln(d:2,' ',m:2,' ',y:4) end;
for i:=1 to 3 do  // Task 3
  if not p.empty then
   begin p.get(y,m,d);writeln(d:2,' ',m:2,' ',y:4) end;
 p.clear; writeln('Enter number'); read(i)
end.
```

Як бачимо, об’єкти, як примірники класу, оголошують в програмі у розділі `var` як звичайні змінні. Для звернення до конкрет-

ного елемента об'єкта (поля, властивості або методу) вказують назву об'єкта і назву елемента, що розділені крапкою.

Приклад 10.5. Побудувати клас `Complex`, який включає опис комплексного типу даних (числа вигляду $m+n\cdot i$, де m і n – дійсні числа, а i – уявна одиниця). Операції над комплексними числами (додавання, віднімання, множення, ділення) і виведення на екран комплексного числа реалізувати у вигляді методів. Перевірити роботу класу на прикладі обчислення виразу $d=\frac{a+b-(2+3\cdot i)\cdot c}{a+2\cdot b}$, де a, b і c – комплексні числа.

Попередні міркування. Специфіка класу накладає певні обмеження на вигляд методів. Об'єкт, який викликатиме метод-процедуру, що реалізує операцію над комплексними числами, буде отримувати значення операції над двома об'єктами – арґументами цього методу. Об'єкт, який викликає методи, що реалізують операції додавання/віднімання, може одночасно бути і об'єктом – арґументом цього методу. Ініціалізацію полів об'єкта будемо здійснювати тільки за допомогою конструктора. Змінити значення полів об'єкта можна, очевидно, під час виконання методів, які реалізують операції над комплексними числами.

Файл модуля `Cmplx`:

```
unit Cmplx;
interface
Type Complex=class
 private
  Re:real; Im:real;
 public
   constructor create(x,y:Real);
   Procedure AddC(x,y:Complex);
   Procedure SubC(x,y:Complex);
   Procedure MulC(x,y:Complex);
   Procedure DivC(x,y:Complex);  Procedure WriteC;
end;
implementation
  constructor Complex.Create;
   begin Re:=x; Im:=y end;
  Procedure Complex.AddC;
   begin  Re:=x.Re+y.Re; Im:=x.Im+y.Im  end;
```

```
  Procedure Complex.SubC;
   begin  Re:=x.Re-y.Re; Im:=x.Im-y.Im  end;
  Procedure Complex.MulC;
   begin   Re:=x.Re*y.Re-x.Im*y.Im;
            Im:=x.Re*y.Im+x.Im*y.Re
   end;
  Procedure Complex.DivC;
   var z:real;
   begin z:=sqr(y.Re)+sqr(y.Im);
         if z=0 then begin writeln('Divide by 0');
                           readln; halt  end;
         Re:=(x.Re*y.Re+x.Im*y.Im)/z;
         Im:=(x.Re*y.Im-x.Im*y.Re)/z
   end;
  Procedure Complex.WriteC;
   begin write('Result='); write(Re:8:2);
    if Im>0 then write('+'); write(Im:8:2,' I'); readln
   end;
end.
```

Програма:

```
program Project5;
{$APPTYPE CONSOLE}
uses SysUtils, Cmplx in 'Cmplx.pas';
var a,b,c,d,e,r:Complex; x,y:real;
begin writeln('Enter 3 coplex numbers (Re Im):');
 read(x,y); a:=Complex.Create(x,y); d:=Complex.Create(0,0);
 read(x,y); b:=Complex.Create(x,y); r:=Complex.Create(0,0);
 read(x,y); c:=Complex.Create(x,y); x:=2; y:=3;
 e:=Complex.Create(x,y); d.mulc(e,c); e.AddC(a,b);
 e.SubC(e,d); d.AddC(a,b);d.AddC(d,b);
 r.DivC(e,d); r.WriteC; readln
end.
```

10.4. Наслідування та поліформізм

У попередніх параграфах було детально описано властивість *інкапсуляції* класу. У цьому параграфі зупинимося на властивостях *наслідування* та *поліформізму*. Властивість **наслідування** означає, що будь-який клас може бути породжений іншим класом, як-от:

```
TNewClass=class(TOldClass) ...
```

Клас-нащадок автоматично успадковує всі поля, методи і властивості класу-предка і може доповнюватися новими. Але разом з тим видалити або перевизначити поля класу-предка у класі-нащадкові неможливо. Отже, чим далі за ієрархією який-небудь клас знаходиться від батьківського класу, тим більше полів він має.

У Object Pascal усі класи є нащадками класу **TObject**. Цей клас не має полів і властивостей, але його методи дають змогу створювати, підтримувати діяльність і знищувати об'єкти. Використання принципу наслідування привело до створення у Delphi розгалуженого дерева класів, у верхній частині якого знаходяться **абстрактні класи**, для яких не можна створити об'єкти. Водночас вони є предками класів, для яких уже створюються реальні об'єкти.

Властивість **поліформізму** дає можливість використовувати однакові назви для методів, які входять у різні класи. При звертанні до таких методів буде виконуватися той з них, що відповідає класові об'єкта. Ця властивість забезпечується через *перекриття* у класові-нащадкові відповідного методу батьківського класу.

Метод, оголошений в класі, можна викликати різними способами, що залежать від **типу** методу, який визначає механізм його перекриття у класах-нащадках. Тип методу визначають за допомогою кодифікатора, який вказують в описі класу після заголовка методу і відділяють від заголовка крапкою з комою. Кодифікатори:

- **`virtual`** – віртуальний метод;
- **`dynamic`** – динамічний метод;
- **`override`** – перевизначений метод;
- **`message`** – опрацювання повідомлення;
- **`abstract`** – абстрактний метод.

За домовленістю всі методи, оголошені в класі, є *статичними* і викликаються як звичайні підпрограми. Визначення конкретних адрес статичних методів та їхнє зв'язування виконуються уже під час компіляції (**раннє зв'язування**). Отже, такі методи не можна перевизначати для реалізації поліморфізму. Якщо у класові-нащадкові є метод з назвою, що збігається з назвою деякого методу класу-предка, то дія методу класу-предка відміняється для класу-нащадка повністю. Наприклад:

```
type TBase=class
        procedure MyMetod; …  end;
     TDesc=class (TBase)
        procedure MyMetod; …  end;
var ObBase: Tbase; ObDesc: Tdesc;
begin … ObBase.MyMetod; … ObDesc.MyMetod; … end;
```

Методи `MyMetod` у обох класах є статичними. Тому в операторі `ObBase.MyMetod;` викликається метод, описаний у класі `TBase`, а в операторі `ObDesc.MyMetod;` викликається метод, описаний у класі `TDesc`.

Динамічні та **віртуальні** методи відрізняються від статичних тим, що заміщення батьківських методів методами нащадків відбувається під час виконання програми (**пізнє зв’язування**). Для оголошення віртуального методу в батьківському класі використовують слово `virtual`, а у класі-нащадкові – слово `override`. Наприклад:

```
type TBase=class
        procedure MyMetod; virtual; … end;
     TDesc=class (Tbase)
        procedure MyMetod; override; …  end;
var ObBase: Tbase; ObDesc: Tdesc;
begin … ObBase.MyMetod; … ObDesc.MyMetod; … end;
```

Для оголошення методу `MyMetod` **динамічним** треба задати:

```
procedure MyMetod; dynamic;
```

Різниця між віртуальними та динамічними методами полягає в особливостях реалізації їхніх викликів. Можна сказати, що віртуальні методи більш ефективні з погляду затрат часу, а динамічні – затрат пам’яті.

Абстрактними називають віртуальні та динамічні методи, які визначені у класі, але не містять ніяких дій, ніколи не викликаються і мають перевизначатися у класах-нащадках. Основне призначення абстрактних методів – бути родоначальниками ієрархії конкретних методів у класах-нащадках. Нариклад:

```
procedure MyAbsMetod; dynamic; abstract;
```

Зупинимося на реалізації конструкторів/деструкторів у класах-нащадках. Якщо у такому класі при створенні об’єктів треба виконати деякі додаткові дії, які відсутні у класі-предкові, то у кон-

структорі класу-нащадка треба **попередньо** викликати конструктор класу-предка, а потім вже здійснювати додаткові дії. Викликати будь-який перекритий метод батьківського класу, у т.ч. конструктор/деструктор, можна за допомогою слова `inherited` (успадкований), наприклад:

```
inherited Create; //виклик конструктора предка
```

Реалізація деструктора класу-нащадка, як правило, **завершується** викликом деструктора батьківського класу `inherited Destroy;` який знищує успадковані поля.

У класі за допомогою слова **`class`** можна визначити **методи класу**, які будуть викликатися і без створення об'єкта. Методи класу призначені для отримання довідкової інформації про клас – назву класу; предка класу, розмір класу тощо. Наприклад:

```
type TAnyClass=class
      class function Info: string;  … end;
var x: string;  …
begin … x:= Info; … end;
```

Приклад 10.6. Взяти за базовий клас `TCircle` (з другого параграфа цього розділу). Побудувати для нього клас-нащадок `TRing` для опрацювання даних про **кільце** (два концентричні кола). Доповнити його новим полем `Q` – для зберігання радіуса другого кола. Перекрити метод обчислення площі круга (для отримання площі кільця). Використати класи `TCircle` і `TRing` для програмування задач:

1. На площині задати масив з восьми об'єктів (перші три – класу `TCircle`, а решта – класу `Tring`).
2. Надрукувати інформацію про площі об'єктів.

Файл модуля `Unit1`:

```
unit Unit1;
interface
 type TCircle=class
  private x,y,r:Real;
  public
   constructor create(x1,y1,r1:Real);
   function getX:Real;  function getY:Real;
   function getR:Real;  function length:Real;
   function square:Real; virtual;
```

```
    procedure setX(x1:Real); procedure setY(y1:Real);
    procedure setR(r1:Real);  end;
   TRing=class(TCircle)
    private  Q:Real;
    public
     constructor create(x1,y1,r1,q1:Real);
     function getQ:Real; procedure setQ(q1:Real);
     function square:Real; override;
   end;
implementation
 constructor TCircle.create;
  begin x:=x1; y:=y1; r:=r1 end;
 function TCircle.getX;  begin getX:=x end;
 function TCircle.getY;  begin getY:=y end;
 function TCircle.getR;   begin getR:=r end;
 function TCircle.length; begin length:=2*Pi*r end;
 function TCircle.square; begin square:=Pi*r*r end;
 procedure TCircle.setX;  begin x:=x1 end;
 procedure TCircle.setY;  begin y:=y1 end;
 procedure TCircle.setR;  begin r:=r1 end;
 constructor TRing.create;
  begin inherited Create(x1,y1,r1);Q:=q1 end;
 function TRing.getQ;     begin getQ:=Q  end;
 procedure TRing.setQ;    begin Q:=q1    end;
 function TRing.square;
  begin if r>Q then square:=Pi*(r*r-Q*Q)
               else square:=Pi*(Q*Q-r*r) end;
end.
```

Програма:

```
program Project6;
{$APPTYPE CONSOLE}
uses SysUtils,  Unit1 in 'Unit1.pas';
var list: array[1..8] of TCircle;
    x,y,r,q:Real; i:byte; S:Real;
begin // реалізація задачі 1
 for i:=1 to 8 do
  begin if i<4
   then begin writeln('Enter centr and radius');
      read(x,y,r); list[i]:=TCircle.Create(x,y,r) end
   else begin  writeln('Enter centr and 2 radiuses');
```

```
    read(x,y,r,q); list[i]:=TRing.Create(x,y,r,q) end
  end;
// реалізація задачі 2
 for i:=1 to 8 do
  begin S:=List[i].Square; writeln('Square=',S) end;
   readln
end.
```

Будь-який метод (і статичний, і віртуальний) може бути перевизначений за допомогою ключового слова **overload**. Для віртуальних методів у цих випадках треба ще додати ключове слово **reintroduce**. Якщо перевизначені таким чином методи різняться числом або типом параметрів (т. з. **сигнатурою**), то при виклику методу з класу-нащадка буде викликатися той метод, список параметрів якого відповідає списку аргументів. Наприклад:

```
type T1 = class(TObject)
 procedure Test(I: integer); overload; virtual; ... end;
 T2 = class(T1)
 procedure Test(S: string); reintroduce; overload; ... end;
MyObject:= T2.Create;
MyObject.Test('Hello!'); // викликається  T2.Test
MyObject.Test(7); // викликається   T1.Test
```

10.5. Властивості та події

10.5.1. Властивості

Поля даних, за принципом інкапсуляції, завжди мають бути захищені від несанкціонованого доступу. Доступ до них, як правило, здійснюється тільки через **властивості**, що включають методи читання і записи полів. Тому поля доцільно оголошувати в розділі **private** (закритому розділі класу). У деяких випадках їх можна вміщувати у **protected** (захищеному розділі класу), щоб можливі нащадки цього класу мали до них доступ. Традиційно ідентифікатори полів збігаються з назвами відповідних властивостей, але з доданням префікса `'F'`. Властивість оголошують так:

```
  property <назва_властивості>:<тип>
     read  < назва_поля (або назва_методу_читання) >
     write < назва_поля (або назва_методу_запису) >
     <директива_запам'ятовування>;
```

Якщо в розділах **read** або **write** цього оголошення записана назва_поля, то передбачається пряме читання або запис даних.

Якщо в розділі **read** записана назва_методу_читання, то читання буде здійснюватися тільки функцією з цією назвою. Функція читання – це функція без параметра, яка повертає значення того типу, який оголошений для властивості. Назву функції читання прийнято починати з префікса **Get**, після якого слідує назва властивості. Якщо **read** відсутній в оголошенні властивості, то ця властивість призначена тільки для запису і користувач не може прочитати її значення.

Якщо в розділі **write** записана назва_методу_запису, то запис буде здійснюватися тільки процедурою з цією назвою. Процедура запису – це процедура з одним параметром того типу, який оголошений для властивості. Назву процедури запису прийнято починати з префікса **Set**, після якого слідує назва властивості. Якщо **write** відсутній в оголошенні властивості, то ця властивість призначена тільки для читання і користувач не може задавати її значення.

Директива запам’ятовування визначає, як треба зберігати значення властивості при збереженні користувачем файлу форми (.dfm). Для цього використовується директива:

```
default <значення_за_домовленістю>
```

Ця директива не задає початкові умови. Це справа конструктора. Директива просто вказує, якщо користувач у процесі проектування не змінив значення властивості за домовленістю, то зберігати значення властивості не треба.

Перелічимо можливості конструкції “*властивість*” детальніше:

- властивості обмежують доступ до полів, указуючи конкретні методи для читання і для запису значення поля;
- звертання до властивостей у програмі здійснюється зазвичай через *оператор присвоювання*, який є дуже простим і зручним засобом роботи з даними;
- властивості можна оголосити з характеристиками “*тільки для читання*” і “*тільки для запису*”;
- методи доступу до властивості можуть бути оголошені *вірту-*

альними, що надає додаткову гнучкість у програмуванні;

- при створенні власних компонентів, після того, як вони будуть поміщені на палітрі компонентів Delphi, їхні опубліковані властивості стають доступними через Object Inspector.

10.5.2. Події

Подія – це спеціальна властивість, яка є покажчиком типу `TNotifyEvent`. Подібний тип використовується у Delphi для подій типу `OnClick` і багатьох інших, які у процедурі опрацювання події мають **тільки один** параметр – `Sender: TObject`. Якщо потрібно ввести у клас подібну подію, досить визначити в оголошенні класу відповідну властивість. Наприклад:

```
private FMyEvent: TNotifyEvent;
public
  property MyEvent:TNotifyEvent read FmyEvent write FMyEvent;
```

Зауваження. У консольних застосуваннях для використання покажчика типу `TNotifyEvent` треба підключати модуль `Classes`.

Залишається тільки викликати у момент виникнення події процедуру опрацювання користувачем (програмістом) цієї події. Назва процедури опрацювання події збігається з назвою властивості, перед якою додається префікс `'On'` (подія `OnMyEvent` для нашого випадку). Виклик процедури опрацювання події має вигляд:

```
OnMyEvent(t);
```

де аргумент `t` відображає об'єкт, в якому відбулася подія. Кожний об'єкт має вбудований покажчик `Self`, який містить адресу розташування цього об'єкта у пам'яті. Найчастіше у процедурі опрацювання події аргументом є саме покажчик `Self`.

Місце, куди треба включати подібний оператор, залежить від виклику події. Наприклад, подія викликається якимось з методів користувача: виклик процедури опрацювання події треба здійснювати з того самого методу. Якщо подія пов'язана з якимось повідомленнями, що надходять від інших застосувань або від операційної системи Windows, то треба передбачити процедуру опрацювання відповідного повідомлення і з нього вже викликати процедуру опрацювання події користувача.

Приклад 10.7. Побудувати клас, який включає опис раціонального типу `Rat` (числа вигляду $\frac{m}{n}$, де m і n – цілі числа, $n \neq 0$) і методи:

- реалізації операцій над раціональними числами (додавання, віднімання, множення, ділення);
- присвоєння початкових значень раціональним числам за допомогою введення їхніх значень з клавіатури;
- виведення на екран раціонального числа;

Для доступу до полів класу використовувати **властивості**.

Попередні міркування. При програмуванні класу будемо вважати, що функція, яка реалізує операцію над раціональними числами, повертатиме значення операції над об’єктом – аргументом фукції і об’єктом – джерелом виклику відповідного методу. Для **внутрішніх** потреб класу введемо процедури:

- спрощення раціонального числа (скорочення спільних множників у чисельнику та знаменнику дробу; присвоєння чисельнику знака числа);
- опрацювання події `myEvent` (поява нуля у знаменнику дробу під час виконання операцій над раціональними числами).

```
unit Unit1;
interface
uses Classes;
Type Rat=class
private Fch,Fzn:integer; FmyEvent:TNotifyEvent;
  procedure Setch(const Value: integer);
  procedure Setzn(const Value: integer);
  function  Getch:integer; function  Getzn:integer;
public
  property ch: integer read Getch write Setch;
  property zn: integer read Getzn write Setzn;
  function Plus(x:Rat):Rat;   Procedure ReadR;
  function Minus(x:Rat):Rat;  Procedure WriteR;
  function Mult(x:Rat):Rat; function Divr(x:Rat):Rat;
  property myEvent: TNotifyEvent read FmyEvent
                                 write FmyEvent;
end;
implementation
 Procedure OnMyEvent(Sender: TObject);
```

```
    begin  writeln('We have 0! '); exit end;
  Procedure Simpl(var x:Rat); var i,n,m,p:integer;
  begin n:=x.ch; m:=x.zn;
   if abs(n)<abs(m) then p:=abs(n) else p:=abs(m);
   for i:=p downto 2 do
    if (n mod i=0) and (m mod i=0) then
      begin n:=n div i; m:=m div i end;
    if m<0 then begin n:=-n; m:=-m end;
    x.ch:=n; x.zn:=m;   end;
  procedure Rat.Setch; begin Fch:= Value end;
  procedure Rat.Setzn; begin Fzn:= Value end;
  function  Rat.Getch:integer; begin Getch:=Fch end;
  function  Rat.Getzn:integer; begin Getzn:=Fzn end;
  function Rat.Plus;  var t:Rat;
  begin t:=rat.Create;
    t.ch:=ch*x.zn+zn*x.ch;  t.zn:=zn*x.zn;
    if t.zn=0 Then begin OnmyEvent(t); exit end;
    simpl(t); Plus:=t;   end;
  function Rat.Minus;  var t:Rat;
  begin t:=rat.Create;
    t.ch:=ch*x.zn-zn*x.ch; t.zn:=zn*x.zn;
    if t.zn=0 Then begin OnmyEvent(t); exit end;
    simpl(t); Minus:=t;   end;
  function Rat.Mult;  var t:Rat;
  begin t:=rat.Create;
   t.ch:=ch*x.ch; t.zn:=zn*x.zn;
   if t.zn=0 Then begin OnmyEvent(t); exit end;
    simpl(t); Mult:=t  end;
  function Rat.DivR;  var t:Rat;
  begin t:=rat.Create;
   t.ch:=ch*x.zn; t.zn:=zn*x.ch;
   if t.zn=0 Then begin OnmyEvent(t); exit end;
    simpl(t); DivR:=t  end;
  Procedure Rat.ReadR;
   begin write('Enter ch: '); read(fch);
      write('Enter zn<>0: '); read(fzn); simpl(self)
   end;
  Procedure Rat.WriteR;
   begin write('Result=',ch:6); write('/',zn:6);
        readln    end;
end.
```

10.5.3. Властивості-масиви

На перший погляд, властивості-масиви відрізняються від простих властивостей приблизно так само, як змінні типу “масив” від змінних простих типів, тобто наявністю **індексу**. Однак спосіб опису і використання індексів у властивостях-масивах має досить великі розходження порівняно з індексацією звичайних масивів:

- головною особливістю є наявність **списку індексних параметрів**, який визначає індекси, що записуються подібно до формальних параметрів процедур із вказівкою типу. Однак список індексних параметрів беруть не у круглі, а у квадратні дужки;
- в описі специфікаторів доступу **`read`** і **`write`** можуть бути зазначені *тільки* ідентифікатори методів. Пряме використання ідентифікатора поля заборонено;
- метод, зазначений у специфікаторі читання, має бути функцією, у якої кількість, типи і порядок розміщення параметрів відповідають опису індексних параметрів властивості. Тип результату, що повертається функцією, має збігатися з типом властивості;
- метод, зазначений у специфікаторі запису, має бути процедурою, у якої кількість, типи і порядок розміщення початкових параметрів відповідають опису індексних параметрів властивості. Крім того, список параметрів процедури має мати один додатковий параметр такого ж типу, як у властивості, для передачі нового значення. Цей параметр може бути параметром-значенням чи параметром-константою;
- властивості-масиви не можуть бути опублікованими.

Наприклад, клас `TVect`, що моделює вектор дійсних чисел розмірності 5, має властивість-масив `Elements` зі списком індексних параметрів `[Index: Integer]`. Ця властивість надає доступ до будь-якого елемента конкретного вектора-об’єкта через його індекс. Повний опис класу `Tvect` див. у прикладі 10.8.

Зазначимо, що директива **`default`** у властивостях-масивах є не специфікатором збереження, а покажчиком того, яка з властивостей-масивів цього класу буде сприйматися за домовленістю. У розглянутому прикладі — це властивість `Elements`. До властивості `Elements` можливі два способи звертання:

- як до будь-якої іншої властивості за допомогою повної назви,

як-от: `MyVector. Elements[i]`, де `MyVector` – деякий вектор-об'єкт типу `Tvect`;

- як до властивості, прийнятої за домовленістю, з указівкою тільки назви об'єкта й індексу – `MyVector[i]`.

Також, як і масиви, властивості-масиви можуть бути *багатоіндексними*. Списки індексних параметрів таких властивостей і формальних параметрів їхніх методів доступу мають мати відповідне розмірності число індексів.

Приклад 10.8. Використавши клас `Tvect`, створити віконне застосування для введення значень двох векторів і обчислення їхнього скалярного добутку.

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes,
 Graphics, Controls, Forms, Dialogs, StdCtrls;
type
 TForm1 = class(TForm)
   Label1: TLabel; // Caption:= '1-ий вектор'
   Memo1: TMemo; //Відображення елементів 1-го вектора
   Label2: TLabel; // Caption:= '2-ий вектор'
   Memo2: TMemo; //Відображення елементів 2-го вектора
   Button1: TButton; //Уведення елементів 1-го вектора
   Button2: TButton; //Уведення елементів 2-го вектора
   Button3: TButton; //Обчислення скалярного добутку
   Label3: TLabel; // Caption:= 'Скалярний добуток'
   Edit1: TEdit; ; //Відображення скалярного добутку
   procedure FormCreate(Sender: TObject);
   procedure FormDestroy(Sender: TObject);
   procedure Button1Click(Sender: TObject);
   procedure Button2Click(Sender: TObject);
   procedure Button3Click(Sender: TObject);
  private { Private dec-ns } public { Public dec-ns }
 end;
 TVect=class private Fvector:Array[1..5] of real;
   function GetVect(Index: Integer): real;
   procedure SetVect(Index: Integer; Value: real);
  public property Elements[Index: Integer]: real
            read GetVect write SetVect; default;
 end;
```

```
var Form1: TForm1; v1,v2:TVect;
implementation  {$R *.dfm}
function TVect.GetVect(Index: Integer): real;
begin Result:=Fvector[Index]; end;
procedure TVect.SetVect(Index: Integer; Value: real);
begin Fvector[Index]:=Value; end;
procedure TForm1.FormCreate(Sender: TObject);
begin v1:=TVect.Create; v2:=TVect.Create;
  edit1.ReadOnly:=true; memo1.ReadOnly:=true;
  memo2.ReadOnly:=true;  end;
procedure TForm1.FormDestroy(Sender: TObject);
begin v1.Free; v2.Free end;
procedure TForm1.Button1Click(Sender: TObject);
var i:byte; ACaption,APrompt,ADefault,s:string;r:real;
begin if memo1.ReadOnly then memo1.ReadOnly:=false;
 Memo1.Clear; edit1.Clear;
 ACaption:='Елементи 1-го вектора'; ADefault:='0';
 for i:=1 to 5 do
  begin APrompt:='Введіть '+IntTostr(i)+' елемент';
   s:=InputBox(ACaption,APrompt,ADefault);
   try r:=StrToFloat(s);
    except on EConvertError do
     begin Showmessage('Вводьте числа');
           memo1.Clear; exit  end
   end; v1[i]:=r; Memo1.Lines.Add(s);
  end;  memo1.ReadOnly:=true;   end;
procedure TForm1.Button2Click(Sender: TObject);
var i:byte; ACaption,APrompt,ADefault,s:string; r:real;
begin If memo2.ReadOnly then memo2.ReadOnly:=false;
 Memo2.Clear; edit1.Clear;
 ACaption:='Елементи 2-го вектора'; ADefault:='0';
 for i:=1 to 5 do
  begin APrompt:='Введіть '+IntTostr(i)+' елемент';
   s:=InputBox(ACaption,APrompt,ADefault);
   try r:=StrToFloat(s);
    except on EConvertError do
     begin Showmessage('Вводьте числа');
        Memo2.Clear; exit  end
   end; v2[i]:=r; Memo2.Lines.Add(s);
  end; memo2.ReadOnly:=true;
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
var s:real;  i:byte;
begin s:=0; for i:=1 to 5 do s:=s+v1[i]*v2[i];
 If edit1.ReadOnly then edit1.ReadOnly:=false;
 edit1.Text:=floatToStr(s); edit1.ReadOnly:=true
end;
end.
```

Зауваження. Зверніть увагу, що у програмі всі звертання до елементів властивості-масиву виконуються так само, як і до елементів звичайного масиву.

? Запитання для самоперевірки

1. Дайте означення класу.
2. Що таке інкапсуляція?
3. Що таке наслідування?
4. Що таке поліморфізм?
5. Для чого використовуються поля класу?
6. Що таке методи класу?
7. Для чого використовується розділ `private`?
8. Для чого використовується розділ `public`?
9. Для чого використовується розділ `published`?
10. Дайте означення конструктора і деструктора.
11. Що таке віртуальний метод?
12. Що таке динамічний метод?
13. Що таке абстрактний метод?
14. Що таке перевизначений метод?
15. Дайте означення властивості.
16. Які особливості опису властивостей-масивів Ви знаєте?

Завдання для програмування

Завдання 10.1. Утворивши відповідний клас однонапрямленого списку, у режимі консолі запрограмувати розв’язання завдання 6.4 (стор. 133).

Завдання 10.2. Утворивши відповідний клас для заданого модуля, у режимі консолі запрограмувати розв’язання завдання 8.4 (стор. 171).

Завдання 10.3. Модифікувавши клас `TVect` (приклад 10.8), створити віконне застосування для розв’язання завдання 4.2 (стор. 87).

11. БІБЛІОТЕКА ВІЗУАЛЬНИХ КОМПОНЕНТІВ (VCL)

План викладу матеріалу

1. Базові класи компонентів
2. Клас `TComponent`
3. Клас `TControl`
4. Клас `TWinControl`
5. Класи – прямі нащадки класу `TPersistent`

Ключові терміни розділу

- *Компоненти і об'єкти*
- *Ієрархія класів VCL*
- Клас `TComponent`
- *Властивості класу* `TControl`
- *Елементи керування* (*ЕК*)
- *Клієнтська область ЕК*
- *Властивості класу* `TWinControl`
- Клас `TPersistent`
- Клас `TGraphicsObject`
- Клас `TObject`
- *Візуальні й невізуальні компоненти*
- *Клас* `TControl`
- *Події та методи класу* `TControl`
- *Батьківський та дочірній ЕК*
- *Параметр* `Sender`
- *Події та методи класу* `TWinControl`
- Клас `TCanvas`
- Клас `TStrings`

Основою створення проектів у Delphi 6 є *компоненти*, розміщені у спеціальній **бібліотеці візуальних компонентів** (**VCL,** Visual Component Library). З погляду програмування компоненти – це об'єкти (або екземпляри) певного класу, які мають *властивості* та *методи*, що між собою тісно пов'язані. Зміна властивості ініціює неявний виклик відповідного методу, а виконання методу часто змінює властивості компонента. Для кожного компонента спроектовано список *подій*, на які він реагує.

11.1. Базові класи компонентів

Бібліотека візуальних компонентів є добре продуманою. Вона максимально використовує можливості наслідування класів. Основу VCL утворюють класи, що описують компоненти. **Компоненти** – екземпляри класів, які є нащадками класу **`TComponent`**. Екземпляри інших класів називають **об'єктами (**наприклад, об'єкти класу `TFont`). **Об'єкти** виконують допоміжні та службові функції й не можуть розміщуватися на формі. Зауважимо, що не всі компоненти є візуальними, деякі з них є видимими тільки на етапі проек-

тування інтерфейсу користувача (наприклад, компонент `Timer`). Ієрархія класів є досить складна. У таблиці 11.1 показано базові класи та їхні похідні класи, які є на вершині ієрархії класів VCL.

Таблиця 11.1. Вершина ієрархії класів VCL

`TObject`	Базовий клас усіх об'єктів **VCL**. Усі класи є прямими або непрямими нащадками **`Tobject`**. Забезпечує створення, керування і знищення об'єктів
`Exception`	Базовий клас усіх виключень
`TPersistent`	Базовий клас усіх об'єктів **VCL**, що допускають операцію присвоєння або беруть участь в операціях з потоками. Об'єкти **`TPersistent`** створюватися не можуть. Клас використовується тільки для створення похідних класів
`TComponent`	Базовий клас **усіх компонентів**. Визначені властивості **`Name`** – назва компонента і **`Owner`** – власник компонента. Об'єкти типу **`TComponent`** не створюються. Клас **`TComponent`** використовується як **базовий**, коли оголошується **клас невізуального** компонента
`TControl`	Базовий клас **усіх візуальних компонентів**. Усі вони мають спільні властивості та методи, які визначають розміри компонента, його розміщення на формі, вирівнювання, видимість тощо. Визначено події, пов'язані з мишею
`TWinControl`	Базовий клас **усіх віконних компонентів**. Визначено властивості та методи, які задають вигляд вікна, фокусування, послідовність табуляції, властивість вікна як контейнера компонентів, процеси перетягування. Визначено події, пов'язані з клавіатурою
`TGraphicControl`	Базовий клас багатьох **невіконних компонентів**, включаючи графічні компоненти, швидкі кнопки. У ньому визначена властивість **Canvas** – полотно, яке є основою отримання зображень

Клас `TObject` є базовим класом для всіх класів компонентів. Зауважимо, що об'єктів цього класу не існує. Клас `TObject` забезпечує створення, керування і знищення об'єктів. Для цього

у класі визначені методи:

```
constructor Create;
```

– виділяє під об'єкт необхідну динамічну пам'ять. Ініціалізацію даних не здійснює, оскільки перевантажується у нащадках;

```
destructor Destroy; virtual;
```

– звільняє динамічну пам'ять, виділену під об'єкт;

```
procedure Free;
```

– звільняє динамічну пам'ять, виділену під об'єкт, якщо об'єкт дійсно було створено (здійснення попередньої перевірки перед звільненням пам'яті).

На наступному рівні ієрархії класів розміщені класи `Exception` і `TPersistent`. Клас `TPersistent` визначає здатність компонентів зберігати себе у файлах і пам'яті, а також бере на себе інші машинозалежні деталі.

Клас `TComponent` є більш прямим базовим класом для компонентів. Він забезпечує функціонування усіх основних компонентів. Невізуальні компоненти створені безпосередньо на основі `TComponent`.

Основою для візуальних компонентів є клас `TControl`, який походить від `TComponent`. Клас `TControl` забезпечує додаткові функції, які потрібні візуальним компонентам. Конкретні візуальні компоненти є нащадками від `TGraphicControl` або `TWinControl`.

Коли компонент розміщується на формі, Delphi 6 генерує для нього код класу на основі класу VCL, який йому відповідає. Водночас створюється покажчик на цей компонент (об'єкт). Для утворення назви класу і покажчика Delphi 6 використовує значення властивості `Name`.

Наприклад, розмістимо на формі компонент `TMemo`. У цей момент Delphi 6 створює новий клас від `TMemo`, а потім створює у пам'яті екземпляр цього класу (об'єкт `Memo1`) і покажчик на цей об'єкт. Для зображення нової форми Delphi 6 раніше вже створив новий клас на основі класу `TForm` і екземпляр цього класу.

Очевидно, що для початку роботи з VCL потрібно трохи ді-

знатися про властивості та методи класів. Побіжно неможливо оглянути усі класи VCL, тому розглянемо тільки деякі з них, які будуть найчастіше використовуватися.

11.2. Клас TComponent

Клас `TComponent` є вершиною в ієрархії компонентів. Від нього породжені всі компоненти разом зі самим застосуванням (клас `TApplication`) і формою (клас `TForm`). Невізуальні компоненти є прямими нащадками `TComponent`. У цьому класі визначено багато властивостей і методів, які є спільним для всіх компонентів. Розглянемо найбільш уживані з цих власивостей:

- **`Property`** `ComponentCount:integer;` – число компонентів, якими володіє цей компонент. Дорівнює кількості елементів у масиві `Components`. На 1 більше останнього компонента `ComponentIndex`, оскільки індекси відлічуються від 0. Може використовуватися разом з `ComponentIndex` у циклах, коли треба змінити якісь властивості всіх компонентів. Властивість доступна тільки під час виконання програми і тільки для читання;
- **`Property`** `ComponentIndex:integer;` – індекс компонента у масиві `Components` власника цього компонента. Відлічується від 0, тобто індекс першого компонента – 0. Властивість доступна тільки під час виконання програми і тільки для читання;
- **`Property`** `Components[Index:integer]: TComponent;` – масив компонентів, якими володіє цей компонент. Дає змогу послатися на будь-який компонент за допомогою його `ComponentIndex`. Властивість доступна тільки під час виконання програми і тільки для читання;
- **`Property`** `Name:` **`String`**; – назва компонента, за якою здійснюється посилання на нього з інших компонентів. Для компонентів задають стандартні назви: `Memo1, Memo2, Label1` тощо, які у великих проектах змінюються на мнемонічні назви;
- **`Property`** `Owner: TComponent;` – визначає власника цього компонента. Форма є власником усіх розміщених на ній компонентів. У свою чергу `Application` є власником усіх форм. Коли звільняється пам’ять, що була зайнята власником, автома-

тично звільняється пам'ять усіх компонентів, якими він володів. Властивість доступна тільки під час виконання програми і тільки для читання;

- **`Property`** `Tag: Longint;` – ця властивість не використовується системою. Програміст може використати її на свій розсуд.

У класі `TComponent` перевизначено конструктор:

```
constructor Create(AOwner: TComponent); virtual;
```

– створює об'єкт класу, розміщує посилання на себе у масиві `Components` свого власника, назва якого вказана у `AOwner`. Значення параметра `AOwner` присвоюється властивості `Owner` створеного компонента.

11.3. Клас TControl

11.3.1. Батьківський і дочірній компоненти

Клас **`TControl`** – базовий клас усіх візуальних компонентів. Усі вони мають спільні властивості та методи, які визначають розміри компонента, його розміщення на формі, вирівнювання, видимість тощо. Визначено основні події, пов'язані з мишею та клавіатурою. Для нащадків класу `TControl` вводять спільну назву – **елементи керування**.

У класі `TControl` вводиться поняття *батьківського* елемента керування (`parent controls`) і *дочірнього* елемента керування. Батьківським елементом може бути тільки **віконний** елемент керування, тобто нащадок `TWinControl`. Дочірній елемент керування можна розмістити прямо на формі або на деякому компоненті групування, наприклад, на панелі (клас `TPanel`). У першому випадку батьківським елементом керування буде форма, а у другому – елемент, що групує компоненти (панель). Зв'язані з батьківським, *дочірні елементи керування* можуть бути як віконними, так і невіконними. Кожний видимий елемент керування обов'язково має покажчик на свій батьківський елемент

```
property Parent:TWinControl;
```

Зв'язок *батьківський-дочірній* дає можливість дочірньому елементу керування використати окремі характеристики батьківського елемента. Дочірні елементи не можуть виходити за границі

батьківських. Якщо батьківський елемент переміщується по екрану, то разом з ним переміщуються і всі дочірні елементи. Для забезпечення зв'язку *батьківський-дочірній* використовують такі властивості класу `TControl`:

- **`Property`** `ParentBiDiMode:Boolean;` – при значенні `True` компонент використовує значення властивості `BiDiMode` свого батька. В іншому випадку використовується власна властивість `BiDiMode`;
- **`Property`** `ParentColor:Boolean;` – при значенні `True` компонент використовує значення властивості `Color` свого батька. Інакше використовується власна властивість `Color`;
- **`Property`** `ParentFont:Boolean;` – при значенні `True` компонент використовує значення властивості `Font` свого батька. В іншому випадку використовується власна властивість `Font`;
- **`Property`** `ParentShowHint:Boolean;`– при значенні `True` компонент використовує значення властивості `ShowHint` свого батька. В іншому випадку використовується власна властивість `ShowHint`.

Оскільки батьківський елемент керування має бути нащадком класу `TWinControl`, то саме у цьому класі визначено решту характеристик зв'язку *батьківський-дочірній*.

Зауважимо, що не треба плутати властивість `Owner`, визначену в класі `TComponent`, з поняттям батьківського елемента керування. Властивість `Owner` забезпечує створення та знищення підлеглих компонентів (компоненти можуть бути невидимими), а зв'язок *батьківський-дочірній* дає змогу керувати візуальним зображенням компонентів – місцем розміщення та перемальовування. У багатьох випадках компонент-власник (`Owner`) і компонент-батько (`Parent`) можуть збігатися.

Оскільки конструктор `Create` при створенні видимих **динамічних компонентів** (створюються на етапі виконання програми) не змінює властивості `Parent`, то цю властивість необхідно змінити програмно, наприклад:

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

```
var lbLabel: TLabel;
begin
 lbLabel:=TLabel.Create(Self); // виклик конструктора
 lbLabel.Parent:= Self; // зміна властивості Parent
 lbLabel.Caption:= 'Дочірній елемент';
end;
```

– параметр `Self` відображає конкретний об'єкт, що викликав метод (у нашому випадку – форма `Form1`). Зміна властивості `Parent` підключає позначку до списку дочірніх елементів форми і віконна функція форми звернеться до потрібного методу позначки, щоб заставити її з'явитися на екрані у момент появи самої форми.

11.3.2. Основні властивості класу `TControl`

Для відображення елементів керування на екрані використовують властивості *позиціонування*:

- **`Property`** `Left:integer;` – задає горизонтальну координату лівого верхнього кута елемента керування у пікселях щодо компонента-батька;
- **`Property`** `Top:integer;` – задає вертикальну координату лівого верхнього кута елемента керування у пікселях щодо компонента-батька;
- **`Property`** `Width:integer;` – задає ширину елемента керування в пікселях;
- **`Property`** `Height:integer;` – задає висоту елемента керування у пікселях.

Наведемо інші **важливі властивості** класу `TControl`:

- **`Property`** `Align: TAlign;` де **`Type`** `TAlign = (alNone, alTop, alBottom, alLeft, alRight, alClient);` – визначає **спосіб вирівнювання** компонента у контейнері (батьківському компоненті):
  - `alTop/alBottom` здійснюється вирівнювання компонента за верхнім/нижнім краєм контейнера (розтягнення компонента на всю ширину контейнера без зміни його висоти);

- `alLeft/alRight` здійснюється вирівнювання компонента за лівим/правим краєм контейнера (розтягнення компонента на всю висоту контейнера без зміни його ширини);
- при `alClient` – елемент керування розширюється до всієї клієнтської області батьківського компонента;
- `alNone` – вирівнювання не відбувається (за домовленістю).

Клієнтська область компонента – це доступна користувачеві частина компонента. Саме у клієнтській частині користувач уводить значення (рядки сиволів) чи малює зображення. У клієнтську область не входить *обрамлення* компонента (заголовок, меню, рамка, статусний рядок тощо);

- **Property** `Anchors: TAnchors;` де **Type** `TAnchors = (akLeft, akTop, akRight, akBottom)` – визначає **прив'язку** цього компонента до меж батьківського зі зміною розмірів останнього.

  Якщо властивість `Anchors` містить значення прив'язаності до протилежних сторін батьківського компонента, наприклад, `[akLeft, akRight]`, то зі зміною розмірів батьківського компонента відбувається розтягування або стиснення цього компонента, оскільки відстані від сторін батьківського компонента зберігаються. При стисненні може відбутися повне знищення зображення цього елемента керування;
- **Property** `AutoSize: Boolean;` – визначає, чи будуть розміри компонента автоматично пристосовуватися до розмірів його вмісту (наприклад, під висоту символів тексту);
- **Property** `BiDiMode: TBiDiMode;` де
  Type `TbiDiMode = (bdLeftToRight, bdRightToLeft, bdRightToLeftNoAlign, bdRightToLeftReadingOnly);` – задає спосіб поведінки об'єктів з урахуванням національної специфіки. За домовленістю рівне `bdLeftToRight`, що забезпечує введення тексту і прокручування інформації зліва направо і характерне для європейських мов. Інші значення використовують для країн Сходу;
- **Property** `Caption:` **String**; – визначає рядок тексту, який ідентифікує компонент для користувача. Зазвичай це напис на позначці, кнопці й інших компонентах;

- **Property** ClientHeight: integer; – визначає висоту клієнтської області у пікселях. Переважно збігається з Height. Однак для деяких компонентів, наприклад, форми, у клієнтську область не входить її обрамлення: заголовок, меню, рамка тощо;
- **Property** ClientWidth: integer; – визначає ширину клієнтської області у пікселях;
- **Property** Color: TColor;

  де **Type** TColor=$00000000..$02FFFFFF; – визначає колір фону елемента керування. Значення кольору може бути задано як чотирибайтове шістнадцяткове число, яке визначає палітру та інтенсивність червоного, зеленого і синього кольорів у форматі RGB. Для найбільш уживаних кольорів уведено константи, наприклад: clBlack, clGreen, clOlive, clGray і т.д.;
- **Property** Cursor: TCursor; – визначає вигляд курсора миші, під час потрапляння його в область компонента;
- **Property** DragCursor: TCursor; – визначає вигляд курсора миші, під час потрапляння його в область компонента у процесі перетягування;
- **Property** Enabled:Boolean; – визначає, чи реагує компонент на події, зв'язані з мишею, клавіатурою і таймером;
- **Property** Font:TFont; – властивість класового типу TFont, яка визначає характеристики шрифту, що використовується для відображення текстової інформації;
- **Property** Hint:String; – містить текст, що відображається у вікні підказки або в рядку статусу;
- **Property** ShowHint:Boolean; – дає змогу або забороняє показувати вікно підказки;
- **Property** Text: **String**; – містить текст, що асоціюється з цим елементом керування, наприклад, – текст, який є у рядку введення Edit;
- **Property** Visible:Boolean; – визначає, чи відобразиться елемент керування на екрані.

11.3.3. Основні події, описані в класі `TControl`

Операційною системою Windows керують події, які виникають унаслідок маніпуляцій з мишею та натискання клавіш на клавіатурі. Прикладні програми увесь час запитують Windows про виникнення подій. Windows повідомляє програму про подію, посилаючи відповідне *повідомлення*. Кількість таких повідомлень приблизно дорівнює 175.

Кожний компонент спроектовано так, щоб він реагував на відповідні події. Як правило, це події Windows, але не обов'язково. Так, компонент `Button` (кнопка) буде реагувати на клацання мишею (подія Windows). Але невізуальні компоненти (наприклад, компоненти доступу до баз даних) можуть реагувати на події, які не належать до Windows, наприклад, досягнення користувачем кінця таблиці.

Реакцію користувача на виникнення події програмують за допомогою *процедур опрацювання події* (`event handlers`). Список подій, на які реагує компонент, наведено на закладці `Events` вікна інспектора об'єктів. У програмі немає потреби опрацьовувати усі події, на які реагує компонент. Якщо подія не опрацьована, то відповідне повідомлення або відкидається без реагування, або виконуються дії за домовленістю. У класі `TControl` описані події, які виникають при користуванні мишею.

Події, які виникають у разі натискання лівої кнопки миші:

```
type TNotifyEvent=procedure(Sender:TObject)of object;
property OnClick:TNotifyEvent;
```

Подія виникає при виборі елемента керування мишею (тобто була натиснута і відпущена ліва клавіша миші, коли курсор миші знаходився над елементом керування). Параметр **`Sender`** – покажчик на компонент, в якому відбулася подія.

```
property OnDblClick:TNotifyEvent;
```

Подія виникає під час подвійного натискання лівої клавіші миші на елементі керування.

Загальні події, які виникають при маніпулюванні мишею:

```
type TMouseEvent = procedure(Sender:TObject; Button:
TMouseButton; Shift:TShiftState;X,Y:integer)of object;
```

property OnMouseDown:TMouseEvent;

Подія виникає при натисканні клавіші миші, коли курсор миші знаходиться над елементом керування. Значення параметра **Button** визначає, яка кнопка миші натиснута (mbLeft –ліва; mbRight – права; mbMiddle – середня). Параметр **Shift** є множиною, яка містить значення натиснутої кнопки миші або натиснутих одночасно допоміжних клавіш Shift, Alt чи Ctrl (ssShift – натиснута клавіша Shift, ssAlt – натиснута клавіша Alt, ssCtrl – натиснута клавіша Ctrl, ssLeft – ліва кнопка миші, ssRight – права кнопка миші, ssMiddle – середня кнопка миші, ssDouble – подвійне натиснення кнопки миші). Параметри X і Y визначають координати курсора миші в клієнтській області компонента. Параметр **Sender** – покажчик на компонент, в якому відбулася подія.

type TmouseEvent = **procedure** (Sender:TObject; Shift: TShiftState; X,Y:integer) **of** object;

property OnMouseMove:TMouseMoveEvent;

Подія виникає при переміщенні миші, коли її курсор знаходиться над елементом керування, і водночас натиснута клавіша миші. Параметри мають те ж значення, що й для події OnMouse-Down.

property On MouseUp:TMouseEvent;

Подія виникає при відпусканні клавіші миші, коли курсор знаходиться над елементом керування.

Події, які призначені для підтримки перетягування:

type TdragDropEvent = **procedure**(Sender,Source: TObject; X,Y:integer) **of** object;

property OnDragDrop:TDropEvent;

Подія виникає, коли елемент керування **Source** опускається на елемент **Sender**. Параметри X, Y – глобальні координати курсора миші під час опускання елемента керування Source.

type TDragOverEvent = **procedure** (Sender, Source: TObject;X, Y:integer; State: TDragState; **var** Accept: boolean) **of** object;

property OnDragOver:TDragOverEvent;

Подія виникає, коли елемент керування **Source**, що переміщується, знаходиться над елементом **Sender**. X, Y – глобальні координати курсора миші. Параметр **State** типу TDragState визначає стан об'єкта, що переміщується, щодо інших об'єктів. Можливі такі стани: dsDragEnter – курсор миші входить у межі компонента; DsDragMove – курсор миші переміщується у межах компонента; DsDragLeave – курсор миші виходить за межі компонента. Параметр **Accept** має набути значення True в процедурі опрацювання події, якщо елемент керування **Sender** є компонентом, що перетягується і False – у протилежному випадку.

```
type TendDragEvent = Procedure(Sender, Target:TObject;
X, Y: integer) of objerct;
property OnEndDarg: TEndDragEvent;
```

Подія виникає, коли закінчується перетягування елемента керування Sender над елементом Target. X, Y – глобальні координати курсора миші на час закінчення перетягування.

```
type TStartDragEvent=procedure(Sender: TObject; var
DragObject: TDragObject) of object;
property OnStartDrag:TStartDragEvent;
```

Подія виникає, коли починається перетягування елемента **Sender**. Параметр **DragObject** містить покажчик на об'єкт, що використовується для формування зображення елемента керування під час перетягування. За домовленістю дорівнює nil, тобто це означає, що переноситися буде сам компонент.

11.3.4. Деякі методи класу TControl

Існує низка методів, які дають змогу змінювати зовнішній вигляд елемента керування на етапі виконання програми:

```
procedure BringToFront;
```

Розміщує елемент керування над іншими елементами у батьківському елементі керування.

```
procedure Hide;
```

Робить поточний елемент керування і всі його дочірні елементи **невидимими**. Властивість Visible набуває значення false.

procedure `Refresh;`

Витирає, а потім заново малює елемент керування.

procedure `SendToBack;`

Розміщує елемент керування нижче від усіх інших елементів у батьківському елементі.

procedure `Show;`

Робить поточний елемент керування видимим. Видимими стають також всі його дочірні елементи, якщо властивість `Visible` у них має значення `true`.

У класі `TControl` розроблено методи, які дають можливість перетворити локальні координати елемента керування, що визначають його розміщення у батьківському елементі, у глобальні координати, які визначають його розміщення на екрані дисплея чи навпаки:

type `TPoint=`**record** `x:longint; y:longint;` **end;**

function `ClientToScreen(`**const** `Point:TPoint):TPoint;`

Повертає глобальні координати точки, яка задана локальними координатами, що містяться у параметрі `Point`. `X` – горизонтальна координата у пікселях, `Y` – вертикальна координата у пікселях.

function `ScreenToClient(`**const** `Point:TPoint):TPoint;`

Повертає локальні координати точки, заданої глобальними координатами, що містяться в параметрі `Point`.

11.3.5. Приклад опрацювання однієї події одночасно для декількох компонентів

Кожна процедура опрацювання події, яка виникає при користуванні мишею, буде мати принаймні один аргумент `Sender`. Залежно від події, процедура опрацювання події може також набувати один або декілька додаткових аргументів.

Що ж являє собою `Sender`? Це покажчик на компонент, який посилає повідомлення процедурі опрацювання події. У випадку процедури опрацювання події для конкретного компонента аргумент `Sender`, у принципі, є зайвим (однак *опускати його не можна*). Взагалі, `Sender` дає змогу мати процедуру опрацювання

однієї події одночасно для декількох компонентів.

Приклад 11.1. Для ілюстрації створимо такий проект:

нехай у вікні форми буде розташований компонент Memo з деяким текстом і три кнопки з підписами “**Приховати**”, “**Показати**” і “**Закрити**”. Після натиснення цих кнопок відповідно буде приховуватися чи показуватися компонент Memo, закриватися вікно форми (закінчуватися робота застосування).

Етап проектування інтерфейсу проекту:

```
Form1: Button1; Button2; Button3; Memo1
Button1.Caption := 'Приховати',
Button2.Caption := 'Показати'
Button3.Caption := 'Закрити'
Button1.Name := 'Hide' // Вводимо осмислені назви
Button2.Name := 'Show', Button3.Name := 'ButClose'
Memo1.Lines := 'Цей компонент можна приховати/показати'
Form1.Caption := 'Демонстрування можливостей Sender'
```

Етап програмування *процедур опрацювання подій*.

Оскільки ми хочемо записати одну **спільну процедуру** – опрацювати подію натиснення кнопки одночасно для трьох різних кнопок, то спочатку запрограмуємо таку **спільну процедуру** для однієї кнопки (наприклад, для кнопки Hide). Назвемо цю процедуру ButClick. Для інших двох кнопок виберемо з випадаючого списку назву цієї спільної процедури ButClick (у рядку події OnClick). Реалізація процедури опрацювання події натиснення кнопок:

```
procedure TForm1.ButClick(Sender: TObject);
begin
 if Sender = Hide then Memo1.Hide()
  else
   if Sender = Show then Memo1.Show()
                    else Form1.Close
end;
```

Відкомпілюємо і запустимо програму на виконання. Клацнемо кожною кнопкою і переконаємося, що вони працюють, як пе-

редбачалося. Отже, ми створили одну **спільну процедуру** опрацювання події натиснення кнопки `OnClick` для трьох кнопок. Для цього було використано параметр `Sender`, щоб визначити, яка кнопка послала повідомлення про подію `OnClick`, а потім запрограмовано приховування/показ компонента `Memo` чи закриття застосування. Можна було б створити окремі процедури опрацювання події для кожної кнопки, але тоді код став би менш компактним.

Зауважимо, що після створення процедури опрацювання події `OnClick` для одного з компонентів, можна зв'язати цю процедуру з подією `OnClick` будь-якого іншого компонента форми.

11.4. Клас `TWinControl`

11.4.1. Основні властивості класу `TWinControl`

Клас `TWinControl` є нащадком класу `TControl` і використовується як базовий для створення **віконних** елементів керування. З кожним віконним елементом керування пов'язане вікно Windows, що визначається спеціальним числовим ідентифікатором – **дескриптором** вікна. Наявність вікна дає змогу активізувати елемент керування під час виконання програми. Це означає, що такі елементи під час виконання програми можуть отримувати фокус уведення і реагувати на події, які виникають при використанні клавіатури.

Характерними представниками елементів `TWinControl` є однорядковий редактор `Edit`, багаторядковий редактор `Memo`, список `ListBox`, кнопка `Button` і т.д. Нагадаємо, що тільки віконні елементи керування можуть містити інші компоненти, тобто бути батьками. У класі `TWinControl` завершується визначення характеристик зв'язку *батьківський-дочірній*.

Властивості, які підтримують зв'язок батьківський-дочірній:

- **`property`** `ControlCount:Integer;` – містить певну кількість дочірніх компонентів для цього компонента. Властивість доступна тільки під час виконання програми і тільки для читання;
- **`property`** `Controls[Index:Integer]:TControl;`– містить список дочірніх елементів, якщо такі є. Нумерація починається

з нуля. Властивість доступна тільки під час виконання програми і тільки для читання;

- **property** `ParentCTl3D: Boolean;` – якщо значення цієї властивості дорівнює `true`, то використовується значення властивості `Ctl3D` батьківського компонента, а інакше власне значення властивості `Ctl3D`.

Властивості, які формують зображення у стилі Windows:

- **property** `Brush: TBrush`; – задає *пензлик*, за допомогою якого малюється фон віконного елемента. Властивість доступна тільки під час виконання програми і тільки для читання;
- **property** `CTl3D: Boolean`; – визначає зовнішній вигляд елемента. Якщо значення властивості дорівнює `true` (за домовленістю), то елемент виглядає об'ємно, а інакше – плоско;
- **property** `Handle: HWND;` – визначає числовий ідентифікатор (дескриптор) вікна керування, за яким Windows може звертатися до цього вікна. Властивість доступна тільки під час виконання програми і тільки для читання;
- **property** `Showing: Boolean;` – якщо значення цієї властивості дорівнює `true`, то компонент видно на екрані, а інакше – ні. Значення `true` властивості `Showing` еквівалентно тому, що його властивість `Visible` і всіх його батьків дорівнює `True`. Якщо ж властивість `Visible` компонента або декого з його батьків дорівнює `false`, то `Showing` дорівнює `false`.

11.4.2. Деякі методи класу `TWinControl`

Наведемо деякі найвживаніші методи класу `TWinControl`:

function `ContainsControl(Control: TControl): Boolean;`

Повертає `true`, якщо поточний елемент керування є батьківським щодо до компонента `Control`.

procedure `InsertControl(AControl: TControl);`

Розміщує компонент `Acontrol` у властивість-масив `Controls` цього віконного елемента.

procedure `RemoweControl(AControl: TControl);`

Видаляє компонент `AControl` з властивості-масиву `Controls` цього віконного елемента.

procedure `DisableAlign;`

Тимчасово забороняє вирівнювання дочірніх компонентів цього віконного елемента. Метод застосовують одночасно з методом `EnableAlign`, що скасовує вплив `DisableAlign`.

function `CanFocus: Boolean; dynamic;`

Повертає значення `True`, якщо віконний елемент може отримати фокус. Для того, щоб елемент міг отримати фокус, у нього й у його батьків властивості `Enabled` і `Visible` мають містити `True`.

procedure `EnableAlign;`

Скасовує дію викликаного раніше методу `DisableAlign` і викликає метод `ReAlign` для вирівнювання дочірніх компонентів.

function `Focused: Boolean; dynamic;`

Повертає `True`, якщо віконний елемент отримав фокус уведення.

procedure `ReAlign;`

Вирівнює дочірні компоненти у віконному елементі.

procedure `SetFocus; virtual;`

Передає фокус уведення віконному елементові.

11.4.3. Основні події, описані в класі `TWinControl`

У класі `TWinControl` описано події, які виникають під час користування *клавіатурою*:

```
Type TKeyEvent = procedure(Sender:TObject;
         var Key: Word; Shift:TShiftState)of object;
```

property `OnKeyDown:TKeyEvent;`

Подія відбувається у момент початку натискання клавіші. Параметр **`Sender`** – покажчик на компонент, в якому відбулася подія. Параметр **`Key`** – **код** натиснутої **клавіші**. Параметр **`Shift`** є множиною, яка містить значення натиснутих одночасно з цією клавішею допоміжних клавіш `Shift`, `Alt` чи `Ctrl` (`ssShift` – натиснута клавіша `Shift`, `ssAlt` – натиснута клавіша `Alt`, `ssCtrl` – натиснута клавіша `Ctrl`).

property `OnKeyUp:TKeyEvent;`

Подія відбувається, коли на клавіатурі відпускають клавішу.

```
Type TKeyPressEvent = procedure(Sender: TObject;
                                var Key: Char)of object;
property OnKeyPress: TKeyPressEvent;
```

Подія відбувається при натисканні символьної клавіші. Параметр **Sender** – покажчик на компонент, в якому відбулася подія. Параметр **Key** – **ANSI-код** натиснутої **клавіші**.

```
property OnEnter:TNotifyEvent;
```

Подія виникає, коли віконний елемент отримує фокус уведення.

```
property OnExit:TNotifyEvent;
```

Подія виникає, коли віконний елемент втрачає фокус уведення.

11.4.4. Приклади опрацювання подій натискання клавіш

Події OnKeyPress, OnKeyUp і OnKeyDown пов'язані з клавіатурою і їх викликають для *активного* елемента керування (елемента, що має фокус уведення) при натисканні клавіші.

Подія OnKeyPress виникає, коли натиснута *символьна* клавіша. Після цього повертається **ANSI-код** натиснутої клавіші. Водночас не перехоплюються спеціальні клавіші, такі як [PrintScreen] чи [Alt], а тільки [Enter], [Esc] і [Backspace]. Параметр **Key** є параметром-змінною, тобто його можна змінювати у процедурі опрацювання події. Це використовують, наприклад, для **фільтрування символів**, що вводяться користувачем, – якщо символ неприпустимий, то, встановивши значення **Key** рівним нулю, запобігають його передачі для дальшого опрацювання (відображення на екрані тощо).

Події OnKeyDown і OnKeyUp викликаються при натисканні (OnKeyDown) чи відпусканні (OnKeyUp) клавіші. Вони відбуваються навіть при натисканні спеціальних клавіш керування, наприклад, функціональних клавіш. Разом з тим передаються два параметри: **Key** і **Shift**. Параметр **Key** має тип Word і містить клавіатурний код (а не ANSI) натиснутої клавіші, а параметр **Shift** інформує про стан клавіш [Shift], [Ctrl] і [Alt].

Фокус уведення – одне з найважливіших понять при звертанні до елементів керування у Windows, адже керування одержує *активний* елемент (елемент, що має фокус). Якщо елемент одержує

фокус, то це належним чином відображається на екрані – текстове поле отримує мерехтливий маркер уведення, а кнопка виділяється пунктирною рамкою навколо напису. Delphi 6 дає змогу обробляти дві події, пов'язані з передачею фокуса: `OnEnter` і `OnExit`. Якщо перейти від одного елемента керування до іншого, то для попереднього елемента викликається подія `OnExit`, а для нового – `OnEnter`. Передача фокуса елементу керування здійснюється за допомогою методу `SetFocus`.

Події, пов'язані з клавіатурою, викликають для активного елемента керування. Якщо властивість форми `KeyPreview` дорівнює `True`, то подія, пов'язана з клавіатурою, передається спочатку формі, а потім поточному елементу керування. Отже, форма може попередньо аналізувати повідомлення. *Цей факт будемо широко використовувати, розробляючи наші проекти*.

Приклад 11.2. **Аналіз натискання клавіш**. Користувач натискає довільну клавішу, а програма видає клавіатурний код клавіші, код ANSI і відповідний символ (при його наявності), а також фіксує натискання однієї чи декількох клавіш керування.

Етап проектування інтерфейсу проекту:

```
Form1: Label1; Label2; Label3; Label4
       Edit1; Edit2; Edit3; Edit4
Form1.Caption:= "Аналіз натискання клавіш"
Form1.KeyPreview:= True // аналіз на рівні форми
Edit1.ReadOnly:= True; Edit2.ReadOnly:= True
Edit3.ReadOnly:= True; Edit4.ReadOnly:= True
Label1.Caption:= 'Код ANSI'
Label2.Caption:= 'Клавіатурний код'
Label3.Caption:= 'Клавіші керування'
Label4.Caption:= 'Символ"
```

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.FormKeyDown(Sender: TObject;
                var Key: Word; Shift: TShiftState);
var msg: string;
begin msg:='';
 edit2.Text:=IntToStr(Key);
 if ssAlt in Shift then msg:='Alt+' + msg;
```

```
 if ssCtrl in Shift then msg:='Ctrl+' + msg;
 if ssShift in Shift then msg:='Shift+' + msg;
 if length(msg)>0 then delete(msg, length(msg), 1);
 edit3.Text:=msg;
end;
procedure TForm1.FormKeyPress(Sender: TObject;
                              var Key: Char);
begin  Edit1.Text:= IntToStr(ord(Key));
       Edit4.Text:=Key;
end;
procedure TForm1.FormKeyUp(Sender: TObject;
            var Key: Word;  Shift: TShiftState);
begin edit1.Text:=''; edit2.Text:='';
      edit3.Text:=''; edit4.Text:='';
end;
```

У проекті властивість форми `KeyPreview` дорівнює `True`, тому події, пов'язані з клавіатурою, передаються формі, а форма попередньо їх аналізує і перетворює.

Зауважимо, що програма дає можливість знайти клавіатурний код чи код ANSI довільної клавіші, а тому може бути корисною при конструюванні проектів, у яких відбувається складний аналіз натискання клавіш.

Приклад 11.3. **Знаходженння дійсних коренів квадратного рівняння.** У програмі будемо аналізувати коди клавіш, які натискатиме користувач при введенні числових даних для коефіцієнтів рівняння. **Символи**, *які не сумісні з представленням чисел*, **будемо блокувати**. У проекті властивість форми `KeyPreview` дорівнює `True`, тому події, пов'язані з клавіатурою, передаються формі, а форма попередньо аналізує дані.

Етап проектування інтерфейсу проекту:

```
Form1: Label1; Label2; Label3; Label4
     Edit1; Edit2; Edit3;  Button1; Button2; Button3
Form1.Caption:= ' Квадратні рівняння '
Form1.KeyPreview:= True // аналіз на рівні форми
Button1.Caption:= 'Корені'
Button1.Caption:= 'Нові значення'
Button3.Caption = 'Вийти'
Edit1.Text:= ''; Edit2.Text:= ''; Edit3.Text:= ''
```

```
Label2.Caption:= 'Коефіцієнт A (<>0) ' // до Edit1
Label3.Caption:= 'Коефіцієнт B'   // до Edit2
Label4.Caption:= 'Коефіцієнт C'   // до Edit3
Label1.Caption:= '' // результат у Label1. Caption
Label1.WordWrap:= True; Label1.AutoSize:= True
```

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.EditKeyPress(Sender: TObject;
                                        var Key: Char);
var buf:string[20];
begin
 if Sender = Edit1
  then buf:=Edit1.Text
  else if Sender = Edit2 then buf:=Edit2.Text
                         else buf:=Edit3.Text;
 case key of
  '0'.. '9', chr (8): ; // допустимі символи
  // Символ ',' має бути один!
  ',': if pos(',',buf)<>0 then key:=chr(0);
  // Символ '-' має бути першим!
  '-': if length(buf)<>0 then key:=chr(0);
  // Завершення введення. Перехід на наступний елемент
  chr(13): if Sender = Edit1
                   then Edit2.SetFocus
                   else if Sender = Edit2
                           then Edit3.SetFocus
                           else button1.SetFocus;
  else key:=Chr(0) // блокування недопустимих символів
 end
end;
procedure TForm1.Button2Click(Sender: TObject);
begin Edit1.text:='';  Edit2.text:='';
      Edit3.text:='';  Label1.caption:='';
      Edit1.SetFocus
end;
procedure TForm1.Button3Click(Sender: TObject);
begin Form1.Close end;
procedure TForm1.Button1Click(Sender: TObject);
var a,b,c:real; // Коефіцієнти рівняння
    d:real; {Дискримінант) x1,x2:real; // Корені рівняння
begin
```

```
 if (Edit1.Text='')or(Edit2.Text='')or(Edit3.Text='')
  then begin
       MessageDlg('Задайте всі коеф.!', mtWarning,[mbOk],0);
       exit     end;
 a:=StrToFloat(Edit1.text);
 b:=StrToFloat(Edit2.text);
 c:=StrToFloat(Edit3.text);
 if a=0
  then begin Label1.font.color:=clRed;
    Label1.caption:= 'Коефіцієнт a=0!";   end
  else
   begin // Обчислення коренів
    d:=Sqr(b)-4*a*c;
    if d<0
     then begin Label1.font.color:=clRed;
       Label1.caption:=' Дійсних коренів нема!' end
     else begin
       x1:=(b+Sqrt(d))/(2*a); x2:=(-b+Sqrt(d))/(2*a);
       Label1.font.color:=clBlack;
       Label1.caption:=' Дійсні корені: '
       + #13 + 'x1=' + FloatToStrF(x1,ffGeneral,7,4)
       + #13 + 'x2=' + FloatToStrF(x2,ffGeneral,7,4);
          end;
   end;
end;
```

Отже, процедура `TForm1.EditKeyPress` має *універсальне значення* і буде використовуватися нами у всіх випадках уведення чисел.

11.5. Класи – прямі нащадки класу TPersistent

Клас **`TPersistent`** – базовий клас усіх об'єктів VCL, що допускають операцію присвоєння або беруть участь своїми властивостями в операціях з потоками. Клас `TPersistent` – абстрактний клас, а тому об'єкти `TPersistent` створюватися не можуть. Клас використовується тільки для створення похідних класів. Безпосередніми нащадками класу **`TPersistent`** є класи:

`TComponent` – вершина ієрархії компонентів (див. 11.2);

`TCanvas` – визначає поверхню для малювання;

TClipboard	– системний буфер обміну;
TGraphic	– абстрактний базовий клас об'єктів, які зберігають і відображають візуальні зображення (породжує класи `TBitmap`, `TIcon`, `TJPEGImage`, `TMetafile`, `TOleGraphic`);
TGraphicsObject	– абстрактний базовий клас графічних об'єктів (породжує класи: пензлика – `TBrush`, шрифту – `TFont` і олівця – `TPen`);
TStrings	– абстрактний базовий клас об'єктів, які інкапсулюють списки рядків.

11.5.1. Клас `TCanvas`

Клас **`TCanvas`** визначає поверхню для малювання (канву) – об'єкт `Canvas`, який не є компонентом, і, використовують як **властивість** різних елементів керування.

Канва складається з окремих точок – *пікселів*. Кожний піксель має горизонтальну і вертикальну координату. Початок координат, тобто точка з координатами (`0`, `0`) розміщена у верхньому лівому куті канви. Горизонтальна вісь напрямлена зліва направо, а вертикальна – зверху вниз. Розмір канви залежить від розміру і особливостей компонента. Так, наприклад, для компонента `Image`, розмір канви визначається властивостями `Height` і `Width`, а для форми – властивостями `ClientHeight` і `ClientWidth`.

На канві розміщений графічний курсор, який визначає поточне розміщення олівця. Малювання простих фігур – ліній, кіл, прямокутників тощо – починається з поточного розміщення цього курсора. У процесі малювання розміщення курсора змінюється. Розглянемо основні властивості `TCanvas`:

- **`property`** `Brush:TBrush;` – задає пензлик канви;
- **`property`** `Font:TFont;` – задає шрифт канви;
- **`property`** `Pen:TPen;` – задає олівець канви;
- **`property`** `PenPos:TPoint;` – визначає поточне положення графічного курсора. Тип `TPoint` визначають так:
  `type` `TPoint =` **`packedrecord`** `X: Longint; Y: Longint;` **`end;`**

- **property** `Pixels:[X,Y:integer]:TColor;` – задає колір пікселя канви; `X` і `Y` – координати пікселя.

В класі `TCanvas` визначені методи, які використовують для побудови **геометричних фігур** – *контурних* (не мають внутрішнього зафарбування) і *зафарбованих* (мають внутрішнє зафарбування). При малюванні контурних фігур використовується тільки *олівець* (`Pen`), а зафарбованих (наприклад, еліпс, многокутник) – пензлик (`Brush`). Тексти зображують відповідно до властивостей, заданих у шрифті `Font`. На етапі виконання програми значення властивостей `Pen`, `Brush` і `Font` можна змінювати. Ці зміни мають бути виконані до початку процесу малювання.

Основні методи створення простих графічних зображень:

procedure `Arc(x1,y1,x2,y2,x3,y3,x4,y4:integer);`

Малює дугу еліпса, який вписаний у прямокутник з координатами лівого верхнього кута (`x1`, `y1`) і правого нижнього кута (`x2`, `y2`). Дуга починається у точці перетину еліпса з прямою, що проходить через центр еліпса і точку (`x3`, `y3`), а закінчується в точці перетину еліпса з прямою, що проходить через центр еліпса і точку (`x4`, `y4`).

procedure `Ellipse(x1,y1,x2,y2:integer);` **overload;**

Малює еліпс, який вписаний в прямокутник з координатами лівого верхнього кута (`x1`, `y1`) і правого нижнього кута (`x2`, `y2`).

type `TFillStyle=(fsSurface,fsBorder);`

procedure `FloodFill(x,y:integer; Color:TColor; Fill Style:TFillStyle);`

Зафарбовує пензлем `Brush` поверхню канви, починаючи з точки (`x`, `y`). Зафарбовує поверхню до границі, яка задана кольором `Color`, якщо параметр `FillStyle` має значення `fsBorder`, або зафарбовується частина поверхні, яка має колір `Color`, якщо параметр `FillStyle` має значення `fsSurface`.

procedure `LineTo(x,y:integer);`

Проводить лінію олівцем `Pen` від поточного положення графічного курсора до точки з координатами (`x`, `y`). Властивість `PenPos` отримує нове значення, визначене координатами (`x`, `y`).

procedure `MoveTo(x,y:integer);`

Переводить графічний курсор у точку з координатами (`x`, `y`).

procedure `Pie(x1,y1,x2,y2,x3,y3,x4,y4:longint);`

Малює сектор еліпса. Дуга еліпса задається так само, як і в методі `Arc`.

procedure `Rectangle(x1,y1,x2,y2:integer);` **overload;**

Малює прямокутник, у якого лівий верхній кут має координати (`x1`, `y1`), а правий нижній кут – координати (`x2`, `y2`).

procedure `RoundRect(x1,y1,x2,y2,x3,y3:integer);`

Малює прямокутник, в якого верхній лівий кут має координати (`x1`, `y1`), правий нижній кут – координати (`x2`, `y2`).Кути прямокутника заокруглені частинами еліпса з осями `x3` і `y3`.

procedure `TextOut(x,y:integer;` **const** `Text:`**string**`);`

Виводить текст `Text` на поверхню канви таким чином, що координати (`x`, `y`) є координатами лівого верхнього кута прямокутника, в якому виводиться текст. Параметри тексту задаються характеристиками шрифту `Font`, колір фону – поточним кольором пензля `Brush`.

function `TextHeight(`**const** `Text:`**string**`):integer;`
function `TextWidth(`**const** `Text:`**string**`):integer;`

Повертають відповідно висоту і довжину тексту `Text`, який визначається шрифтом `Font`.

У класі `TCanvas` визначено **події**:

property OnChange:TNotifyEvent;

Виникає перед тим, як у канві мають відбутися зміни.

property OnChanging:TNotifyEvent;

Виникає зразу ж після того, як у канві відбулися зміни.

11.5.2. Клас `TGraphicsObject`

Delphi має декілька класів загального призначення, які дають змогу виводити графічні зображення на поверхню компонентів. Це класи: `TFont` (шрифт), `TPen` (олівець) і `TBrush` (пензель), які є нащадками класу `TGraphicsObject`.

Клас `TFont` визначає властивості шрифту у будь-якому компонентові, що може містити деякий текст. Наведемо деякі властивості цього класу:

- **type** `TFontCharset=0..255;` **property** `Charset:TFontCharset` **nodefault;** – визначає набір символів шрифту;
- **property** `Color:TColor;` – визначає колір символів;
- **property** `Height:integer;` – визначає висоту шрифту у пікселях;
- **type** `TFontName=type` **string; property** `Name: TFontName;` – визначає назву шрифту;
- **type** `TFontPitch=(fpDefault,fpVariable,fpFixed);` **property** `Pitch:TFontPitch;` – визначає ширину шрифту (`fpDefault` – ширина задається типом шрифту; `fpVariable` – ширина символів змінюється; `fpFixed` – ширина символів є фіксованою);
- **property Size:integer;** – визначає висоту шрифту у пунктах (1 пункт =1/72 дюйма);
- **type** `TFontStyle= (fsBold, fsItalic, fsUnderline, fsStrikeOut);` `TFontStyles=`**set of** `TfontStyle;` **property** `Style:TfontStyles;` – визначає накреслення шрифту (`fsBold` – напівжирний; `fsItalic` – курсив; `fsUnderline` – підкреслений; `fsStrikeOut` – перекреслений;

Зауважимо, що тексти у Delphi 6 пишуть тільки горизонтально.

Клас `TPen` задає властивості олівця, який створює зображення ліній. Основні властивості цього класу:

- **property** `Color:TColor;` – визначає колір лінії (за домовленістю – чорний);
- **property** `Mode:TPenMode;` – визначає додатково колір малювання ліній олівцем. Деякі значення `Mode`, які визначені у типі `TPenMode`: `pmBlack` –чорна лінія; `pmWhite` – біла лінія; `pmNop` – безколірна лінія; `pmNot` – колір, протилежний до кольору екрана; `pmCopy` – колір визначається властивістю `Color` і т.д. (див. електронну довідку). За домовленістю колір лінії визначається властивістю `Color`;

- **property** `Style:TPenStyle;` – визначає стиль лінії. Значення `Style`, які визначені у типі `TPenStyle`: `psSolid` – суцільна лінія (за домовленістю); `psDash` – штрихова лінія; `psDot` – пунктирна лінія; `psDashDot` – штрихпунктирна лінія; `psDashDotDot` – штрихпунктирна лінія з двома пунктирами; `psClear` – невидима лінія; `psInsideFrame` – лінія всередині рамки;
- **property** `Width:integer;` – визначає товщину лінії у пікселях (за домовленістю – *один* піксель).

Клас `TBrush` задає властивості пензля, який зафарбовує замкнуті області. Основні властивості пензля:

- **property** `Bitmap:TBitmap;` – вказує на об'єкт типу `Tbitmap`, який містить побітове зображення розміром 8x8 пікселів; Якщо `Bitmap` не порожний, то шаблон заповнення визначається ним, а не властивістю `Style`, а інакше – значення `nil`.
- **property** `Color:TColor;` – визначає колір зафарбування (за домовленістю – білий);
- **property** `Style:TBrushStyle;` – визначає орнамент пензля. Значення `Style`, які визначені у типі `TBrushStyle`: `bsSolid` – суцільне зафарбування; `bsClear` – відсутність зафарбування; `bsHorizontal/bsVertical` – горизонтальні/вертикальні лінії; `bsFDiagonal/bsBDiagonal` – ліві/праві діагональні лінії; `bsCross` – клітинка; `bsDiagCross` – коса клітинка.

11.5.3. Клас `TStrings`

Абстрактний клас **`TStrings`** містить методи і властивості, що дають змогу маніпулювати зі **списками рядків**. Цей клас широко використовується у властивостях багатьох компонентів. Основні властивості цього класу:

- **property** `Count: integer;` – кількість рядків у списку. Властивість тільки для читання;
- **property** `Strings[Index: integer]:` **`String;`** – визначає текст рядка з вказаним індексом. Індекс першого рядка – 0;
- **property** `Text:` **`String;`** – подає весь список як один рядок, всередині якого використовуються роздільники – символи повернення каретки і переведення рядка;

Основні методи класу `TStrings`*:*

function `Add(`**const** `S: String): virtual;`

Додає рядок `S` у кінець списку. Повертає індекс цього рядка. У вікнах редагування рядок, що додається, може виявитися розбитим на декілька, тоді індекс, що повертається, ні на що не вказує.

procedure `Clear;` **virtual; abstract;**

Видаляє усі рядки списку.

procedure `Delete(Index: integer): virtual; abstract;`

Видаляє зі списку вказаний рядок з індексом `Index`.

procedure `Exchange(Index11,Index12: integer);` **virtual;**

Переставляє місцями рядки списку з індексами `Index1` і `Index2`.

procedure `Insert(Index: integer;` **const** `S: String);`
virtual; abstract;

Вставляє вказаний рядок `S` у список під рядком з індексом `Index`. Якщо `Index` = `0`, то рядок вставляється на першу позицію.

procedure `LoadFromFile(`**const** `FileName: String);` **virtual;**

Заповнює список рядками тексту з вказаного файлу `FileName`.

procedure `SaveToFile(`**const** `FileName: String);` **virtual;**

Зберігає рядки списку у файлі з вказаною назвою `FileName`.

procedure `Move(CurIndex, NewIndex: integer):` **virtual;**

Змінює елемент списку з позиції `CurIndex` у позицію `NewIndex`.

? Запитання для самоперевірки

1. Що таке компонент?
2. Яка різниця між компонентом і об’єктом?
3. Опишіть ієрархію класів VCL.
4. Яка різниця між візуальними й невізуальними компонентами?
5. Опишіть клас `TComponent`.
6. Опишіть основні властивості класу `TControl`.
7. Опишіть основні методи класу `TControl`.
8. Опишіть основні події класу `TControl`.
9. Що таке елемент керування?
10. Що таке батьківський та дочірній елемент керування?
11. Що таке клієнтська область елемента керування?
12. Для чого використовують параметр `Sender`?
13. Опишіть події та методи класу `TWinControl`.

12. ПРАКТИЧНЕ ВИКОРИСТАННЯ КОМПОНЕНТІВ

План викладу матеріалу

1. Компоненти введення і відображення текстових даних
2. Перемикачі та кнопки
3. Діалогові вікна
4. Стандартні діалоги
5. Компоненти меню
6. Приклад використання меню та діалогових вікон на прикладі побудови текстового редактора
7. Багатовіконний інтерфейс
8. Таймер
9. Створення і опрацювання графічних зображень

Ключові терміни розділу

- *Позначки*
- *Клас* `TCustomEdit`
- *Компонент класу* `TListBox`
- *Компонент класу* `TComboBox`
- *Компонент класу* `TUpDown`
- *Компонент класу* `TButton`
- *Компонент класу* `TRadioButton`
- *Компонент класу* `TCheckBox`
- *Компонент класу* `TOpenDialog`
- *Компонент класу* `TOpenPictureDialog`
- *Стандартні діалоги*
- *Компонент класу* `TPopupMenu`
- *Компонент класу* `TActionList`
- *Допоміжні форми*
- *Стиль інтерфейсу MDI*
- *Таймер*
- *Компонент класу* `TShape`
- *Панель*
- *Однорядкові текстові редактори*
- *Багаторядкові текстові редактори*
- *Компонент класу* `TStringGrid`
- *Компонент класу* `TSpinEdit`
- *Компонент класу* `TSpeedButton`
- *Компонент класу* `TRadioGroup`
- *Компонент класу* `TCheckListBox`
- *Компонент класу* `TSaveDialog`
- *Компонент класу* `TSavePictureDialog`
- *Компонент класу* `TMainMenu`
- *Компонент класу* `TImageList`
- *Однодокументний інтерфейс*
- *Модальність форми*
- *Батьківська та дочірня форми*
- *Компонент класу* `TImage`
- *Компонент класу* `TPaintBox`

У цьому розділі описано найбільш використовувані компоненти. Для зручності компоненти об’єднані за функціональними ознаками (тобто одночасно описано компоненти з різних сторінок палітри компонентів, якщо вони виконують схожі функції).

Вивчення компонентів підтримується побудовою демонстраційних проектів, які підібрано так, щоб для початку їх можна було виконати, докладаючи мінімальні зусилля, не використовуючи деталі, несуттєві для конкретної задачі. Водночас дотримано принципу наростання складності при побудові проектів. Демонстраційні проекти мають яскраво виражений навчальний характер, а тому їх можна вдосконалювати і розширювати.

Описуючи компоненти, звертаємо увагу на властивості та методи, які є значущими та унікальними для конкретного компонента (визначають його функціональність). Зауважимо, що загальні властивості та події компонентів, які наслідуються ними від базових класів (`TComponent`, `TControl`, `TWincontrol` тощо), описано в одинадцятому розділі. В разі потреби ці властивості та події будуть деталізуватися та уточнюватися.

При описі компонента його піктограму наводити не будемо, адже ідентифікувати компонент на палітрі компонентів надзвичайно просто за його підказкою (зупинити на декілька секунд маркер миші над компонентом). Якщо підказка не виводиться, то для її активізації треба виконати команди:

❑**ПКМ** //над панеллю швидкого доступу – отримуємо контекстне меню

➢**Customize …** 🗀**Options** ☑**Show tooltips** ❑**Close**

12.1. Компоненти введення і відображення текстових даних

12.1.1. Перелік основних компонентів

У VCL є дуже багато компонентів, які дають змогу відображати, вводити та редагувати текстову інформацію. У таблиці 12.1 перелічені основні компоненти з їхньою короткою характеристикою і вказівкою основних властивостей, які містять уведений або відображений текст (результати роботи програми).

У всіх компонентах шрифт тексту, його розмір, накреслення і колір визначає властивість `Font`, яка має багато підвластивостей.

Таблиця 12.1. Основні компоненти введення/відображення тексту

Клас	Сторінка	Опис
1	2	3
TLabel (текстовий підпис або позначка)	Standart	Використовується для розміщення тексту, який не змінює користувач. Основна властивість – Caption
TStaticText (позначка з рамкою)	Additional	Аналог компонента Label. Властивість BorderStyle забезпечує можливість обведення тексту рамкою або виділення його вгнутою частиною форми. Основна властивість – Caption
TPanel (панель)	Standart	Контейнер для групування елементів керування. Може також відображати текст з можливостями об'ємного оформлення панелі. Основна властивість – Caption
TEdit (однорядковий текстовий редактор або поле введення)	Standart	Використовується для введення і редагування однорядкових текстів. Може бути використаний і для відображення тексту. Властивість BorderStyle забезпечує можливість обведення вікна редактора рамкою. Основна властивість – Text
TMaskEdit (спеціальний текстовий редактор рядка)	Additional	Використовується для форматування даних або для введення символів згідно з шаблоном. Основні властивості – Text і EditMask
TMemo (вікно багаторядкового редактора)	Standart	Використовується для введення і редагування багаторядкових текстів. Може використовуватися і для відображення тексту. Властивість BorderStyle забезпечує можливість обведення вікна рамкою. Основна властивість – Lines

Закінчення табл. 12.1

1	2	3
TRichEdit (вікно багаторядкового редактора у форматі RTF)	Win32	Вікно редагування для введення, редагування і відображення багаторядкових текстів; дає змогу вибирати атрибути шрифту, шукати текст тощо. Властивість BorderStyle забезпечує можливість обведення вікна рамкою. Основна властивість – Lines
TListBox (список вибору)	Standart	Вміщує список текстових рядків і дає змогу вибрати під час виконання програми деякий рядок і повідомити про цей вибір. Основна властивість – Items
TComboBox (комбінований список вибору)	Standart	Об'єднує функції ListBox і Edit. Користувач може або ввести текст, або вибрати його зі списку. Основна властивість – Items
TCheckListBox (список множинного вибору)	Additional	Компонент є комбінацією властивостей списку ListBox і незалежного перемикача CheckBox. Дає змогу вибору декількох рядків під час виконання програми
TStringGrid (таблиця текстових рядків)	Additional	Використовується для відображення текстової інформації у таблиці з рядків і стовпців. Основна властивість – Cells

12.1.2. Позначки, панель

Компонент класу **TLabel** (текстовий підпис або позначка) – **невіконний** елемент керування, який відображає на формі прямокутну область статичного тексту. Ієрархія:

TObject – TPersistent – TComponent – TControl – TGraphicControl – TCustomLabel – TLabel

Клас TLabel є спадкоємцем класу TCustomLabel, у якому визначені основні властивості позначок. Цей клас є прямим спадкоємцем класу TGraphicControl, і тому позначки реагують тільки на події від миші.

Зазвичай текст є поясненням (розгорнутою назвою) іншого компонента. Текст позначки задають через властивість `Caption`, яка має тип **`String`**. Властивість `Alignment` визначає спосіб вирівнювання тексту:

```
type TAlignment = (taLeftJustify, taRightJustify, taCenter);
property Alignment: TAlignment;
```

Позначка можлива як однорядкова, так і багаторядкова. Якщо текст не вміщується у відведене для позначки місце на формі, то варіанти виведення наступних символів можуть бути такими: текст буде або **обрізаний**, або **перенесений** на наступний рядок, або позначка автоматично **змінюватиме** свої **розміри** для виведення усього тексту. Вибирати варіант виведення символів, що не поміщаються на позначці, можна за допомогою властивостей `AutoSize` і `WordWrap` (див. табл. 12.2).

Таблиця 12.2. Варіанти виведення символів на позначці

AutoSize	**WordWrap**	**Варіант виведення символів**
`false`	`false`	Розміри позначки не збільшуються. Символи, що не поміщаються, не виводяться
`false`	`true`	Розміри позначки не збільшуються. При досягненні правої межі позначки буде здійснюватися перенесення слів доти, доки розміри області позначки це дозволятимуть
`true`	`false`	Зі збільшенням довжини рядка перенесення слів не буде, а горизонтальний розмір позначки буде збільшуватися доти, доки текст не досягне правого краю форми
`true`	`true`	У разі досягнення правої межі позначки відбуватиметься перенесення слів на наступний рядок. При потребі збільшуватиметься вертикальний розмір позначки. Якщо ж окреме **слово** не поміщається на позначці, то водночас буде збільшуватися горизонтальний розмір позначки

Компонент класу **TStaticText** (позначка з рамкою) –**віконний** елемент керування, який відображає на формі прямокутну область статичного тексту. Ієрархія:

TObject – TPersistent – TComponent – TControl – TWinControl – TCustomStaticText – TStaticText

Компонент класу **TStaticText** подібний до компонента класу TLabel. Властивість BorderStyle забезпечує можливість обведення тексту рамкою або виділення його вгнутою частиною форми. Текст позначки задається через властивість Caption. Перенесення довгого тексту здійснюється автоматично, якщо значенням властивості AutoSize є false і розмір компонента достатній для розміщення декількох рядків.

Компонент класу **TPanel** (панель) – створює порожний контейнер (прямокутну область), який може містити інші компоненти. Ієрархія:

TObject – TPersistent – TComponent – TControl – TWinControl – TCustomControl – TCustomPanel – TPanel

Цей компонент можна використати для створення на формі панелей інструментів, рядків стану тощо. Може також відображати текст (властивість Caption). Компонент має можливості об'ємного оформлення панелі (властивості BevelInner, BevelOuter, BevelWidth, BorderWidth, BorderStyle). Властивість Align дає змогу "прив'язати" панель до визначеного місця на формі. Якщо панель переміщають на формі, то усі компоненти панелі переміщаються разом з нею. На панелі неможливо розмістити текст з декількох рядків.

Приклад 12.1. Для ілюстрації можливостей позначки відображати дані створимо **проект табулювання квадратичної функції**:
нехай у вікні форми буде розміщенийний компонент Label1 для відображення результатів табулювання функції і дві кнопки з підписами "**Обчислити**" і "**Закрити**". Після натиснення цих кнопок відповідно буде табулюватися функція чи закриватися вікно форми (заінчуватися робота застосування).

Етап проектування інтерфейсу проекту:

```
Form1: Button1; Button2; Label1
Button1.Caption:= 'Обчислити'
```

```
Button2.Caption:= 'Закрити'
Label1.Caption:= '' // Початковий порожній рядок
Label1.AutoSize:= true, Label1.WordWrap:= false
// Нові рядки формуватимуться програмно, треба тільки
// задати можливість розширення позначки
Form1.Caption := 'Табулювання функції'
```

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.Button1Click(Sender: TObject);
var x, dx, y: real; i: integer;
begin x:=-2; dx:=0.5;
 label1.Caption:=' Аргумент   Функція;
  for i:=1 to 10 do
    begin y:= 7*x*x-5*x;
       label1.Caption:=label1.Caption + chr(13)
       +floatToStrf(x,ffFixed,8,2)+
       '         '+ floatToStrf(y,ffFixed,8,2);
       // Перехід до нового рядка здійснюється
       // за допомогою символу #13
       x:=x+dx
    end
end;
procedure TForm1.Button2Click(Sender: TObject);
begin close end;
```

Для відокремлення значень аргументу та функції використовуємо пропуски, а тому треба відключити автоматичне перенесення слів (`WordWrap:=false`). Функція `FloatToStrF(…)` перетворює раціональне число у рядок символів за допомогою формату з фіксованою крапкою `ffFixed` (див. параграф 3.4).

12.1.3. Властивості та методи класу `TCustomEdit`

Для однорядкових і багаторядкових редакторів базовим абстрактним класом є клас **`TCustomEdit`**, у якому інкапсульовані основні методи і властивості, що використовуються при редагуванні текстів. Створити об’єкти класу `TCustomEdit` неможливо. Цей клас використовується тільки для створення похідних класів, що успадковують особливості опрацювання текстів. Властивості цього класу наведено у таблиці 12.3, а методи – у таблиці 12.4.

Таблиця 12.3. Властивості класу `TCustomEdit`

Властивість	**Тип**	**Опис**
1	2	3
`AutoSelect`	`Boolean`	Визначає, чи виділяється весь текст при отриманні компонентом фокуса
`AutoSize`	`Boolean`	Визначає, чи висота компонента пристосовується до розміру тексту
`BorderStyle`	`TBorder-Style`	Визначає наявність рамки навколо вікна редагування
`CanUndo`	`Boolean`	Вказує, чи зроблені операції редагування, які можна відмінити командою `Undo`. Властивість тільки для читання
`CharCase`	`TEdit-CharCase`	Визначає регістр тексту, що відображається. Можливі значення `TEditCharCase`: `ecNormal` – символи відображаються з урахуванням регістра, у якому вони вводилися; `ecUpperCase`/`ecLowerCase` – символи тексту приводяться до верхнього/нижнього регістра
`HideSelection`	`Boolean`	У разі `true` робить невидимим виділений текст, коли компонент втрачає фокус. Інакше – виділення залишається
`MaxLength`	`Integer`	Встановлює максимальну довжину тексту. У разі *нуля* – довжина довільна
`Modified`	`Boolean`	Вказує, чи був змінений текст
`PasswordChar`	`char`	Вказує символ, який з'являється у вікні редагування замість символу, що вводиться користувачем
`ReadOnly`	`Boolean`	Встановлює заборону редагування тексту користувачем
`SelLength`	`Integer`	Вказує довжину виділеного тексту
`SelStart`	`Integer`	Вказує на перший символ виділеного тексту. Якщо виділення немає, то вказує на символ, перед яким стоїть курсор

Закінчення табл. 12.3

1	2	3
SelText	**string**	Містить виділений текст. Можна задавати значення SelText, щоб замінити ним виділений текст. Якщо виділення немає, то значення SelText буде вставлено згідно значення SelStart

Таблиця 12.4. Методи класу TCustomEdit

Метод	Опис
Clear	Видалення всього тексту
ClearSelection	Видалення виділеного тексту
ClearUndo	Очищення буфера, що використовується для команди Undo
CopyToClipboard	Копіювання у буфер Clipboard виділеного тексту
CutToClipboard	Вирізування у буфер Clipboard виділеного тексту
PasteFromClipboard	Вставка тексту з буфера Clipboard замість виділеного тексту або, якщо немає виділення, то вставка тексту в позицію курсора
SelectAll	Виділення всього тексту
Undo	Скасування результатів останньої операції

У класі TCustomEdit визначена тільки одна подія OnChange, яка виникає при спробі зміни користувачем тексту. Чи сталася зміна насправді, можна визначити через властивість Modified.

Вікна редагування мають багато функцій, які властиві більшості редакторів. Наприклад, у них передбачені типові комбінації клавіш: Ctrl+C – копіювання виділеного фрагмента тексту в буфер обміну; Ctrl+X – вирізання виділеного фрагмента тексту в буфер обміну; Ctrl+V – вставка тексту з буфера обміну; Ctrl+Z – відміна останньої команди редагування.

12.1.4. Однорядкові текстові редактори

Компонент класу **TEdit** (однорядковий текстовий редактор або поле введення) – відображає на формі вікно для введення і редагування одиночного рядка тексту. Введений рядок є значенням властивості Text типу **String**. Ієрархія:

TObject – TPersistent – TComponent – TControl – TWinControl – TCustomEdit – TEdit

Компонент класу **TMaskEdit** відрізняється від компонента класу TEdit тим, що у ньому можна задавати рядок маски (шаблону) у властивості EditMask. Це забезпечує синтаксично правильне введення таких даних, як номери телефонів, адреси, дати, час тощо. Деталі застосування шаблонів можна знайти у вбудованій електронній довідці Delphi. Ієрархія:

TObject – TPersistent – TComponent – TControl – TWinControl – TCustomEdit – TCustomEditMask –TEditMask

Компоненти класу **TEdit** і **TMaskEdit** можна використовувати і для відображення даних, якщо встановити властивість ReadOnly у true. У цьому випадку, на відміну від позначок, є можливість прокручувати текст, що не поміщається у вікні.

Властивість AutoSize в однорядкових вікнах редагування задає тільки автоматичне розширення вертикального розміру (наприклад, при зміні розміру шрифту).

Властивість AutoSelect, яка встановлена у true, показує, що при отриманні однорядковим редактором фокуса увесь текст виявиться виділеним. Властивість AutoSelect доцільно встановлювати у true, якщо за умовами роботи користувач швидше буде замінювати текст, що є у вікні, ніж доповнювати його.

12.1.5. Багаторядкові текстові редактори

Компоненти класів **TMemo** і **TRichEdit** є вікнами редагування багаторядкових текстів. Зміст області редагування визначає масив рядків, який є значенням властивості Lines, що має тип TStrings.

У компонента класу **TMemo** формат шрифту є однаковим для всього тексту і визначається властивістю Font. Ієрархія:

TObject – TPersistent – TComponent – TControl – TWinControl – TCustomEdit – TCustomMemo –TMemo

Компонент класу **TRichEdit** працює з текстом у збагаченому форматі **RTF** (тобто можна по-різному формувати окремі фрагменти тексту). Для зміни атрибутів тексту використовують властивість SelAttributes. Компонент має також властивість DefAttributes, яка задає атрибути тексту за домовленістю. За вирівнювання, відступи і т.п. у межах поточного абзаца відповідає властивість Paragraph, значення підвластивостей якої можна змінювати тільки під час виконання програми. Ієрархія:

TObject – TPersistent – TComponent – TControl – TWinControl – TCustomEdit – TCustomMemo – TcustomRichEdit – TRichEdit

Якщо зберегти у файлі текст, який був введений у компонент класу TMemo, то буде створений текстовий файл, що містить тільки символи і не містить елементів форматування. При подальшому читанні цього файлу у компоненті класу TMemo формат тексту буде визначатися поточним станом властивості Font цього компонента.

Для компонента класу TRichEdit у файлі зберігається не тільки текст, але й елементи форматування, а тому в разі відкриття цього файлу у компоненті класу TRichEdit отримаємо початкове (збережене) форматування тексту.

Коротко зупинимося на спільних властивостях цих компонентів. Властивості Alignment і Wordwrap мають такий самий смисл, що й у позначках. Властивості WantReturns і WantTab задають допустимість уведення користувачем у текст символів переведення рядка і табуляції. Властивість ScrollBars визначає наявність у вікні полос прокручування тексту.

Основна властивість вікон класів TMemo і TRichEdit – властивість Lines типу Tstrings, що містить текст вікна у вигляді списку рядків. Початкове значення тексту можна встановити у процесі проектування, натиснувши кнопку з трикрапкою властивості Lines у вікні інспектора об’єктів. Після цього відкриється вікно редактора списку рядків, у якому можна вводити/редагувати текст.

Під час виконання програми текст у вікна класів TMemo/TRichEdit вводять за допомогою методів класу TStrings. Доступ до окремого рядка тексту можна отримати за допомогою властивості Strings[Index: Integer]. Індекс першого рядка – 0. Отже, Memo1.Lines.Strings[0] – це текст першого рядка.

Приклад 12.2. Для ілюстрації можливостей вікон класу **TMemo** відображати дані створимо **проект табулювання квадратичної функції**:

нехай у вікні форми буде розташований компонент **Memo** для відображення результатів табулювання функції і дві кнопки з підписами "**Обчислити**" і "**Закрити**". Після натиснення цих кнопок відповідно буде табулюватися функція чи закриватися вікно форми.

Етап проектування інтерфейсу проекту:

```
Form1: Button1; Button2; Memo1
Button1.Caption := 'Обчислити'
Button2.Caption := 'Закрити'
Memo1.ScrollBars:= ssBoth
Form1.Caption := 'Табулювання функції'
```

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.Button1Click(Sender: TObject);
var x, dx, y: real; i: integer;  st: String;
begin x:=-2; dx:=0.5;
 memo1.Lines.Clear;
 st:=' Аргументи    Функція ';
 If Memo1.ReadOnly then Memo1.ReadOnly:=false;
 Memo1.Lines.Append(st);
  for i:=1 to 10 do
    begin y:= 7*x*x-5*x;
       st:=floatToStrf(x,ffFixed,8,2)+
          '         '+ floatToStrf(y,ffFixed,8,2);
          x:=x+dx;
      Memo1.Lines.Append(st);
    end;
    Memo1.ReadOnly:=true;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin close end;
```

12.1.6. Компоненти відображення списків рядків

Компоненти класів **TListBox і TComboBox** відображають списки рядків. Використання цих компонентів забезпечує безпомилкове введення інформації користувачем у тих випадках, коли він повинен вибрати одну відповідь з деякої множини значень.

Компоненти списків відрізняються один від одного передусім тим, що **TListBox** *тільки відображає* дані і дає змогу користувачеві вибрати з них те, що йому треба, а **TComboBox** *дає можливість також і редагувати* дані. Крім того розрізняють і форму відображення списків: **TListBox** відображає список у розкритому вигляді і автоматично додає у список вертикальну смугу прокручування, якщо всі рядки не вміщуються у вікні компонента[1]; **TComboBox** дає змогу відображати список як у розгорненому вигляді, так і у вигляді випадаючого списку.

Основною властивістю компонентів є властивість `Items` типу `TStrings`. Заповнити список (властивість `Items`) під час проектування можна, якщо натиснути кнопку з трьома крапками біля цієї властивості у вікні інспектора об'єктів. Після цього відкривається вікно редагування списку. Кожний записаний у вікні рядок буде відповідати окремому рядку списку. Під час виконання програми можна працювати з властивістю `Items`, користуючись властивостями і методами класу `TStrings`.

Компонент класу **TCheckListBox** – це список з індикаторами (незалежними перемикачами). Виглядає він так само, як компонент класу `TListBox`, але біля кожного рядка є індикатор, який користувач може перемикати. Індикатори можна перемикати і програмно, якщо список використовується для виведення і необхідно в ньому відмітити якусь характеристику деякого об'єкта, наприклад, наявність товару певного найменування на складі.

Усі властивості, що характеризують компонент класу `TCheckListBox` як список, аналогічні класу `TListBox`, за винятком властивостей, що визначають множинний вибір. Ці властивості компоненту класу `TCheckListBox` не потрібні, оскільки у ньому множинний вибір можна здійснювати установкою незалежних перемикачів (або індикаторів). Властивості, пов'язані з індикаторами, описані у параграфі 12.2.

---

[1] Горизонтальна смуга прокручування може відображатися тільки у випадку, якщо значення властивості `Columns`>1. Відображення цієї смуги можна задати програмно (див. приклад 12.4).

12.1.7. Компонент класу `TListBox`

Компонент класу **`TListBox`** має властивість `MultiSelect`, яка дає змогу користувачеві здійснити множинний вибір у списку.

Якщо `MultiSelect = false` (значення за домовленістю), то користувач може вибрати тільки один елемент зі списку. У цьому випадку можна дізнатися індекс вибраного рядка з властивості `ItemIndex`**,** яка доступна тільки під час виконання програми. Якщо жоден рядок не вибраний, то `ItemIndex` =–1. Початкове значення `ItemIndex` **неможливо** задати під час проектування (`ItemIndex =-1` – за домовленістю). Якщо ж все-таки потрібно задати цій властивості якесь інше значення, ніж за домовленістю, то зробити це можна, наприклад, у процедурі опрацювання події `OnCreate` форми**,** ввівши в нього оператор

```
ListBox1.ItemIndex:=0;
```

Приклад 12.3. Для ілюстрації можливостей компонента класу `TListBox` відображати дані створимо **проект переведення фунтів у кілограми.** У різних країнах коефіцієнт переведення фунтів у кілограми є різним, а тому у проекті буде створено список країн. Залежно від вибору країни у списку програма за допомогою оператора `case` вибиратиме відповідний коефіцієнт перерахунку і здійснюватиме перерахування фунтів у кілограми (див. рис. 12.1).

Етап проектування інтерфейсу проекту:

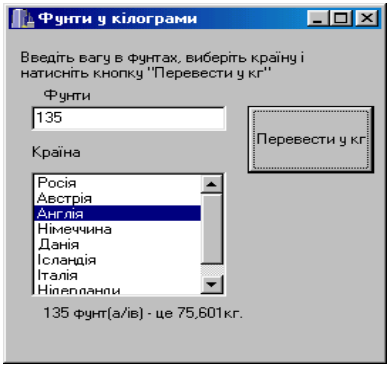


Рис. 12.1. Переведення фунтів у кг

Form1: `Button1; Label1; Label2; Label3; Label4; Edit1; ListBox1`

```
Label1.Caption:= 'Введіть вагу в фунтах, виберіть країну і натисніть кнопку "Перевести у кг" '
Label2.Caption:= 'Фунти'
Label3.Caption:= 'Країна'
Label4.Caption:= ''
Edit1.Text:= ''
Button1.Caption:='Перевести у кг'
Listbox1.Items:= ''
//Заповнення списку у програмі
```

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items.Add('Росія');
  ListBox1.Items.Add('Австрія');
  ListBox1.Items.Add('Англія');
  ListBox1.Items.Add('Німеччина');
  ListBox1.Items.Add('Данія');
  ListBox1.Items.Add('Ісландія');
  ListBox1.Items.Add('Італія');
  ListBox1.Items.Add('Нідерланди');
  ListBox1.Itemindex:=0;
end;
procedure TForm1.Button1Click(Sender: TObject);
var
   funt:real; { вага у фунтах }
   kg:real;   { вага у кілограмах }
   k:real;    { коефіцієнт перерахунку }
begin
 case ListBox1.ItemIndex of
      0: k:=0.4059;  { Росія }
      1: k:=0.453592;{ Англія }
      2: k:=0.56001; { Австрія }
      3..5,7:k:=0.5; { решта країн }
      6: k:=0.31762; { Італія }
 end;
 Label4.Caption:='';
 try
   funt:=StrToFloat(Edit1.Text);
   kg:=k*funt;
   label4.caption:=Edit1.Text
                 + ' фунт(а/ів) - це '
                 + FloatToStrF(kg,ffFixed,6,3)
                 + 'кг.';
 except
    on EConvertError do // помилка перетворення
      begin ShowMessage('Задайте число.');
            form1.Edit1.SetFocus           end
 end
end;
```

У разі множинного вибору (`MultiSelect = true`) значення `ItemIndex` ***відповідає*** тому елементу списку, який знаходиться у фокусі. При множинному виборі перевірити, чи вибрано даний елемент, можна за допомогою властивості

```
property Selected[Index: Integer]: Boolean;
```

На спосіб множинного вибору впливає властивість `ExtendedSelect`. Якщо `ExtendedSelect = true`, то користувач може виділити інтервал елементів, виділивши один з них, а потім натиснути клавішу `Shift` і перевести курсор до іншого елемента. Виділити не прилеглі один до одного елементи користувач може, якщо під час вибору буде увесь час утримувати натисненою клавішу `Ctrl`. Якщо ж `ExtendedSelect = false`, то клавіші `Shift` і `Ctrl` при виборі не працюють –виділити елементи списку можна тільки мишею.

Властивість `Columns` визначає число стовпців, в яких буде відображатися список, якщо він не вміщується цілком у вікні **`ListBox`** в одному стовпці. Якщо значення властивості `Columns>1`, то у цьому випадку можливою є поява горизонтальної смуги прокручування. Однак таке рішення не є досконалим, адже відповідно до наших стереотипних уявлень однотипна інформація має відображатися в одному стовпці. Компонент `Listbox` не допускає появи горизонтальної смуги прокручування, якщо `Columns=1`.

Тому процедурі відображення вікна необхідно посилати повідомлення Windows з вимогою дорисувати до списку горизонтальну смугу прокручування. Цю дію реалізує метод `Perform`.

Властивість `Sorted` дає змогу упорядкувати список за алфавітом. При `Sorted = true` нові рядки додаються не у кінець списку, а за алфавітом.

Приклад 12.4. Для ілюстрації можливостей множинного вибору рядків у списку **`ListBox`** створимо **проект вибору страви з меню** (списку страв). Список страв сформуємо на етапі проектування інтерфейсу. Передбачимо можливість появи горизонтальної смуги прокручування, якщо довжини деяких рядків списку перевищуватимуть горизонтальний розмір **`ListBox`**. На рис. 12.2 зображено результати роботи застосування.

Етап проектування інтерфейсу користувача:

```
Form1: Button1; Label1; Label2; Memo1; ListBox1
Label1.Caption:='Меню'; Label2.Caption:='Замовлено'
Memo1.Lines ❑(…):=''
Memo1.ScrollBars:=ssBoth
Listbox1.Items ❑(…):=
' Салат грецький
  Борщ український
  Солянка
  Чанахі
  Котлета київська
  Шашлик
  Картопля фрі
  Картопля пюре
  Кава
  Чай
  Пиріжок
  Рулет   '
```

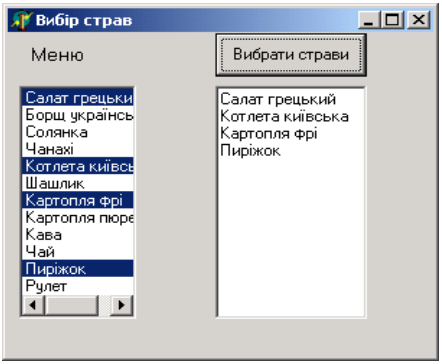


Рис. 12.2. Реалізація мультивибору

```
Listbox1.MultiSelect := true
Listbox1.ExtendedSelect := true
Button1.Caption:= 'Вибрати страви'
Form1.Caption:= 'Вибір страв'
```

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.FormCreate(Sender: TObject);
var maxWidth, w, i:integer; text:String;
begin
 maxWidth:= 0;
 // Обчислити максимальну довжину рядка [pixels]
 With ListBox1 do
  begin
   for i:=0 to Items.Count-1 do
    begin
      text:=Items.Strings[i];
      w:=Canvas.TextWidth(text);
      if w> maxWidth then maxWidth:=w;
    end;
     perform(LB_SETHORIZONTALEXTENT, maxWidth, 0);
  end;
  Memo1.ReadOnly:=true;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var i:integer; text:String;
begin
  If Memo1.ReadOnly then Memo1.ReadOnly:=false;
  Memo1.Lines.Clear;
  With ListBox1 do
   for i:=0 to Items.Count-1 do
    if Selected[i] then
      begin text:=Items.Strings[i];
            Memo1.Lines.Add(text)
      end;
  Memo1.ReadOnly:=true;
end;
```

12.1.8. Компонент класу TComboBox

Стиль зображення компонента класу **TComboBox** визначається властивістю Style, яка може набувати таких значень:

csDropDown	Комбінація текстового поля редагування і випадаючого списку з рядками однакової висоти
csSimple	Комбінація текстового поля редагування і статичного (розгорненого) списку фіксованої довжини, яка задається властивістю Height. Цей стиль використовують для коротких списків
csDropDownList	Випадаючий список з рядками однакової висоти. Цей стиль використовують при забороні редагування
csOwnerDrawFixed	Випадаючий список csDropDown з графічним прорисовуванням елементів однакової висоти
csOwnerDrawVariable	Випадаючий список типу csDropDown з графічним прорисовуванням елементів різної висоти

Вибір користувача або введений ним текст можна визначити за значенням властивості `Text`. Якщо ж треба визначити індекс вибраного користувачем елемента списку, то можна скористатися властивістю `ItemIndex`, яка доступна тільки під час виконання. Все сказане раніше про `ItemIndex` є справедливим і для компонента класу `TComboBox`. Якщо у вікні компонента `TComboBox` проводилося редагування, то `ItemIndex=-1`.

Наголосимо, що для компонента класу `TComboBox` завдання початкового значення `ItemIndex` є ще актуальнішим, ніж для компонента класу `TListBox`. Якщо початкове значення не задане, то у момент запуску застосування вікно редагування списку буде відображати не один з елементів списку, а значення властивості `Text`. Отже, треба або вводити у подію створення форми оператор, який задає значення `ItemIndex`, або вводити під час проектування у властивість `Text` якесь запрошення до дальших дій. Інакше користувач буде у здивуванні, що ж треба робити зі списком.

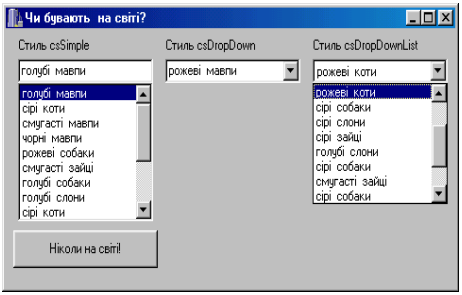


Рис. 12.3. Видалення рядків зі списків `TComboBox`

Приклад 12.5. Для ілюстрації роботи зі списками `TComboBox` побудуємо **проект**, який дає змогу знаходити у трьох комбінованих списках тварин, які зафарбовані датчиком випадкових чисел у неадекватний колір, а потім видаляти їх (див. рис. 12.3).

Етап проектування інтерфейсу користувача:

Form1: `Button1; Label1; Label2; Label3; ComboBox1; ComboBox2; ComboBox3`

```
Button1.Caption:= 'Ніколи на світі!'
Label1.Caption:= 'Стиль csSimple'
Label2.Caption:= 'Стиль csDropDown'
Label3.Caption:= 'Стиль csDropDownList'
ComboBox1.Name:= 'Simple';  ComboBox1.Style:=csSimple
ComboBox2.Name:='DropDown'; ComboBox2.Style:=csDropDown
ComboBox3.Name:= 'DropDownList'
```

```
ComboBox3.Style:=csDropDownList
Form1.Caption:= 'Чи бувають на світі?'
```

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.FormCreate(Sender: TObject);
var i: integer; buff: String;
const
 cols: array[0..4] of String = ('сірі', 'голубі',
                      'чорні', 'рожеві', 'смугасті');
 they: array[0..4] of String = ('мавпи', 'коти',
                      'собаки', 'зайці', 'слони');
begin  randomize();
 //Заповнити всі списки 15 випадковими комбінаціями пар слів
 for i:=0 to 14 do
  begin
   buff:=cols[random(5)]+'  '+they[random(5)];
   Simple.Items.Add(buff);
   buff:=cols[random(5)]+'  '+they[random(5)];
   DropDown.Items.Add(buff);
   buff:=cols[random(5)]+'  '+they[random(5)];
   DropDownList.Items.Add(buff);
  end
end;
procedure TForm1.Button1Click(Sender: TObject);
 var Index:integer;
begin
 // Видалити рядки, які вибрані зі списків
 Index:=Simple.Items.IndexOf(Simple.Text);
 Simple.Items.Delete(Index);
 Index:=DropDown.Items.IndexOf(DropDown.Text);
 DropDown.Items.Delete(Index);
 Index:=DropDownList.Items.IndexOf(DropDownList.Text);
 DropDownList.Items.Delete(Index);
end;
```

12.1.9. Компонент класу `TStringGrid`

Компонент класу **`TStringGrid`** – використовується для відображення інформації у вигляді таблиці, яка містить набір комі-

рок, розміщених у рядках і стовпцях. Таблиця може мати смуги прокручування, причому певне число перших рядків і стовпців може бути фіксованим і не прокручуватися. Кожна комірка може містити текстове значення (рядок символів) з асоційованими об'єктами. Таким чином, можна задавати заголовки стовпців і рядків, які будуть постійно наявними у вікні компонента. Ієрархія:

`TObject – TPersistent – TComponent – TControl – TWinControl – TCustomControl – TCustomGrid – TCustomDrawGrid – TDrawGrid – TStringGrid`

Як і компонент класу `TStringGrid`, компонент класу **`TDrawGrid`** призначений для відображення інформації у вигляді таблиці. Відрізняється він тим, що може містити у комірках графічні зображення.

Основні властивості компонента класу `TStringGrid`:

- **`property`** `Cells[ACol, ARow: Integer]:` **`string`**`;` – повертає рядок символів, що міститься у комірці з індексами стовпця `ACol` і рядка `ARow`;
- **`property`** `Cols[Index: Integer]:` **`TStrings`**`;` – повертає список рядків і пов'язаних з ними об'єктів, що містяться у стовпці з індексом `Index`;
- **`property`** `Rows[Index: Integer]:` **`TStrings`**`;` – повертає список рядків і пов'язаних з ними об'єктів, що містяться у рядку з індексом `Index`;
- **`property`** `Objects[ACol, ARow: Integer]: TObject;` – повертає об'єкт, що пов'язаний з рядком символів, який міститься у комірці з індексами `ACol` і `ARow`.

Усі ці властивості доступні під час виконання програми. Задавати тексти можна програмно або для окремих комірок, або відразу по стовпцях і рядках за допомогою методів класу `TStrings`.

Властивості `ColCount` і `RowCount` визначають відповідно число стовпців і рядків, а властивості `FixedCols` і `FixedRows` – число фіксованих стовпців і рядків, які не прокручуються. Колір фону фіксованих комірок визначається властивістю `FixedColor`. Властивості `LeftCol` і `TopRow` визначають відповідно індекси пер-

шого видимого на екрані в цей момент стовпця, що прокручується, і першого видимого рядка, що прокручується.

Властивість `ScrollBars` визначає наявність у таблиці смуг прокручування. Причому смуги прокручування з'являються і зникають автоматично залежно від того, вміщується таблиця у відповідний розмір, чи ні.

Властивість `Options` є множиною, яка визначає такі властивості таблиці:

наявність розділових вертикальних і горизонтальних ліній у **фіксованих** (`goFixedVertLine` і `goFixedHorzLine`) і **не фіксованих** (`goVertLine` і `goHorzLine`) комірках;

- можливість для користувача змінювати за допомогою миші розміри стовпців і рядків таблиці (`goColSizing` і `goRowSizing`);
- можливість для користувача переміщувати стовпці і рядки таблиці (`goCoIMoving` і `goRowMoving`);
- можливість редагувати вміст таблиці (`goEditing`);
- та багато іншого.

Здебільшого компонент `TStringGrid` використовується для вибору користувачем якихось значень, відображених у комірках. Властивості `Col` і `Row` показують індекси стовпця і рядка виділеної комірки. Можливе також виділення користувачем множини комірок, рядків і стовпців.

Серед множини подій компонента `TStringGrid` потрібно відзначити подію `OnSelectCell`**,** яка виникає у момент вибору користувачем комірки. У процедуру опрацювання цієї події передаються параметри `ACol` і `ARow` – стовпець і рядок виділеної комірки, й булівський параметр `CanSelect` – допустимість вибору. Параметр `CanSelect` можна використати для заборони виділення комірки, задавши його значення `false`. А параметри `ACol` і `ARow` можуть використовуватися для якоїсь реакції програми на виділення користувача.

12.1.10. Уведення цілих чисел (компоненти класу TUpDown і TSpinEdit)

У бібліотеці VCL є спеціалізовані компоненти, що забезпечують уведення цілих чисел – компоненти **TUpDown** і **TSpinEdit**. Компонент класу TUpDown перетворює вікно редагування TEdit у компонент, в якому користувач може вибирати ціле число за допомогою кнопок зі стрілками. Якщо до того ж встановити у true значення властивості Readonly вікна редагування, то користувач просто не зможе ввести у вікно свій текст і змушений буде обмежитися вибором числа. Компонент класу **TSpinEdit** – це поєднання компонентів **TEdit** і **TUpDown** в одному компонентові.

Компонент класу **TUpDown** (кнопка-лічильник) міститься на сторінці *Win32* палітри компонентів, а компонент класу **TSpinEdit** (кнопка-лічильник з вікном редагування) – на сторінці *Samples*.

Основна властивість компонента класу **TUpDown** – властивість Associate, що зв'язує компонент з одним із віконних компонентів, зазвичай з Edit. Властивість AlignButton визначає розміщення UpDown щодо вікна редагування: *зліва* (значення udLeft) або *справа* (значення udRight). Властивість Orientation визначає спосіб розміщення кнопок: *горизонтальний* (значення udHorizontal) або *вертикальний* (значення udVertical). Властивість ArrowKeys визначає, чи будуть керувати компонентом клавіші клавіатури зі стрілками.

Властивості Min і Max компонента UpDown задають відповідно мінімальне і максимальне значення чисел, властивість Increment задає приріст числа при кожному натисненні на кнопку. Властивість Position визначає поточне значення числа. Ця властивість використовується для того, щоб дізнатися, яке число задав користувач. Властивість Wrap визначає, як поводиться компонент при досягненні максимального/мінімального значення. Якщо Wrap =false**,** то при збільшенні/зменшенні числа до максимального/мінімального значення це число фіксується на граничному значенні і дальші натискання кнопки ні до чого не приведуть. Якщо ж Wrap=true**,** то спроба перевищити максимальне число веде до його

скидання на мінімальне значення. Аналогічно, спроба зменшити мінімальне число веде до його скидання на максимальне значення.

Якщо у компонента **Edit**, зв'язаного з **UpDown**, властивість `ReadOnly` дорівнює `false`, то користувач зможе редагувати число, не користуючись кнопками зі стрілками. Це зручно, якщо необхідне число є далеко від поточного, а крок приросту `Increment` є малим. Але разом з тим виявляється серйозний недолік компонента **UpDown**: ніщо не заважає користувачеві ввести не цифри, а якісь інші символи.

Властивості компонента класу **TSpinEdit** схожі на розглянуті, тільки мають інші назви: властивості `Min`, `Max` і `Position` названі відповідно `MinValue`, `MaxValue` і `Value`. Загалом компонент **SpinEdit** у багатьох випадках зручніший простого поєднання `UpDown` і `Edit`.

Приклад 12.6. Створимо проект табулювання неперервної на числовій осі функції $y = e^x \cdot \sqrt{x^4 + 8}$ на проміжку $[a; b]$ (див. рис. 12.4).

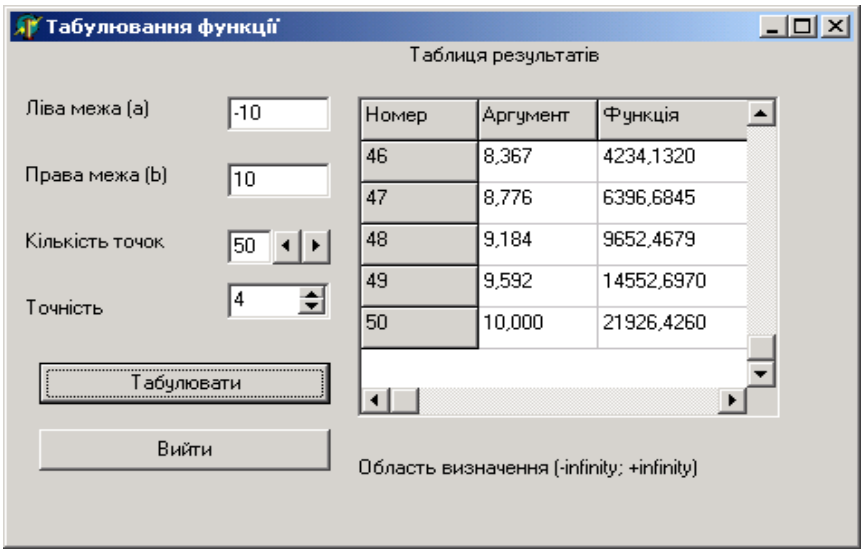


Рис. 12.4. Табулювання неперервної функції

Попередні **зауваження** до проекту:

- для відображення результатів табулювання функції використаємо компонент класу `TStringGrid`;
- **кількість точок** (величина k), у яких обчислюватиметься функція, будемо вибирати за допомогою компонента класу `TUpDown`. Вважаємо, що k може набувати натуральних значень від 2 до 50;
- **точність** (величина n – може набувати цілі значення від 0 до 6) – задає кількість цифр після коми для значення функції. Значення n вибиратимемо за допомогою компонента класу `TSpinEdit`;
- на рівні форми контролюватимемо правильність уведення числових даних для меж табулювання (величини a і b);
- реалізуємо контроль наявності самого факту введення даних;
- реалізуємо контроль за межами табуляції (треба, щоб $a \le b$);
- окремо розглядатимемо випадок, коли $a = b$.

Етап проектування інтерфейсу користувача:

```
Form1: Button1; Button2; Label1; …; Label6;
StringGrid1; UpDown1; SpinEdit1; Edit1; Edit2; Edit3
Form1.Caption:= 'Табулювання функції'
Form1.KeyPreview:= True
Form1.OnCreate:= FormCreate
Form1.OnKeyPress:= FormKeyPres
Button1.Caption:= 'Обчислити'
Button2.Caption:= 'Вийти'
Label1.Caption:= 'Ліва межа (a)'
Label2.Caption:= 'Права межа (b)'
Label3.Caption:= 'Кількість точок'
Label4.Caption:= 'Точність'
Label5.Caption:= 'Таблиця результатів'
Label6.Caption:=
          'Область визначення (-infinity; +infinity)'
// Назви вікон редагування відображають їхнє призначення
Edit1.Name:= 'aEdit';   aEdit.Text = '-10'
Edit2.Name:= 'bEdit';   bEdit.Text = '10'
```

```
Edit3.Name:= 'kEdit'  // поле, пов'язане з UpDown1
kEdit.ReadOnly:= True; kEdit.Text = '50'
SpinEdit1.MaxValue:= 6
SpinEdit1.MinValue:= 0
SpinEdit1.Value:= 2
UpDown1.Associate:= kEdit
UpDown1.Min:= 2;  UpDown1. Max:= 50
UpDown1.Orientation:= udHorizontal
UpDown1.Position:= 50
UpDown1.Wrap:= True
// Властивості компонента StringGrid1
StringGrid1.Name:= 'tab'
tab.BorderStyle:= bsSingle
tab.ColCount:=3;        tab.RowCount:= 51
tab.FixedCols:= 1       tab.FixedRows:= 1
tab.Options:=
             [goFixedVertLine, goFixedHorzLine,
              goVertLine, goHorzLine, goRangeSelect,
              goRowSizing, goColSizing]
```

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.FormKeyPress(Sender: TObject;
                              var Key: Char);
var buf:string[20];
begin
 // Аналіз елемента, що спричинив подію
 if Sender = aEdit
      then  buf:=aEdit.Text
      else  buf:=bEdit.Text;
 // Аналіз натиснутої клавіші
 case key of
  '0'.. '9', chr (8): ; // допустимі символи
  // Символ ',' може бути тільки один!
  ',': if pos(',',buf)<>0 then key:=chr(0);
  // Символ '-' може бути тільки першим!
  '-': if length(buf)<>0 then key:=chr(0);
```

```
  // Завершення введення. Перехід на черговий елемент
  chr(13):
          if Sender=aEdit
             then  bEdit.SetFocus
             else button1.SetFocus;
  else key:=Chr(0) // Блокування недопустимих клавіш
 end
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 Tab.Cells[0,0]:='Номер';
 Tab.Cells[1,0]:='Аргумент';
 Tab.Cells[2,0]:='Функція'
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    a,b,x,y,h:real;
    stx, sty:string;
    i,n,k:integer;
function f(x:real):real;
 begin
   result:=exp(x)-sqrt(Sqr(x)*Sqr(x)+8)
 end;
begin
 if (aEdit.Text='')or(bEdit.Text='') then
  begin // Контроль факту введення меж
   MessageDlg('Уведіть межі!', mtWarning,[mbOk],0);
   exit
  end;

 // Отримання меж
 a:=StrToFloat(aEdit.text);
 b:=StrToFloat(bEdit.text);

 // Очищення рядків таблиці
```

```
 for i:=1 to 50 do Tab.Rows[i].Clear;

 // Отримання кількості точок
 k:=StrToInt(kEdit.Text);

 // Отримання точності
 n:=SpinEdit1.Value;

 // Контроль за узгодженістю меж
 if a>b then
  begin
   ShowMessage('Ліва межа більша за праву!');
   aEdit.AutoSelect:=true;
   aEdit.SetFocus; exit
  end;
 // Ситуація, коли межі збігаються
 if a=b then
  begin
   x:=a;    stx:=FloatToStrF(x,ffFixed,8,3);
   y:=f(x); sty:=FloatToStrF(y,ffFixed,8,n);
   Tab.Cells[0,1]:='1'; Tab.Cells[1,1]:=stx;
   Tab.Cells[2,1]:=sty;  exit
  end;
 // Табулювання функції у k точках
 h:=(b-a)/(k-1); x:=a;
 for i:=1 to k do
  begin
    if x>b then x:=b; // Врахування похибки
    stx:=FloatToStrF(x, ffFixed, 8, 3);
    y:=f(x); sty:=FloatToStrF(y,ffFixed,8,n);
    Tab.Cells[0,i]:=IntToStr(i);
    Tab.Cells[1,i]:=stx;
    Tab.Cells[2,i]:=sty; x:=x+h
  end
end;

procedure TForm1.Button2Click(Sender: TObject);
```

12.2. Перемикачі та кнопки

12.2.1. Компонент класу TButton (командна кнопка)

Компонент класу **TButton** є стандартною командною кнопкою Windows, яка ініціалізує якусь дію. Напис на кнопці задається властивістю Caption. У написі кнопки можна передбачати використання клавіш прискореного доступу, виділяючи для цього один із символів напису (перед таким символом треба поставити символ '&'). Символ '&' не відображається, а черговий за ним символ буде підкреслений. Тоді користувач замість натискання кнопки може у будь-який момент натиснути клавішу Alt разом із клавішею виділеного символу. Основна подія кнопки – OnClick, яка виникає при натисканні кнопки.

Якщо властивість Cancel кнопки дорівнює true, то натискання користувачем клавіші Esc буде еквівалентним натисканню цієї кнопки. Цю властивість доцільно задавати рівною true для кнопки, що скасовує певну дію чи реалізує вихід з діалогового вікна або форми.

Якщо властивість Default кнопки дорівнює true, то натискання користувачем клавіші Enter буде еквівалентним натисканню цієї кнопки. Правда, якщо у момент натискання Enter у фокусі знаходиться інша кнопка, то все-таки спрацює саме кнопка у фокусі.

Ще одна властивість – ModalResult використовується у модальних формах. У звичайних формах значення цієї властивості має дорівнювати mrNone.

З методів “кнопки” є сенс відзначити метод Click. Виконання цього методу еквівалентно автоматичному натисканню кнопки (тобто буде викликана подія кнопки OnClick).

12.2.2. Командна кнопка з написом і піктограмою

Командна кнопка з написом і піктограмою є компонентом класу **TBitBtn**. На поверхні цієї кнопки можна розмістити як напис, так і зображення (піктограму), які задають властивістю Glyph. Ця властивість може містити від одного до чотирьох зображень однакового розміру (від 16×16 до 32×32 пікселів).

Перше зображення, якщо його переглядати зліва направо, відповідає спокійному стану кнопки; друге – відповідає недоступ-

ній кнопці (властивість `Enabled` дорівнює `false`); третє – відповідає натиснутій кнопці; четверте – використовується у кнопках з фіксацією (компонент класу `TSpeedButton`). Зображення можна підготувати у відповідному редакторі (**Picture Editor**), але найліпше скористатися готовим набором, який міститься у каталозі

`C:\Program Files\Common Files\Borland Shared\Images\Buttons`

Число піктограм у зображенні відображається автоматично у властивості `NumGlyphs`. Розміщення зображення і напису (властивість `Caption`) на кнопці визначають властивості `Margin`, `Layout` і `Spacing`. Якщо властивість **`Margin`** дорівнює –1 (значення за домовленістю), то зображення і напис розміщують у центрі кнопки. Положення зображення щодо напису визначається властивістю **`Layout`**, що може набувати значення: `blGlyphLeft` (зліва, це значення прийняте за домовленістю), `blGlyphRight` (праворуч), `blGlyphTop` (угорі), `blGlyphBottom` (унизу). Якщо ж `Margin > 0`, то залежно від значення `Layout` зображення і напис зміщуються до того чи іншого краю кнопки, відступаючи від неї на число пікселів, задане значенням `Margin`.

Властивість **`Spacing`** задає число пікселів, які розділяють зображення і напис на поверхні кнопки (`Spacing=4` – за домовленістю). Якщо задати `Spacing = 0`, зображення і напис будуть розміщені впритул один до одного. Якщо задати `Spacing = -1`, то текст з'явиться між зображенням і краєм кнопки.

Дуже часто у застосуваннях використовують кнопки стандартного призначення: вибір альтернативи **Так** або **Ні**, **Повторити**, **Ігнорувати**, **Допомогти** тощо. У Windows вигляд таких кнопок є зафіксованим, тому `BitBtn` має властивість `Kind`, яка дає змогу вибрати вид кнопки: `bkAll`, `bkCancel`, `bkClose`, `bkHelp`, `bkIgnore`, `bkNo`, `bkOk`, `bkRetry`, `bkYes`. За домовленістю значення властивості дорівнює `bkCustom` – *замовлена* (вигляд кнопки визначає програміст). Інші властивості, методи і події відповідають аналогічним елементам кнопки класу `TButton`.

12.2.3. Компонент класу `TSpeedButton` (швидка кнопка)

Компонент класу **`TSpeedButton`** (швидка кнопка) використовується для створення інструментальних панелей і у випадках, коли потрібно зафіксувати натиснутий стан кнопки. Зауважимо, що командні кнопки класів `TButton` і `TBitBtn` не можна зафіксувати, коли вони натиснуті.

У кнопок класу **`TSpeedButton`**, як і в інших кнопок, є властивість `Caption` – напис, який зазвичай залишається порожнім, тому що замість напису використовується піктограма. Зображення на кнопці задається властивістю `Glyph` так само, як і для кнопок `TBitBtn`. Властивості `NumGlyphs`, `Layout`, `Margin` і `Spacing` відповідають аналогічним властивостям кнопки `TBitBtn`.

Якщо швидкі кнопки об'єднані в групу (шляхом розміщення їх на спільній панелі), то кожній кнопці треба задати груповий індекс (властивість `Groupln dex`), що не дорівнює нулю. Якщо `Groupln dex = 0`, то кнопка поводить себе так само, як і кнопки класу `TButton` чи `TBitBtn`.

Властивість `AllowAllUp` при значенні `true` надає дозвіл на зміну стану кнопки групи, а властивість `Down` при значенні `true` задає початковий стан кнопки – *натиснута*.

Коли `Groupln dex>0` і `AllowAllUp=true`, то кнопка при натисканні користувача утоплюється і залишається у цьому стані. При повторному натисканні кнопки вона звільняється і переходить у звичайний стан.

Якщо є декілька кнопок, що мають однакове ненульове значення `GroupIndex`, то вони утворюють групу *взаємозалежних кнопок*, у якій натиснутою може бути тільки одна кнопка. Коли одна кнопка знаходиться в утопленому стані і користувач натискає іншу кнопку, то перша кнопка звільняється, а друга фіксується у натиснутому стані.

У випадках, коли швидкі кнопки не є взаємозалежними, поводження утопленої кнопки при повторному натисканні її залежить від значення властивості `AllowAllUp` кнопки. Якщо воно дорівнює `true`, то кнопка звільниться. Якщо ж `AllowAllUp` дорівнює `false`, то натискання кнопки не змінить вигляд кнопки. Втім, і в цьому випадку, як

і при будь-якому натисканні кнопки, виникає подія `OnClick`, яку можна опрацювати.

Стан кнопки під час виконання програми можна визначити за значенням властивості `Down`: якщо воно дорівнює `true`, то кнопка натиснута. Під час події `OnClick` значення `Down` уже дорівнює тому стану, якого набуває кнопка у результаті її натискання.

12.2.4. Компонент класу **TRadioButton**

Компонент класу **TRadioButton** (залежний перемикач або радіокнопка) використовується у сукупності з іншими такими ж перемикачами для вибору однієї альтернативи з декількох взаємовиключних альтернатив. Залежний перемикач зазвичай об'єднується у групу щонайменше ще з одним таким же залежним перемикачем.

При ввімкненні одного залежного перемикача групи інші перемикачі автоматично вимикаються. Об'єднання залежних перемикачів у групу здійснюється зазвичай панелями: `TGroupBox`, `TPanel` тощо. Є також компонент групи радіокнопок класу `TRadioGroup`, у який не треба переносити перемикачі – вони там вже є наявні.

Властивість **Caption** компонента `TRadioButton` містить напис, що з'являється поблизу радіокнопки. Значення властивості **Alignment** визначає, з якої сторони від радіокнопки з'явиться напис: `taLeftJustify` – ліворуч, `taRightJustify` – праворуч (це значення прийняте за домовленістю). Властивість **Checked** типу `boolean` визначає, обраний цей перемикач користувачем, чи ні. Оскільки на початку виконання звичайно треба, щоб одна з радіокнопок групи була обрана, то її властивість `Checked` треба встановити у `true` у процесі проектування.

12.2.5. Компонент класу **TRadioGroup**

Компонент класу **TRadioGroup** застосовують для формування групи регулярно розміщених радіокнопок, з яких у будь-який момент часу може бути увімкнена тільки одна.

Напис у лівому верхньому куті панелі визначає властивість `Caption`. Написи кнопок та їхню кількість визначає властивість `Items`, що має тип `TStrings`. Під час проектування завдання значень властивості

`Items` здійснюється редактором списку рядків. Скільки рядків буде у ньому записано, стільки і буде залежних перемикачів. Під час виконання програми формувати список `Items` можна, використовуючи методи і властивості класу `TStrings`. Кнопки можна розмістити у декілька стовпців (не більш 17), задавши властивість `Columns` (за домовленістю `Columns = 1`).

Визначити, яку з кнопок вибрав користувач, можна за властивістю `ItemIndex`, яка вказує індекс обраної кнопки (відлік починають з 0). За домовленістю `ItemIndex = -1`, що означає відсутність обраної кнопки. Якщо треба, щоб у момент виконання якась із кнопок була обрана (це практично завжди необхідно), то треба встановити відповідне значення `ItemIndex` під час проектування. Якщо радіокнопки використовуються для відображення даних, то встановлювати значення `ItemIndex` можна під час виконання програми.

12.2.6. Компонент класу `TCheckBox`

Компонент класу **`TCheckBox`** (індикатор або незалежний перемикач) використовується для включення і виключення певної опції (властивості) чи для ідентифікації стану деякого об'єкта. При кожному клацанні мишею на індикаторі його стан змінюється, проходячи загалом послідовно через три значення: *виділене* (поява чорної галочки), *проміжне* (сіре вікно індикатора і сіра галочка) і *не виділене* (порожнє вікно індикатора). Цим трьом станам відповідає три значення властивості **`State`**: `cbChecked`, `cbGrayed`, `cbUnchecked`. Усі три стани допускаються, якщо значення іншої властивості `AllowGrayed` дорівнює `true`. Якщо ж `AllowGrayed = false` (значення за домовленістю), то допустимими є тільки два стани: виділений і не виділений. І `State`, і `AllowGrayed` можна встановлювати як під час проектування, так у разі виконання програми.

Проміжний стан звичайно використовується, якщо індикатор застосовується для відображення якоїсь характеристики об'єкта, що може бути у трьох різних станах. Наприклад, якщо індикатор має показати, який регістр використовувався при написанні деякого фрагмента тексту, то в випадку, якщо весь текст написаний у верхньому регістрі, індикатор може приймати виділений стан, якщо в нижньому –

не виділений, а якщо використовувалися обидва регістри – проміжний.

Перевіряти стан індикатора можна не тільки за значенням `State`, але і за значенням властивості `Checked`. Якщо `Checked = true`, то індикатор обраний, тобто `State = cbChecked`. Якщо `Checked = false`, то `State` дорівнює `cbUnchecked` або `cbGrayed`. Встановлення `Checked` у `true` під час проектування або виконання програми автоматично переключає `State` у `cbChecked`.

Напис на індикаторі задається властивістю `Caption`, а її розміщення щодо індикатора – властивістю `Alignment`.

Основна подія індикатора – `OnClick`, яка настає при клацанні мишею на компонентові. В процедурі опрацювання цієї події можна аналізувати властивості `Checked` і `State`.

12.2.7. Компонент класу `TCheckListBox`

Компонент класу **`TCheckListBox`** (список рядків з індикаторами) аналогічний компоненту списку рядків `TListBox`, за винятком того, що поряд з кожним елементом знаходиться вікно індикатора, який користувач може вмикати і вимикати, позначаючи елементи списку. Основна властивість компонента `Items`, що має тип `TStrings`. Під час проектування завдання значень властивості `Items` здійснюється редактором списку рядків. У разі виконання програми працювати з цією властивістю можна, користуючись властивостями і методами класу `TStrings`.

Індекс обраного користувачем рядка визначається властивістю `ItemIndex`, яка доступна тільки під час виконання програми. Якщо жодний рядок не обрано, то `ItemIndex = -1`. Початкове значення `ItemIndex` під час проектування задати неможливо (дорівнює –1). Задати цій властивості якесь інше значення можна у процедурі опрацювання події форми `OnCreate`, ввівши, наприклад, оператор:

```
CheckListBoxl.ItemIndex:=0;
```

Властивість `Columns` визначає число стовпців списку, якщо він не міститься цілком у вікні компонента `TCheckListBox`. Якщо властивість `Sorted = true`, нові рядки додаються в список за алфавітом.

Властивість `Style`, встановлена у `lbStandard` (значення за домовленістю), відповідає списку рядків. Інші значення `Style` дають змогу відображати у списку не тільки текст, але і зображення.

Стан індикаторів визначають дві властивості: `State` і `Checked`. Обидві ці властивості можна розглядати як індексовані масиви, кожен елемент яких відповідає індексу рядка. Ці властивості можна встановлювати чи читати програмно. Наприклад, оператори

```
CheckListBoxl.Checked[l]:=true;
CheckListBoxl.State[2]:=cbGrayed;
```

встановлюють індикатор другого рядка у стан обраного, а індикатор третього рядка – у проміжний стан (нагадаємо, що індекси починаються з 0). Оператор:

```
for i:=0 to CheckListBoxl.Items.Count-1 do
  if CheckListBoxl.Checked[i] then …
```

перевіряє стан індикаторів списку, і для обраних користувачем рядків здійснює якісь дії (на місці цих дій поставлено три крапки).

12.2.8. Приклади використання кнопок і перемикачів

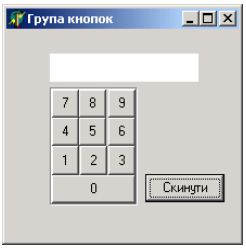


Рис. 12.5. Кнопковий набір числа

Приклад 12.7. **Реалізація кнопкового набору натурального числа**. Розглядається панель мікрокалькулятора (див. рис. 12.5), на якій реалізовані тільки дві функції: **очищення** операнда (позначка `Label1`) і **набір** натурального значення у цьому компонентові за допомогою цифрових кнопок.

Попередні міркування. Для відображення числа вибрано позначку, оскільки тільки вона дає можливість притискати текст до правої межі, як це є у мікрокалькуляторах. Щоб позначка не загубилася на сірому фоні форми, змінимо колір позначки (`Color=clBtnHighlight`), заборонимо підганяти ширину під розмір тексту (`AutoSize=False`) і задамо праве вирівнювання тексту (`Alignment=taRightJustify`).

Спочатку розмістимо на формі компонент класу `GroupBox1` (група елементів). У рамці цього компонента розмістимо десять

командних кнопок (`Button1, …, Button10`) і замінимо їхні написи (`Caption`) на відповідну цифру. Виділимо всі кнопки і на закладці **Events** клацнемо на події `OnClick`, що дасть змогу приписати всім кнопкам спільну процедуру опрацювання цієї події. Одинадцята кнопка відповідатиме за очищення поля позначки.

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.Button10Click(Sender: TObject);
begin
 with Sender As TButton do
  begin
    if not((Label1.Caption='')and(Caption='0')) then
       Label1.Caption:=Label1.Caption + Caption;
  end
end;

procedure TForm1.Button11Click(Sender: TObject);
begin Label1.Caption:=''; end;
```

Зауважимо, що оператор **`if`** у процедурі `TForm1.Button10Click` не допускає набору нуля, як старшої цифри числа.

Приклад 12.8. Демонстрування роботи зі списком класу `TCheck-ListBox`, різноманітними перемикачами і кнопками на прикладі **обчислення** однієї з **величин** (*суми*, *добутку*, *середнього арифметичного*, *мінімуму* або *максимуму*) для виділених елементів списку (див. рис. 12.6).

Етап проектування інтерфейсу проекту:

Form1: `Button1; Label1; Label2; Edit1; Edit2`
`CheckListBox1; GroupBox1 // панель радіокнопок`
`RadioButton1;... RadioButton5;`
`CheckBox1;...; CheckBox3; GroupBox2 //панель індикаторів`
`BitBtn1;...; BitBtn4 //1-'Вийти'; 2 - 'Очистити список'` і т.д.

Для `CheckListBox1`, `CheckBox1`, `CheckBox2`, `CheckBox3` залишимо значення властивості `AllowGrayed` за домовленістю (`false`). Інші властивості компонентів можна задати, дивлячись на рис. 7.6. Початковий стан списку, полів уведення, кнопок і перемикачів задають у процедурі `TForm1.FormCreate`.

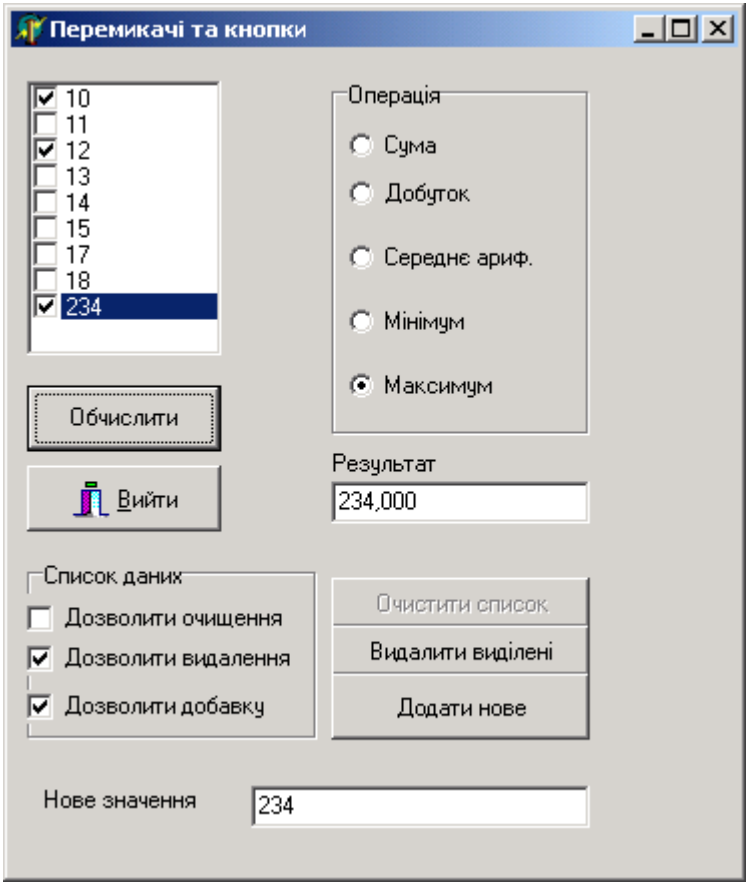


Рис. 12.6. Робота зі списком, перемикачами і кнопками

Етап програмування *процедур опрацювання подій*:

```
procedure TForm1.FormCreate(Sender: TObject);
 var i:integer;
begin // Початкове заповнення списку
 for i:=0 to 9 do
     CheckListBox1.Items.Add(IntToStr(10+i));
// Початкові стани кнопок,перемикачів і полів
 Edit1.ReadOnly:=true;
 Edit2.ReadOnly:=true;
```

```
 Checkbox1.Checked:=False;
 checkbox2.Checked:=False;
 Checkbox3.Checked:=False;
 BitBtn2.Enabled:=False;
 BitBtn3.Enabled:=False;
 BitBtn4.Enabled:=False;
end;
procedure TForm1.BitBtn1Click(Sender: TObject);
begin close end;
procedure TForm1.Button1Click(Sender: TObject);
  var i,n:integer; x:real; s:String;
begin
 if (RadioButton1.Checked) or (RadioButton3.Checked) then
  begin x:=0; n:=0;
   for i:=0 to CheckListBox1.Items.Count-1 do
    if CheckListBox1.Checked[i] then
     begin S:=CheckListBox1.Items[i];
           x:=x+StrToFloat(s); inc(n)
     end
  end;
 if RadioButton2.Checked then
  begin  x:=1; n:=0;
   for i:=0 to CheckListBox1.Items.Count-1 do
    if CheckListBox1.Checked[i] then
      begin S:=CheckListBox1.Items[i];
            x:=x*StrToFloat(s); inc(n)
      end
  end;
 if (RadioButton4.Checked) or (RadioButton5.Checked) then
  begin n:=0;for i:=0 to CheckListBox1.Items.Count-1
              do if CheckListBox1.Checked[i] then inc(n)
  end;
 if n=0 then begin ShowMessage('Нічого не вибрано!');
                   Edit1.Text:=''; exit
             end;
 if RadioButton3.Checked then x:=x/n;
 if RadioButton4.Checked then
  begin i:=-1;
   repeat inc(i) until CheckListBox1.Checked[i];
   s:=CheckListBox1.Items[i]; x:=StrtoFloat(s); inc(i);
```

```
   while i<CheckListBox1.Items.Count do
    begin if CheckListBox1.Checked[i]then
           begin s:=CheckListBox1.Items[i];
             if x>StrtoFloat(s) then x:=StrtoFloat(s)
           end;
     inc(i)
    end
  end;
 if RadioButton5.Checked then
  begin i:=-1;
   repeat inc(i) until CheckListBox1.Checked[i];
   s:=CheckListBox1.Items[i]; x:=StrtoFloat(s); inc(i);
   while i<CheckListBox1.Items.Count do
    begin if CheckListBox1.Checked[i]then
           begin s:=CheckListBox1.Items[i];
             if x<StrtoFloat(s) then x:=StrtoFloat(s)
           end;
     inc(i)
    end
  end;
 Edit1.Text:=FloatToStrf(x,ffFixed,8,3)
end;
procedure TForm1.Edit2KeyPress(Sender: TObject;
var Key: Char); // Перевірка введення чисел
var buf:string[20];
begin buf:=Edit2.Text;
 case key of
  '0'.. '9', chr (8): ; // допустимі символи
  // Символ ',' має бути один!
  ',': if pos(',',buf)<>0 then key:=chr(0);
  // Символ '-' має бути першим!
  '-': if length(buf)<>0 then key:=chr(0);
  // Завершення введення.
  chr(13): bitBtn4.SetFocus;
  else key:=Chr(0) //Блокування недопустимих символів.
 end
end;
procedure TForm1.CheckBox3Click(Sender: TObject);
begin
 if CheckBox3.Checked then
  begin Edit2.ReadOnly:=False; BitBtn4.Enabled:=true end
```

```
 else begin
       Edit2.ReadOnly:=true; BitBtn4.Enabled:= False
      end
end;
procedure TForm1.BitBtn4Click(Sender: TObject);
begin
 if Edit2.Text='' then
  begin MessageDlg('Задайте всі коеф.!',mtWarning,
[mbOk],0);
        exit end;
 CheckListBox1.Items.Add(Edit2.Text)
end;
procedure TForm1.CheckBox2Click(Sender: TObject);
begin
 if CheckBox2.Checked
    then  BitBtn3.Enabled:=true
    else  BitBtn3.Enabled:=False
end;
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
 if CheckBox1.Checked
    then  BitBtn2.Enabled:=true
    else  BitBtn2.Enabled:=False
end;
procedure TForm1.BitBtn3Click(Sender: TObject);
var i,n:integer;
begin n:=0;
 for i:=0 to CheckListBox1.Items.Count-1 do
   if CheckListBox1.Checked[i] then inc(n);
 if n=0
   then begin ShowMessage('Нічого не вибрано!');
              Edit1.Text:='';  exit
        end
   else
    for i:=0 to CheckListBox1.Items.Count-1 do
     if CheckListBox1.Checked[i]
        then CheckListBox1.Items.Delete(i);
end;
end.
```

12.3. Діалогові вікна

12.3.1. Загальні положення

У програмах часто доводиться виконувати стандартні дії: відкривати та зберігати файли, задавати атрибути шрифтів, вибирати кольори палітри та інше. Розробники VCL подбали про те, щоб включити у бібліотеку прості для використання компоненти, які реалізують відповідні діалогові вікна. Для цього у VCL є спеціальні класи, з одним спільним предком – класом **`TCommonDialog`**, який є прямим нащадком класу **`TComponent`**. Діалогові вікна є прикладами невізуальних компонентів. Вони розміщені на сторінці `Dialogs` палітри компонентів:

Компонент	Опис
`OpenDialog`	Призначений для створення вікна діалогу “*Відкрити файл*”
`SaveDialog`	Призначений для створення вікна діалогу “*Зберегти файл*”
`OpenPictureDialog`	Призначений для створення вікна діалогу “*Відкрити малюнок*”, який відкриває графічний файл
`SavePictureDialog`	Призначений для створення вікна діалогу “*Зберегти малюнок*”, який зберігає зображення у графічному файлі
`FontDialog`	Призначений для створення вікна діалогу “*Шрифти*” – вибір атрибутів шрифту
`ColorDialog`	Призначений для створення вікна діалогу “*Колір*” – вибір кольору
`PrintDialog`	Призначений для створення вікна діалогу “*Друк*”
`PrinterSetupDialog`	Призначений для створення вікна діалогу “*Встановлення принтера*”
`FindDialog`	Призначений для створення вікна діалогу “*Знайти*” – контекстний пошук у тексті
`ReplaceDialog`	Призначений для створення вікна діалогу “*Замінити*” – контекстна заміна тексту

На сторінці Win 3.1 палітри компонентів можна знайти компоненти – фрагменти діалогів, які дають змогу будувати власні діалогові вікна:

FileListBox	Відображає список усіх файлів каталогу
DirectoryListBox	Відображає структуру каталогів диска
DriveComboBox	Випадаючий список доступних дисків
FilterComboBox	Випадаючий список фільтрів пошуку файлів

На сторінці Samples палітри компонентів можна знайти компонент CDirectoryOutline, який використовується для відображення структури каталогів.

Основний метод, яким здійснюється виклик діалогу:

```
function Execute: Boolean; virtual; abstract;
```

– відкриває діалогове вікно і, якщо користувач зробив у ньому деякий вибір, то метод повертає значення true. Водночас у властивостях компонента-діалогу запам'ятовується вибір користувача, який можна прочитати та використати у дальших операціях. Якщо ж користувач у діалозі натиснув кнопку Cancel (*Скасувати*) або кнопку Esc, то метод Execute поверне false. Тому стандартне звертання до діалогу має такий вигляд:

```
If назва_компонента.Execute then оператор_вибору;
```

Щоб закрити діалогове вікно програмно, потрібно застосувати метод CloseDialog:

```
назва_компонента.CloseDialog;
```

Якщо необхідно маніпулювати позицією діалогового вікна в реальному часі, потрібно використовувати такі властивості, як Handle, Left, Top і Position. Всі діалоги, розміщені на панелі Dialogs, окрім PrinterDialog, мають властивість Options, яка впливає на вигляд і можливості діалогового вікна. Щоб побачити різноманітні опції, потрібно в інспекторі об'єктів два рази клацнути мишею на Options.

12.3.2. Компоненти класів `TOpenDialog` і `TSaveDialog`

Компонент класів **`TOpenDialog`** і **`TSaveDialog`** – це компоненти, які використовують найчастіше.

Компонент класу **`TOpenDialog`** зазвичай використовують за допомогою команд меню `New` або `Open` у пункті меню `File`. Компонент класу **`TSaveDialog`** здебільшого використовують команди меню `Save As` або `Save` (при збереженні нового файлу) у пункті меню `File`.

Після натискання кнопки `Open`, яка знаходиться у діалоговому вікні `OpenDialog`, діалог автоматично закривається. Властивість, в якій повертається назва вибраного файлу, – `FileName`. Значення цієї властивості можна задати перед звертанням до діалогу:

```
OpenDialog1.FileName='test.cpp';
```

Це значення з'явиться у діалоговому вікні як значення назви файлу за домовленістю.

Аналогічно, після натискання кнопки `Save`, яка знаходиться у діалоговому вікні `SaveDialog`, діалог автоматично закривається. Властивість, в якій задається назва збереженого файлу, – `FileName`. Значення цієї властивості можна задати перед звертанням до діалогу:

```
SaveDialog1.FileName='test.cpp';
```

Це значення з'явиться у діалоговому вікні як значення назви файлу за домовленістю.

Типи потрібних файлів, які з'являються у випадаючому списку *Тип файлу* вікна діалогу, задають за допомогою властивості `Filter`. У процесі проектування цю властивість найлегше задати за допомогою **редактора фільтрів (**викликається натисканням кнопки з трьома крапками у інспекторі об'єктів):

- на його лівій панелі `Filter Name` записують текст, який побачить користувач у випадаючому списку *Тип Файлу*;
- на правій панелі `Filter` записують шаблони фільтра, які в одному рядку розділені крапками з комою (наприклад: `.txt; .doc`). Позначення `*.*` означає – *усі типи* файлів.

Після виходу з вікна редактора фільтрів задані тексти і шаблони з'являться у властивості `Filter` у вигляді рядка, в якому вони розділені вертикальними лініями. Аналогічно, якщо потрібно, можна задати властивість `Filter` під виконання програми.

Властивість `FilterIndex` визначає номер фільтра, який буде за домовленістю показаний у момент відкриття діалогу.

Властивість `InitialDir` визначає початкову директорію, що буде відкрита у момент початку роботи користувача з діалогом. Якщо значення не задане, то відкривається поточний каталог чи той, який був відкритий при останньому звертанні користувача до каталогу у процесі виконання програми.

Властивість `DefaultExt` визначає значення розширення файлу за домовленістю. Якщо значення не задане, користувач повинен вказати у діалозі повну назву файлу з розширенням. Якщо ж задати значення `DefaultExt`, то користувач може писати у діалозі назву без розширення. У цьому випадку буде прийнято задане розширення.

Властивість `Title` дає змогу задати заголовок діалогового вікна. Якщо ця властивість не задана, вікно відкривається заголовком, який визначений у системі.

Властивість `Options` визначає умови вибору файлу:

`ofAllowMultiSelect`	Дає змогу користувачу вибирати декілька файлів
`OfCreatePromt`	Якщо користувач написав назву файлу, якого нема, з'являється зауваження та запит, чи треба створити файл із заданою назвою
`OfEnableIncludeNotify`	Дає змогу посилати повідомлення у діалог
`OfEnableSizing`	Дає можливість користувачу змінювати розмір діалогового вікна
`OfExtentsionDifferent`	Показує, що розширення вибраного файлу відрізняється від `DefaultExt`

`OfFileMustExit`	У випадку, якщо користувач написав назву файлу, якого немає, з'являється повідомлення про помилку
`OfHideReadOnly`	Видаляє з діалогу індикатор "*Відкрити тільки для читання*"
`OfNoChangeDir`	Після натискання на кнопку `OK` відтворюється поточний каталог, незалежно від того, який каталог був відкритий при пошуку файлу
`OfNoDereferenceLinks`	Забороняє перевизначити кнопки швидкого доступу у діалоговому вікні
`OfNoLongNames`	Відображається не більше восьми символів назви
`OfNoNetworkButton`	Забирає з діалогового вікна кнопку пошуку у мережі. Діє, якщо ввімкнено `ofOldStyleDialog`
`ofNoReadOnly`	Якщо користувач вибрав файл тільки для читання, то генерується повідомлення про помилку
`OfNoTestFileCreate`	Забороняє вибір у мережі захищених файлів та недосяжних дисків при збереженні файлу
`OfNoValidate`	Не дає змоги писати у назвах файлів заборонені символи, але не заважає вибирати файли з забороненими символами
`OfOldStyleDialog`	Створює діалог вибору файлу у старому стилі
`ofOverwritePromt`	Якщо при збереженні файлу користувач написав назву наявного файлу, з'являється зауваження, що файл з такою назвою вже є, і здійснюється запит про перезапис файлу
`OfPathMustExit`	Генерує повідомлення про помилку, якщо вказано каталог, якого немає

`OfReadOnly`	За домовленістю встановлює індикатор “*Відкрити тільки для читання*” при відкритті діалогу
`OfShareAware`	Ігнорує помилки порушення умов колективного доступу та дає змогу незалежно від них здійснювати вибір
`OfShowHelp`	Відтворює у діалозі кнопку “*Довідка*”

За домовленістю усі зазначені опції, крім `ofHideReadOnly`, вимкнені. Якщо дозволено відмічати декілька файлів за допомогою опції `ofAllowMultiSelect`, то список вибраних файлів можна прочитати у властивості `Files` типу `TStrings`.

У компонентах діалогів відкриття та збереження файлів передбачена можливість опрацювання низки подій. Таке опрацювання може бути використане, якщо розглянутих вище опцій не вистачає, щоб встановити обмеження на вибір файлу конкретною програмою. Подія `OnCanClose` виникає при нормальному закритті користувачем діалогового вікна після вибору файлу. В разі відмови користувача від діалогу (натисканням кнопки “*Відміна*” чи клавіші `Esc`) подія `OnCanClose` не настає. У процедурі опрацювання події `OnCanClose` можна додатково перевірити вибраний файл і, якщо він не відповідає умовам, то можна повідомити про це користувачу та задати значення `false` переданому параметру `CanClose`, що не дасть змоги користувачу закрити діалогове вікно.

Можна також написати процедури опрацювання подій `OnFolderChange` – зміна каталогу, `OnSelectionChange` – зміна назви файлу, `OnTypeChange` – зміна типу файлу. У цих процедурах можна передбачити повідомлення користувачу.

12.3.3. Компоненти класів `TOpenPictureDialog` і `TSavePictureDialog`

Властивості компонентів класів **`TOpenPictureDialog`** і **`TSavePictureDialog`** не відрізняються від властивостей компонентів класів `TOpenDialog`, `TPictureDialog`. Єдина відмінність полягає у використанні шаблонів фільтрів для відкриття/збережен-

ня графічних файлів у властивості `Filter`. У цих компонентах використовують такі фільтри:

All	*.jpg; *.jpeg; *.bmp; *.ico; *.emf; *.wmf
Jpeg Image File	*.jpg; *.jpeg
Bitmaps	*.bmp
Icons	*.ico
Enhanced Metafiles	*.emf
Metafiles	*.wmf

У компонентах діалогів відкриття та збереження графічних файлів передбачена можливість опрацювання низки подій, які аналогічні розглянутим попередньо (див. 12.3.2).

Приклади застосування компонентів – діалогових вікон будуть розглянуті згодом.

12.4. Стандартні діалоги

У проектах часто доводиться відображати різні прості діалогові вікна, щоб дати користувачеві якісь вказівки або задати нескладне питання, на яке можлива одна зі стандартних відповідей: *так*, *немає*, *відмінити*, *перервати*. Розробляючи проект, зручно користуватися готовими стандартними діалогами, які оформляють у вигляді **стандартних процедур** і/або **функцій**. Найпростішою з таких підпрограм є процедура:

```
procedure ShowMessage(const Msg: string);
```

що відображає вікно повідомлення з кнопкою OK. Текст повідомлення задається параметром `Msg`. Заголовок вікна збігається з назвою файлу проекту, що виконується. Подібно працюють і дві інші процедури:

```
procedure ShowMessageFmt(const Msg: string; Params:
                         array of const);
```

– виводить форматований рядок повідомлення `Msg`, у який при виведенні вставляються параметри з `Params`;

```
procedure ShowMessagePos(const Msg: string; X, Y:
Integer);
```

– виводить рядок повідомлення `Msg` у позицію з координатами лівого верхнього кута (`X, Y`).

Подальші функції відображають вікна, в яких користувачеві задається питання і аналізується отримана відповідь. Основна з цих функцій:

```
function MessageDlg(const Msg: string; DlgType:
TMsgDlgType; Buttons: TMsgDlgButtons; HelpCtx:
Longint): Word;
```

Виклик `MessageDlg` відображає діалогове вікно і чекає відповіді користувача. Повідомлення у вікні задається параметром функції `Msg`. Вигляд вікна задається параметром `DlgType` типу множини `TMsgDlgType`, яка містить такі значення:

Значення	**Опис**
`mtWarning`	Вікно зауважень: містить жовтий окличний знак
`mtError`	Вікно помилок: містить червоний стоп-сигнал
`mtInformation`	Інформаційне вікно: містить блакитний символ “i”
`mtConfirmation`	Вікно підтвердження: містить зелений знак питання
`mtCustom`	Вікно без малюнка. Заголовок відповідає назві файлу додатка, що виконується

Параметр `Buttons` визначає, які кнопки будуть наявні у вікні. Тип `TMsgDlgBtns` параметра `Buttons` є множиною, що включає позначення різних кнопок. При натисненні кнопки функція повертає зафіксовану константу, яка відповідає натиснутій кнопці. Можливі значення типів кнопок:

Кнопка вікна	**Натиснута кнопка**	**Опис**
1	2	3
`mbYes`	`mrYes`	Кнопка з написом “*Yes*”
`mbNo`	`mrNo`	Кнопка з написом “*No*”

1	2	2
mbOK	mrOK	Кнопка з написом “*OK*”
mbCancel	mrCancel	Кнопка з написом “*Cancel*”
mbAbort	mrAbort	Кнопка з написом “*Abort*”
mbRetry	mrRetry	Кнопка з написом “*Retry*”
mbIgnore	mrIgnore	Кнопка з написом “*Ignore*”
mbAll	mrAll	Кнопка з написом “*All*”
mbNoToAll	mrNoToAll	Кнопка з написом “*NoToAll*”
mbYesToAll	mrYesToAll	Кнопка з написом “*YesToAll*”

Необхідні кнопки заносять у Buttons операцією [], оскільки параметр Buttons є множиною (наприклад, [mbAll, mbNo]). Якщо не занести у цей параметр нічого, у вікні не буде жодної кнопки і користувачеві доведеться закривати вікно системними кнопками Windows.

Крім множини значень, які відповідають окремим кнопкам, визначені три константи, які аналогічні найуживанішим комбінаціям кнопок:

mbYesNoCancel	Вводить у вікно кнопки Yes, No і Cancel
mbOkCancel	Вводить у вікно кнопки OK і Cancel
mbAbortRetryIgnore	Вводить у вікно кнопки Abort, Retry та Ignore
mbAbortIgnore	Вводить у вікно кнопки Abort та Ignore

Ці константи мають тип TMsgDlgButtons і можуть безпосередньо включатися у виклик функції замість параметра Buttons.

Параметр HelpCtx визначає екран контекстної довідки, відповідний діалоговому вікну. Цей екран довідки буде з’являтися при натисненні користувачем клавіші F1. Якщо довідку не заплановано, то при виклику MessageDlg треба задати нульове значення параметра HelpCtx.

Функція MessageDlgPos у всьому подібна до MessageDlg, але відображає діалогове вікно повідомлення у заданому місці екрана. Координати лівого верхнього кута діалогового вікна визна-

чають параметри X і Y. Інші параметри тотожні функції MessageDlg:

```
function MessageDlgPos(const Msg: string; DlgType:
TMsgDlgType; Buttons: TMsgDlgButtons; HelpCtx:
Longint; X, Y: Integer): Word;
```

Приклад використання MessageDlg:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 if MessageDlg('Вас вітає автор! Вийти звідси?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes then
  begin
   MessageDlg('Починає роботу програма!',
              mtInformation, [mbOk], 0); Close
  end
end;
```

Функція **CreateMessageDialog** дає змогу створити діалогове вікно повідомлення у вигляді об'єкта форми. Функція тільки створює вікно, але не відображає його. Відображення здійснюється звичайними для форм методами Show або ShowModal. Однією з переваг створення об'єкта діалогового вікна є те, що в ньому, як в звичайній формі, можна задати кириличний текст заголовка. У попередніх функціях це було неможливо. Недоліком застосування функції CreateMessageDialog є те, що об'єкт діалогового вікна зберігається в пам'яті весь час, поки він не буде знищений явно методом Free. Це призводить до непродуктивних витрат пам'яті.

Функція MessageBox є методом змінної Application типу TApplication. Але оскільки тематично цей метод дуже близький розглянутим вище функціям, обговоримо його в цьому параграфі.

Метод MessageBox відображає діалогове вікно із заданими кнопками, повідомленням і заголовком і дає змогу проаналізувати відповідь користувача. У багатьох випадках це вікно подібне вікнам, що створюються функціями MessageDlg і CreateMessageDialog. Заголовок вікна може бути написаний кирилицею; є також написи російською мовою на кнопках.

Функція `MessageBox` інкапсулює функцію `MessageBox` API Windows. Параметр `Text` – це текст повідомлення, яке може перевищувати 255 символів. Для довгих повідомлень здійснюється автоматичне перенесення тексту. Параметр `Caption` – це текст заголовка вікна. Він також може перевищувати 255 символів, але не переноситься. Параметр `Flags` – множина індикаторів, що визначають вигляд і поведінку діалогового вікна. Конкретні значення індикаторів та приклади застосування `MessageBox` можна знайти у електронній довідці.

12.5. Компоненти меню

12.5.1. Компонент класу `TMainMenu`

Компонент класу **`TMainMenu`** визначає головне меню форми. Після встановлення компонента на форму потрібно створити його опції. Для цього необхідно натиснути на компонентові правою кнопкою миші і вибрати `Menu Designer` у контексному меню.

Утворення опцій не спричиняє проблем. Перейдемо до вікна інспектора об’єктів і введемо текст опції у рядок `Caption`, після чого натиснемо `Enter` – опція готова і можна переходити до чергової.

Кожна опція головного меню може розкриватися у список підопцій або містити команду. Для створення підопцій клацнемо мишкою на рядку нижче опції і введемо першу підопцію. Будемо продовжувати введення, поки не буде створений увесь список підопцій, після чого треба клацнути на порожньому прямокутнику справа від першої опції і ввести другу опцію. Процес набагато складніше описати, ніж виконати.

У назвах опцій можна вказати символ “&” перед символом, який визначить клавішу швидкого вибору опції. Для встановлення розділювальної риски, що відокремлює групи підопцій, треба черговий елемент меню назвати “-” (дефіс). Для створення розгалужених меню, тобто таких, у яких підопції викликають нові списки підопцій, треба клацнути на підопції і натиснути Clrl+“→”.

Кожен елемент меню є об’єктом класу `TMenuItem`. Деякі важливі властивості цього класу описані у таблиці:

property Bitmap:TBitmap;	Містить посилання на зв'язане зображення, яке з'являється ліворуч від опції. Властивість ігнорується, якщо встановлена властивість ImageIndex
property Checked: Boolean;	Якщо True, поряд з опцією з'являється галочка
property Count: Integer;	Містить кількість опцій у підлеглому меню, зв'язаному з цим елементом
property ImageIndex: Integer;	Містить індекс зв'язаного з опцією зображення з компонента TimageList. Це зображення з'являється ліворуч від опції
property Items [Index: Integer]:MenuItem;	Дає змогу звернутися до будь-якої опції підлеглого меню за її індексом
property MenuIndex: Integer;	Визначає індекс опції у списку Items батьківської опції

12.5.2. Компонент класу **TPopupMenu**

Компоненти класу **TPopupMenu** використовуються для створення контекстозалежних меню, що з'являються після натискання правої кнопки миші. Це меню можна створити для будь-якого віконного компонента. Щоб зв'язати клацання правої миші на компонентові з розкриттям контекстозалежного меню, у властивість PopupMenu компонента необхідно помістити назву меню. Контекстозалежне меню створюється за допомогою конструктора меню і містить елементи класу TMenuItem, тому процес створення і властивості такого меню не відрізняються від TMainMenu.

Головна подія для обох типів меню – подія OnClick, яка дає змогу викликати відповідну процедуру опрацювання опції меню.

12.5.3. Компонент класу **TImageList**

Компонент класу **TImageList** є контейнером для збереження малюнків однакового розміру і може бути корисним при програмному створенні набору інструментальних кнопок, і там, де потрібний індексований доступ до зображень. Компонент не має власної канви і тому не може відображати збережені у ньому малюнки. Його метод Draw одержує канву від іншого компонента.

12.6. Використання меню та діалогових вікон на прикладі побудови текстового редактора

Продемонструємо використання меню та діалогових вікон на прикладі **однодокументного текстового редактора**, який даватиме змогу створювати нові файли, відкривати вже наявні, редагувати і зберігати їх, а також використовувати буфер обміну для роботи з фрагментами тексту. Поряд з використанням раніше описаних компонентів, у цьому параграфі ознайомимося з новими компонентами класів `TStatusBar` (рядок стану), `TToolBar` (панель інструментів), `TActionList` (список акцій); використанням підпорядкованих форм тощо.

12.6.1. Початкове розміщення компонентів на формі

Розміщуємо на формі компонент `Memo1` з властивостями: `Align:=alClient`; `ScrollBars:=ssBoth`; `Lines:= ''`. У компонентові `Memo1` будемо переглядати і редагувати текст.

Зі сторінки Win32 розмістимо на формі компонент **`StatusBar1`**. Відредагуємо його властивість `Panels` (ця властивість є набором компонентів – *панелей*, на яких виводиться потрібна користувачеві інформація). Редактором цієї властивості є діалогове вікно, яке відкриється після натискання кнопки з трикрапкою.

Створимо панель, на якій буде з'являтися назва редагованого файлу. Для цього у діалоговому вікні властивості `Panels` натиснемо кнопку `Add`. Для нової панелі встановимо властивості:

`Width:=100`; `Text:= 'Без назви'`

Після цього натискаємо кнопку `OK`.

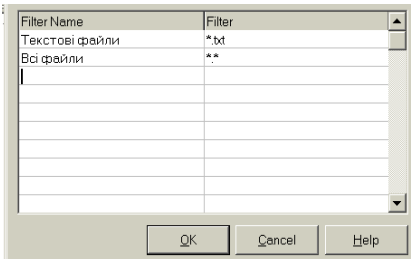


Рис. 12.7. Редактор фільтрів

На формі розміщуємо компонент **`ToolBar1`** (зі сторінки Win32) і діалоги: **`OpenDialog1`** і **`SaveDialog1`** (`DefaultExt:='txt'`). Типи файлів у діалогах задають у властивості `Filter`, яку заповнюють за допомогою редактора фільтрів (див. рис. 12.7).

12.6.2. Компонент класу `TActionList`

Для опрацювання одних і тих же подій різних інтерфейсних елементів (натискання різноманітних кнопок, виконання команд головного і контекстозалежних меню, натискання кнопок на панелі інструментів, натискання швидких клавіш тощо) використаємо компонент класу `TActionList` зі сторінки Standard, який містить колекцію компонентів класу `TAction` (акцій або дій).

Конкретний об'єкт класу `TAction` містить процедуру опрацювання події `OnExecute`. З цією процедурою можна пов'язати довільний елемент керування, що має властивість (і одночасно подію) `Action`. Щоб наголосити на важливості цієї властивості/події, у інспекторі об'єктів її виділяють червоним кольором. Значенням властивості/події `Action` елемента керування і є назва процедури опрацювання події `OnExecute` деякої акції.

На компонентові класу `TActionList` двічі клацнувши мишею, потрапимо у редактор акцій (дій). Натискання правої кнопки миші або вибір маленької кнопки зі стрілкою вниз дасть змогу вибрати одну з команд: `New Action` (нова акція) або `New Standard Action` (нова стандартна акція). Перша з них вводить нову подію будь-якого типу. За домовленістю ці події називаються `Action1`, `Action2` і т. д. Інша команда відкриває вікно, в якому можна вибрати необхідні стандартні дії. Після цього у правому вікні (`Actions`) редактора з'являються назви вибраних дій, а у лівому (`Categories`) – категорії дій.

Для кожної акції можна задати властивості: напису (`Caption`); швидких клавіш (`ShortCut`); написи на ярличках підказок і у рядку стану (`Hint`); ідентифікатор теми контекстної допомоги (`HelpContext`), доступність акції (`Enabled`) та її видимість (`Visible`), індекс зображення (`ImageIndex`), що зв'язаний з акцією. На сторінці подій інспектора об'єктів для кожної акції визначено три події: `OnExecute`, `OnUpdate`, `OnHint`. Усі акції у проекті введемо командою `New Action`, хоча акції роботи з буфером можна ввести командою `New Standard Action`.

Для нашого проекту створюємо **дев'ять** акцій:

Name	**Caption/Hint**	**ImageIndex**	Назва процедури **OnExecute**
acExit	Вихід	0	acExitExecute
acNew	Створити	1	acNewExecute
acOpen	Відкрити	2	acOpenExecute
acSave	Зберегти	3	acSaveExecute
acSaveAs	Зберегти як…	4	acSaveAsExecute
acCut	Вирізати	5	acCutExecute
acCopy	Копіювати	6	acCopyExecute
acPaste	Вставити	7	acPasteExecute
acSelAll	Виділити все	8	acSelAllExecute

12.6.3. Заповнення панелі інструментів кнопками

За допомогою миші вибираємо компонент ToolBar1 і розміщуємо на ньому дванадцять компонентів класу TToolButton (дев'ять активних і три пасивні кнопки панелі інструментів). Активні кнопки можна розмістити на панелі за допомогою команди New Button з контекстного меню компонента TToolBar1.

Кнопки, які мають певну функціональну спорідненість, можна об'єднати у логічні групи (наприклад, група кнопок для роботи з буфером обміну, група кнопок для роботи з файлами тощо). Логічні групи кнопок відділяються одна від одної за допомогою пасивних кнопок (або кнопок-розділювачів). Пасивні кнопки можна розмістити на панелі за допомогою команди New Separator контекстного меню компонента TToolBar1.

Створимо на панелі чотири логічні групи кнопок: 1-ша група – одна кнопка виходу з програми; 2-га група – чотири кнопки для

роботи з файлами; 3-тя група – три кнопки для роботи з буфером обміну; 4-та група – одна кнопка для виділення всього тексту.

Отже, кнопками-розділювачами будуть кнопки: `ToolButton2`; `ToolButton7` і `ToolButton11`. Надамо кнопкам мнемонічні назви і під'єднаємо до них потрібні акції:

- `ToolButton1.Name:=tbExit`; `tbExit.Action:=acExit`
- `ToolButton2.Name:=tbSepar1`
- `ToolButton3.Name:=tbNew`; `tbNew.Action:=acNew`
- `ToolButton4.Name:=tbOpen`; `tbOpen.Action:=acOpen`
- `ToolButton5.Name:=tbSave`; `tbSave.Action:=acSave`
- `ToolButton6.Name:=tbSaveAs`; `tbSaveAs.Action:=acSaveAs`
- `ToolButton7.Name:=tbSepar2`
- `ToolButton8.Name:=tbCut`; `tbCut.Action:=acCut`
- `ToolButton9.Name:=tbCopy`; `tbCopy.Action:=acCopy`
- `ToolButton10.Name:=tbPaste`; `tbPaste.Action:=acPaste`
- `ToolButton11.Name:=tbSepar3`
- `ToolButton12.Name:=tbSelAll`; `tbSelAll.Action:=acSelAll`

Для активних кнопок встановимо властивість `ShowHint:=True`.

12.6.4. Формування набору зображень

На створених кнопках доцільно розмістити піктограми. Для цього розмістимо на формі компонент `ImageList1`, який зберігатиме посилання на набір зображень (піктограм) однакових розмірів. Конкретну піктограму ідентифікують за допомогою індекса (індексація зображень починається з нуля).

Посилання на зображення у компонентові `ImageList1` формується за допомогою редактора списків зображень (*ImageList Editor*), який викликають подвійним клацанням миші на цьому компонентові або командою контекстного меню `ImageList Editor`. За допомогою кнопки `Add` додаємо у список посилання на потрібні файли зображень з набору зображень Borland (каталог ...**Program Files\CommonFiles\Borland Shared \Images\Buttons**): `Doorshut`, `Filenew`, `Fileopen`, `Fileclose`, `Filesave`, `Cut`, `Copy`, `Paste`, `Fldmany` (файли з розширенням .bmp, перераховані за зростанням індексу від 0 до 8).

Кожне зображення містить дві суміщені піктограми. Тому спочатку у додатковому діалоговому вікні дозволяємо розділити зображення на дві окремі піктограми. Після цього у списку зображень отримуємо дві схожі піктограми (одна дещо приглушена, має сірий відтінок). Щоб не порушити індексування зображень, приглушену піктограму видаляємо зі списку (виділяємо піктограму і натискаємо кнопку `Delete`).

Після цього зв'яжемо компоненти `ActionList1` і `ToolBar1` з компонентом `ImageList1`. Для цього:

```
ActionList1.Images:= ImageList1
ToolBar1.Images:= ImageList1
```

Для кожної акції класу `TAction` кнопки класу `TToolButton`, виберемо відповідну піктограму за допомогою властивості `ImageIndex`.

12.6.5. Формування головного меню форми

Зі сторінки Standart перенесемо на форму компонент `MainMenu1`. Його основна властивість – `Items`. Заповнення цієї властивості відбувається за допомогою *Конструктора Меню*, який викликаємо подвійним клацанням миші на компоненті `MainMenu1` або натискаючи кнопку з трикрапкою поблизу властивості `Items` у вікні інспектора об'єктів.

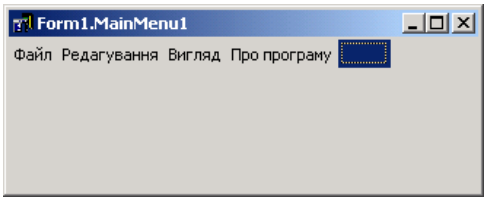


Рис. 12.8. Опції головного меню

Процес створення опцій і підопцій головного меню форми (компонент `MainMenu1`) детально описано у пункті 12.5.1. Опції головного меню зображено на рисунку 12.8. Зауважимо, що кожна опція чи підопція є об'єктом класу `TMenuItem`, які мають стандартні назви `N1`, `N2` і т.д. Замінимо ці назви на більш осмислені:

- `N1.Name:= miFile; miFile.Caption:= 'Файл'`
- `N2.Name:= miEdit; miEdit.Caption:= 'Редагування'`
- `N3.Name:= miView; miView.Caption:= 'Вигляд'`
- `N4.Name:= miAbout; miAbout.Caption:= 'Про програму'`

Для створення підопцій опції **Файл** головного меню скористаємося відповідними акціями:

- `N1.Action:= acNew; N1.Caption:= 'Створити'`
- `N2.Action:= acOpen; N2.Caption:= 'Відкрити'`
- `N3.Action:= acSave; N3.Caption:= 'Зберегти'`
- `N4.Action:= acSaveAs; N4.Caption:= 'Зберегти як … '`
- `N5.Caption:= '-' // розділювач меню`
- `N6.Action:= acExit; N6.Caption:= 'Вихід'`

Замінимо назви підопцій меню на більш осмислені:

```
N1.Name:= miFileNew;   N2.Name:= miFileOpen;
N3.Name:= miFileSave;  N4.Name:= miFileSaveAs;
N5.Name:= miFileSepar; N6.Name:= miFileExit
```

Проектуючи підопції, використаємо властивість `ShortCut`, що визначає клавіші швидкого доступу до підопції, за допомогою яких користувач, навіть не заходячи у меню, може у будь-який момент викликати виконання процедури, зв'язаної з цією підопцією. Для цього потрібно відкрити випадаючий список властивості `ShortCut` у інспекторі об'єктів і вибрати з нього потрібну комбінацію клавіш:

```
miFileNew.ShortCut ⇗ Ctrl+N
miFileSave.ShortCut ⇗ Ctrl+S
```

Властивість `Bitmap` містить графічне зображення, яке з'являється на етапі виконання проекту зліва близько пункту (опції чи підопції) меню. Для завантаження зображення (файл з розширенням `.bmp`) є відповідний редактор.

Створюючи підопції для опції **Редагування** головного меню скористаємося відповідними акціями:

- `N1.Action:=acCut; N1.Caption:= 'Вирізати'`
- `N2.Action:=acCopy; N2.Caption:= 'Копіювати'`
- `N3.Action:=acPaste; N3.Caption:= 'Вставити'`
- `N4.Caption:= '-' // розділювач меню`
- `N5.Action:=acSelAll; N5.Caption:= 'Виділити все'`

Замінимо назви підопцій меню на більш осмислені:

```
N1.Name:= miEditCut; N2.Name:= miEditCopy;
```

```
N3.Name:= miEditPaste; N4.Name:= miEditSepar;
N5.Name:= miEditSelAll
```

Визначаємо клавіші швидкого доступу до підопцій:

```
miEditCut.ShortCut ⇗ Ctrl+X
miEditCopy.ShortCut ⇗ Ctrl+C
miEditPaste.ShortCut ⇗ Ctrl+V
miEditSelAll.ShortCut ⇗ Ctrl+A
```

Опція меню Вигляд головного меню містить дві підопції, які приховують/відновлюють панель інструментів і рядок стану:

- `N1.Checked:=true; N1.Caption:='Панель інструментів'`
  `N1.Name:= miViewToolBar // N1 → miViewToolBar`
- `N2.Checked:=true; N2.Caption:='Рядок стану'`
  `N2.Name:= miViewStatusBar //N2 → miViewStatusBar`

Підопції `miViewToolBar` і `miViewStatusBar` для опції Вигляд впливають тільки на панель інструментів і рядок стану, а тому для них запишемо процедури опрацювання події `OnClick`:

```
procedure TForm1.miViewToolBarClick(Sender: TObject);
begin
 if miViewToolBar.Checked
  then begin miViewToolBar.Checked:=false;
             ToolBar1.Visible:=false   end
  else begin miViewToolBar.Checked:=true;
             ToolBar1.Visible:=true    end
end;
procedure
         TForm1.miViewStatusBarClick(Sender:TObject);
begin
 if miViewStatusBar.Checked
  then begin miViewStatusBar.Checked:=false;
            StatusBar1.Visible:=false
       end
  else begin miViewStatusBar.Checked:=true;
            StatusBar1.Visible:=true
       end
end;
```

12.6.6. Формування контекстозалежних меню

Створимо у проекті два контекстозалежні меню. Для цього перенесемо на форму два компоненти класу **TPopupMenu**. Формування контекстозалежного меню відбувається за допомогою *Конструктора Меню*, що викликається подвійним натисканням на компонентові TPopupMenu. А далі робимо все аналогічно створенню головного меню програми. Слід зауважити, що властивості ShortCut (швидкі клавіші), Enabled (контролювання доступу), Bitmap (зображення, що з'являється близько опції меню) і Checked (встановлення чи відміна опції меню) є ідентичними однойменним властивостям головного меню.

Перше контекстозалежне меню PopupMenu1 (під мнемонічною назвою pumView) буде з'являтися при натисканні правої кнопки миші на панелі інструментів, чи у рядку стану, і міститиме дві підопції, які приховують/відновлюють панель інструментів чи рядок стану:

- N1.Caption:= 'Панель інструментів'
  N1.Name:= pumViewToolBar // N1 → pumViewToolBar
  pumViewToolBar.OnClick ⇗ miViewToolBarClick
- N2.Caption:= 'Рядок стану'
  N2.Name:= pumViewStatusBar //N2 → pumViewStatusBar
  pumViewStatusBar.OnClick ⇗ miViewStatusBarClick

Після цього зв'яжемо компоненти StatusBar1 і ToolBar1 з компонентом pumView. Для цього:

StatusBar1.PopupMenu ⇗ pumView
ToolBar1.PopupMenu ⇗ pumView

Друге контекстозалежне меню PopupMenu2 (під мнемонічною назвою pumEdit) буде з'являтися при натисканні правої кнопки миші на компонентові Memo1 і міститиме команди роботи з буфером обміну і команду виділення всього тексту:

- N1.Action:=acCut; N1.Caption:= 'Вирізати'
- N2.Action:=acCopy; N2.Caption:= 'Копіювати'
- N3.Action:=acPaste; N3.Caption:= 'Вставити'
- N4.Caption:= '-' // розділювач меню

- `N5.Action:=acSelAll; N5.Caption:= 'Виділити все'`

Замінимо назви підопцій меню на більш осмислені:

```
N1.Name:= pumEditCut; N2.Name:= pumEditCopy;
N3.Name:= pumEditPaste; N4.Name:= pumEditSepar;
N5.Name:= pumEditSelAll
```

Після цього зв'яжемо компонент `Memo1` з компонентом `pumEdit`. Для цього: `Memo1.PopupMenu` ⇗ `pumEdit`

12.6.7. Програмування процедур опрацювання подій

Запрограмуємо передусім процедуру опрацювання події *створення головної форми проекту*, передбачивши у ній очищення вмісту компонента `Memo1` і відключення акцій, які є недоступними на початку роботи застосування:

```
procedure TForm1.FormCreate(Sender: TObject);
begin Memo1.Clear;
      acPaste.Enabled:=false;
      acCut.Enabled:=false;
      acCopy.Enabled:=false;
      acSelAll.Enabled:=false;
      acSave.Enabled:=false;
      acSaveAs.Enabled:=false
end;
```

Запрограмуємо процедури опрацювання подій *натискання клавіш та клацання мишею*, передбачивши у них активізацію/деактивізацію акцій роботи з буфером обміну і виділення всього тексту:

```
procedure TForm1.Memo1MouseUp(Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
 if Memo1.Lines.Count>0 then acSelAll.Enabled:=true
  else begin acSelAll.Enabled:=false; exit end;
 if Memo1.SelLength>0 then
   begin acCut.Enabled:=true; accopy.Enabled:=true;
        acPaste.Enabled:=true  end
  else begin acCut.Enabled:=false;acCopy.Enabled:=false end
end;
```

```
procedure TForm1.Memo1KeyUp(Sender: TObject; var Key:
Word;  Shift: TShiftState);
begin
 if Memo1.Lines.Count>0
  then acSelAll.Enabled:=true
  else begin acSelAll.Enabled:=false; exit end;
 if Memo1.SelLength>0 then
   begin acCut.Enabled:=true;
         accopy.Enabled:=true; acpaste.Enabled:=true
   end
  else
   begin
       acCut.Enabled:=false; accopy.Enabled:=false;
   end
end;
```

Запрограмуємо процедуру опрацювання події *виконання акції* `acNew`, передбачивши у ній деактивізацію акцій роботи з буфером обміну і виділення всього тексту для нового файлу і активізацію акцій збереження файлу:

```
procedure TForm1.acNewExecute(Sender: TObject);
begin
 if (Memo1.Lines.Count>0) and
    (MessageDlg('Зберегти вмістиме?',
     mtConfirmation, [mbYes, mbNo],0)=MRYES)
  then acSaveExecute(Sender);
  Memo1.Clear;
  StatusBar1.Panels.Items[0].Text:= 'Без назви';
  acSave.Enabled:=true;   acSaveAs.Enabled:=true;
  acPaste.Enabled:=false; acCut.Enabled:=false;
  acCopy.Enabled:=false;  acSelAll.Enabled:=false
end;
```

Аналогічні речі треба передбачити й у процедурі опрацювання події *виконання акції* `acOpen`:

```
procedure TForm1.acOpenExecute(Sender: TObject);
begin
 if (Memo1.Lines.Count>0) and
```

```
    (MessageDlg('Зберегти вмістиме?',
     mtConfirmation, [mbYes, mbNo],0)=MRYES)
     then acSaveAsExecute(Sender);
 Memo1.Clear;
 if OpenDialog1.Execute then
  Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
 StatusBar1.Panels.Items[0].Text:=OpenDialog1.FileName;
 acSave.Enabled:=true;    acSaveAs.Enabled:=true;
 acPaste.Enabled:=false;  acCut.Enabled:=false;
 acCopy.Enabled:=false;
 if Memo1.Lines.Count>0 then acSelAll.Enabled:=true
end;
```

Запрограмуємо процедури опрацювання подій *виконання акцій* acExit, acSave і acSaveAs:

```
procedure TForm1.acExitExecute(Sender: TObject);
begin
 if (Memo1.Lines.Count>0) and
    (MessageDlg('Зберегти вмістиме?',
     mtConfirmation, [mbYes, mbNo],0)=MRYES)
  then acSaveExecute(Sender)
 Close
end;
procedure TForm1.acSaveExecute(Sender: TObject);
begin
 if StatusBar1.Panels.Items[0].Text='Без назви'
  then acSaveAsExecute(Sender)
  else
    Memo1.Lines.SaveToFile(StatusBar1.Panels.Items[0].Text)
end;
procedure TForm1.acSaveAsExecute(Sender: TObject);
begin
if SaveDialog1.Execute
 then Memo1.Lines.SaveToFile(SaveDialog1.FileName);
 StatusBar1.Panels.Items[0].Text:=SaveDialog1.FileName
end;
```

Запрограмуємо процедури опрацювання подій *виконання акцій* роботи з буфером обміну і виділення всього тексту:

```
procedure TForm1.acCutExecute(Sender: TObject);
begin
 with Memo1 do
  if SelLength>0 then CutToClipboard;
 acCut.Enabled:=false;
 acCopy.Enabled:=false;
end;
procedure TForm1.acCopyExecute(Sender: TObject);
begin
 with Memo1 do
  if SelLength>0
   then begin CopyToClipboard; SelLength:=0 end;
 acCut.Enabled:=false;
 acCopy.Enabled
:=false
end;
procedure TForm1.acPasteExecute(Sender: TObject);
begin
     Memo1.PasteFromClipboard
end;
procedure TForm1.acSelAllExecute(Sender: TObject);
begin
     Memo1.SelectAll;
     acPaste.Enabled:=true;
     acCut.Enabled:=true;
     acCopy.Enabled:=true
end;
```

12.6.8. Однодокументний інтерфейс (SDI)

Створений текстовий редактор має стиль *інтерфейсу з одним документом* (single document interface, SDI), у якому можна відкрити тільки один документ (текстовий файл) – щоб відкрити інший документ, попередній треба закрити.

Потрібно зауважити, що під однодокументним інтерфейсом не слід розуміти наявність у програмі тільки одного вікна. Наприклад, програма “Провідник” у Windows також є SDI-програмою. Однак вона може створювати похідні вікна для пошуку файлів чи папок, задання параметрів перегляду властивостей файлів тощо.

Будь-який проект Delphi має, як мінімум, одну зв’язану з ним форму, яка називається **головною**, – ця форма з’являється на екрані у початковий момент виконання програми. Проте програма може мати багато форм, кожна з яких розв’язує якусь локальну задачу і з’являється на екрані за потребою. Стиль форми визначає її властивість `FormStyle`:

```
TFormStyle = (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop);
property FormStyle: TFormStyle;
```

- `fsNormal` – визначає звичайну форму, що використовується для різних цілей, у т. ч. – для загального керування проектом (головна форма);
- `fsMDIChild` і `fsMDIForm` – використовують при створенні багатодокументних програм у стилі MDI (MDI – multi document interface). Цей стиль передбачає створення головного вікна MDI (його зазвичай називають **рамковим**), всередині якого за потребою з’являються **дочірні** вікна. Дочірні вікна не можуть виходити за границю свого власника – рамкового вікна. Для створення рамкового вікна використовується стиль `fsMDIForm`, а для створення дочірнього вікна – стиль `fsMDIChild`;
- `fsStayOnTop` – використовують для вікон, які завжди мають розміщуватися над усіма іншими вікнами програми. Цей стиль перешкоджає перекриттю вікна іншими вікнами, навіть, якщо воно стає неактивним і губить фокус уведення. Зрозуміло, що цей стиль використовується у ситуаціях, коли вікно містить інформацію, яка потребує підвищеної уваги користувача.

Сучасні багатовіконні програми найчастіше будують у стилі SDI, який на противагу MDI, не накладає обмежень на положення і розміщення допоміжних форм, кожна з яких у разі необхідності може мати своє головне меню (у стилі SDI реалізоване, наприклад, середовище Delphi). Для створення форми у цьому випадку вико-

ристовують стиль `fsNormal`. Зауважимо, що у рамках SDI–застосувань можуть бути використані рамкові MDI–форми зі своїми дочірніми вікнами, тому термін SDI має дуже умовний характер і застосовується переважно для протиставлення давно використовуваному терміну MDI.

12.6.9. Підключення допоміжних форм. Модальність форми

Для створення нової форми проекту достатньо виконати команди меню: ➢**File** ➢**New** ➢**Form** або вибрати потрібну форму з репозиторія: ➢**File** ➢**New** ➢**Other…** 🗀**Forms** ⇗ **About box** ❑ **OK** (для нашого проекту вибираємо стандартну допоміжну форму, яка відображає дані про програму: розгорнута назва проекту, версія, автор, дата створення, коментар тощо). Найперша підключена до проекту форма (стандартна назва –`Form1`) стає головною формою проекту, вікно якої автоматично з'являється на екрані у момент старту програми.

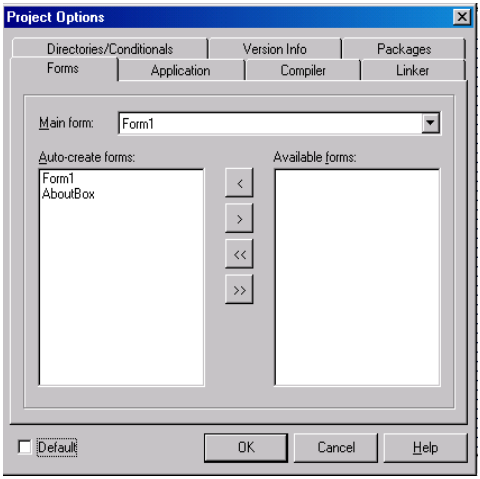


Рис. 12.9. Закладка опцій форм

Програміст може вказати будь-яку форму, вікно якої стане головним. Змінити умови появи форм можна на закладці опцій форм, яку викликають командами ➢**Project** ➢**Options** 🗀**Forms** (див. рис.12.9).

Головну форму вибирають у списку *Main form*. У нашому проекті нічого змінювати не треба: `Form1` – головна форма, а `Aboutbox` – домоміжна.

Користуючись вікнами, що розміщені на закладці *Forms* нижче, можна встановити, які форми мають бути створені автоматично, а які за потребою (у нашому проекті – це форма `Aboutbox`). Виділимо форму `Aboutbox` у лівому вікні *(Auto-create forms)* і за допомогою кнопки зі стрілкою, направленою вправо, пе-

ремістимо її у праве вікно доступних форм *(Available forms)*. Зауважимо, що для допоміжної форми створюється новий модуль **Unit2**. Для кожної автоматично створюваної форми Delphi додає у файл головного модуля проекту оператор створення:

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

Для форм, які були виключені зі списку автоматичного створення, аналогічний метод `Create Form` треба виконати у момент, коли форма має бути створена.

У момент створення форми виникає подія `OnCreate`. Опрацювання цієї події часто використовувалося у наших проектах для початкового настроювання компонентів форми, створення списків тощо.

За потребою методами `Show` або `ShowModal` форму можна зробити видимою. Останній метод відкриває форму, як **модальну**. Це означає, що керування передається цій формі і користувач не може передати фокус іншій формі цієї програми до тих пір, доки він не закриє модальну форму.

Методи `Show` і `ShowModal` можна застосовувати тільки до невидимої у цей момент форми. Якщо немає впевненості, що форма є невидимою, то перед тим, як використовувати ці методи, треба перевірити властивість `Visible` цієї форми.

У разі використання методів `Show` або `ShowModal` виникає подія форми `OnShow`. При опрацюванні цієї події можна використовувати якусь оперативну інформацію, яку отримують у процесі виконання програми, адже подія `OnShow` виникає кожного разу, коли форма стає видимою – на відміну від події `OnCreate`, яка настає для кожної форми лише один раз у момент її створення. Методом `Hide` форму у будь-який момент можна зробити невидимою. У цей час у ній виникає подія `OnHide`. Закрити форму можна методом `Close`. Водночас у формі, що закривається, виникає певна послідовність подій (`OnCloseQuery`, `OnClose`, `OnDestroy`), які можна опрацьовувати. Їхнє призначення – перевірити можливість закриття форми і вказати дії, які перед цим треба виконати.

Необхідно пам'ятати, що для виконання методів `Create-Form`, `Show`, `ShowModal`, `Hide` і для обміну інформацією між формами, модулі відповідних форм мають "знати" один про одного. Наприклад, якщо у модулі **`Unit1`** використовується форма модуля **`Unit2`**, то в операторі **`uses`** модуля **`Unit1`** треба ввести назву іншого модуля (**`Unit2`**). Якщо форма модуля **`Unit2`** має використовувати якусь інформацію з модуля **`Unit1`**, то відповідно у операторі **`uses`** модуля **`Unit2`** треба ввести назву першого модуля **`Unit1`**. Тоді, якщо оператори **`uses`** в обох модулях розміщені у розділах **`Interface`**, виникнуть проблеми з недопустимими круговими посиланнями *(circular unit reference)* і компілятор видасть відповідну помилку.

Від недопустимих кругових посилань можна звільнитись, якщо їх розімкнути, помістивши один чи обидва оператори **`uses`** у розділ **`implementation`**. Взагалі, простіше не включати назви модулів програми у оператори **`uses`** вручну, а використовувати для цього команду **➢File ➢Use Unit**, яка автоматизує процес і гарантуватиме відсутність кругових посилань. Незалежно від способу підключення у модулі **`Unit1`** нашого проекту матимемо опис:

```
...
var Form1: TForm1;
implementation
 uses unit2;
 {$R *.dfm}
procedure TForm1.acNewExecute(Sender: TObject);
...
```

Виклик допоміжної форми відбувається за допомогою опції головного меню "*Про програму*". Процедура опрацювання події виконання цієї опції меню має вигляд:

```
procedure TForm1.miAboutClick(Sender: TObject);
begin
     if not Assigned(AboutBox) then
       AboutBox:=TAboutBox.Create(Self);
     AboutBox.ShowModal
end;
```

Отже, ми створили текстовий редактор у стилі інтерфейсу SDI. Далі розглянемо принципи побудови проектів у стилі інтерфейсу MDI.

12.7. Багатовіконний інтерфейс

12.7.1. Принципи побудови проектів у стилі інтерфейсу MDI

Багатодокументний інтерфейс використовується переважно при створенні програм, які орієнтуються на роботу з однотипними документами. Прикладами таких програм є MS Word і MS Excel.

Основою програми MDI є **батьківське** вікно, у робочій області якого можуть розміщуватись дочірні вікна. Очевидно, що користувачеві буде незручно працювати з невпорядкованими вікнами: одні вікна закриватимуть інші, кількість відкритих документів буде важко визначити, важко знайти потрібне вікно.

Для вирішення цієї проблеми у стандартний набір методів форми стилю `fsMDIForm` (рамкове або батьківське вікно MDI) входить розміщення дочірніх вікон каскадом (метод `Cascade`), мозаїкою (метод `Tile`) чи впорядкування їхніх піктограм (метод `ArrangeIcons`). Нагадаємо, що дочірні форми (вікна) мають стиль `fsMDIChild`, а стиль форми визначає її властивість `FormStyle` (див. пункт 12.6.8).

Якщо при використанні методу `Tile` додатково задати значення властивості `TileMode` батьківської форми (`tbHorizontal` або `tbVertical`), то це приведе до горизонтального чи вертикального розміщення дочірніх вікон відповідно. У Delphi дочірні вікна за домовленістю розміщують каскадом.

Для активізації попереднього або наступного дочірнього вікна використовують методи батьківської форми `Previous` і `Next` відповідно. Для закриття активного дочірнього вікна застосовують метод `Close` об’єкта `ActiveMDIChild`, який є властивістю батьківського вікна MDI.

На етапі виконання програми дочірні вікна не мають власного меню. Керувати ними можна за допомогою меню головного вікна. Проте під час проектування **можна створити** головне меню дочірньої форми. Водночас на етапі виконання програми меню до-

чірньої форми **приєднується** до меню батьківської форми згідно з такими правилами:

- пункти утвореного меню розміщуються відповідно до зміни значень властивості `GroupIndex`;
- у разі збігу значень `GroupIndex` пункт меню дочірнього меню **замінить** відповідний пункт меню батьківського вікна.

12.7.2. Взаємодія батьківської та дочірніх форм MDI

Для батьківської форми у інспекторі об'єктів потрібно встановити `FormStyle=fsMDIForm`, а для дочірньої – `fsMDIChild`. Дочірні форми (вікна) документів створюються користувачем у процесі виконання програми і їхня кількість заздалегідь невідома, тому їх треба виключити з числа автоматично створюваних форм. У батьківській формі обмежуються лише головним меню та інструментальними панелями, залишаючи решту місця для дочірніх вікон. Для створення дочірнього вікна використовується метод `Create` (при опрацюванні подій, пов'язаних з пунктами меню `File`: `New` чи `Open`), наприклад:

```
procedure TMainForm.CreateMDIChild(const Name:string);
var Child: TMDIChild;
begin  // створення дочірнього вікна для файлу, назва
       // якого передається через параметр Name
 Child:=TMDIChild.Create(Application);
 Child.Caption:=Name;
 // заповненя Memo1 дочірнього вікна
 if FileExists(Name) then
                Child.Memo1.Lines.LoadFromFile(Name);
end;
procedure TMainForm.FileNew1Execute(Sender: TObject);
begin CreateMDIChild('Без назви ' +
            IntToStr(MDIChildCount + 1));
end;
```

Під час виконання програми можна використовувати такі властивості батьківського вікна для керування дочірніми вікнами: `MDIChildCount` – кількість відкритих дочірніх вікон; `ActiveMDIChild` – дискриптор активного дочірнього вікна; `MDIChild-`

`ren` – масив `TForm`, який містить дискриптори дочірніх вікон у порядку їхнього відкриття.

У Delphi є готовий шаблон текстового редактора у стилі інтерфейсу MDI, який можна активізувати за допомогою команд: **➢File ➢New ➢Other… 🗀Projects ⇗ MDI Application ❑ OK.**

12.8. Таймер

Компонент класу **`TTimer`** є невізуальним компонентом, який використовується для активування деякої операції через фіксовані проміжки часу. Компонент розміщений на закладці `System` палітри компонентів. Ієрархія:

`TObject - TPersistent - Tcomponent.`

У класі `TTimer` визначено такі властивості:

- **`property`** `Enabled:Boolean;` – визначає готовність реагувати на власну подію `OnTimer` (значення `True`);
- **`property`** `Interval:Cardinal;` – визначає проміжок часу у мілісекундах (1 секунда=1000 мілісекунд), після якого виникає подія `OnTimer`;

У класі `TTimer` описана подія

```
property OnTimer:TNotifyEvent;
```

яка виникає після встановленого у властивості `Interval` проміжку часу. Це є подія за домовленістю.

12.9. Створення і опрацювання графічних зображень

12.9.1. Компонент класу `TImage`

Компонент класу `TImage` використовується для розміщення на формі деякого малюнка з файлу зображення, який може бути бітовою картою (розширення `.bmp`), піктограмою (розширення `.ico`), метафайлом (розширення `.wmf`). Компонент розміщений на закладці `Additional` палітри компонентів. Ієрархія:

`TObject - TPersistent - TComponent - TControl - TGraphicControl.`

У класі **`TImage`** визначено такі властивості:

- **`property`** `Canvas:TCanvas;` – використовується для формування зображення на етапі виконання програми;

- **property** `Center:Boolean;` – визначає вирівнювання зображення по центру компонента (значення `True`), або зображення розміщується у лівому верхньому куті компонента (значення `False` – за домовленістю);
- **property** `Picture:TPicture;` – визначає зображення, що розміщується у компонентові;
- **property** `Strech:Boolean;` – визначає можливість збільшення/зменшення зображення до розмірів компонента (значення `True`). За домовленістю має значення `False`.

Як і інші графічні елементи керуванняя, компонент класу `TImage` опрацьовує всі події від миші. Події за домовленістю не має. На етапі проектування при розміщенні зображення у компонент класу `TImage` можна використовувати вікно вибору зображення, яке розкривається при активуванні компонента або при виборі властивості `Picture` у інспекторі об'єктів. Основне поле вікна вибору призначене для розміщення вибраного малюнка. Крім того, у вікні є кнопки: `Load` –завантаження зображення з файлу; `Save` – для запису зображення у файл; `OK` – підтвердження; `Cancel` – скасування.

12.9.2. Компонент класу `TShape`

Екземплярами класу **`TShape`** є компоненти-фігури – кола, еліпси, різні прямокутники і квадрати, які розміщують на канві. Компонент розміщений на закладці `Additional` палітри компонентів. Ієрархія:

`TObject - TPersistent - TComponent - TControl - TGraphicControl.`

У класі **`TShape`** визначено такі властивості:

- **property** `Brush:TBrush;` – визначає пензель для зафарбовування поверхні фігури;
- `property Pen:TPen;` – визначає олівець для малювання контура фігури;
- **type** `TShapeType=(stRectangle, stSquare, stRoundRect, stRoundSquare, stEllipse, stCircle);`

property `Shape:TShapeType;` – визначає фігуру, яку виводять на канву: `stRectangle` – прямокутник; `stSquare` – квадрат; `stRoundRect` – прямокутник із заокругленими кутами; `stRoundSquare` – квадрат із заокругленими краями; `stEllipse` – еліпс; `stCircle` – коло.

Подія за домовленістю – `OnDragDrop`.

Приклад. На формі періодично (через секунду) має змінюватися еліпс жовтого кольору на червоний прямокутник і навпаки.

```
unit Unit1;
interface
uses  Windows, Messages, SysUtils, Variants, Classes,
      Graphics, Controls, Forms, Dialogs, ExtCtrls;
type
  TForm1 = class(TForm)
    Shape1: TShape;
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
  private  { Private declarations }
  public   { Public declarations }
  end;
var Form1: TForm1;
implementation {$R *.dfm}
procedure TForm1.Timer1Timer(Sender: TObject);
 begin
   if shape1.Brush.Color = clYellow
    then  shape1.Brush.Color:=clRed
    else  shape1.Brush.Color:=clYellow;
   if shape1.Shape = stEllipse
     then  shape1.Shape:=stRectangle
     else  shape1.Shape:=stEllipse
 end;
end.
```

12.9.3. Компонент класу `TPaintBox`

Компонент класу `TPaintBox` надає канву для малювання у обмеженій області форми. Для малювання використовують усі можливості канви – шрифт, олівець, пензель, а також методи, що дають змогу будувати геометричні фігури на етапі виконання про-

грами. Компонент розміщений на закладці System палітри компонентів. Ієрархія:

TObject – TPersistent – TComponent – TControl – TGraphicControl.

У класі TPaintBox визначена подія

```
property OnPaint:TNotifyEvent;
```

яка виникає перед перемальовуванням компонента.

Подія за домовленістю – OnClick.

? Запитання для самоперевірки

1. **Що таке позначка? Для чого вона використовується?**
2. **Що таке панель? Для чого вона використовується?**
3. **Опишіть основні компоненти введення і відображення даних.**
4. Опишіть основні властивості класу TCustomEdit.
5. Опишіть основні методи класу TCustomEdit.
6. Яка різниця між компонентами класів TEdit і TMaskEdit?
7. Які компоненти є представниками багаторядкових текстових редакторів? Яка між ними різниця?
8. Як реалізовується мультивибір у компоненті класу TListBox?
9. Опишіть основні стилі компонента класу TComboBox.
10. Що таке компонент класу TStringGrid? Для чого він використовується?
11. Опишіть основні властивості класу TStringGrid.
12. **Опишіть основні компоненти введення і відображення цілих чисел.**
13. **Що таке командна кнопка? Для чого вона використовується?**
14. **Що таке командна кнопка з написом і піктограмою? Для чого вона використовується?**

15. Що таке швидка кнопка? Для чого вона використовується?

16. Що таке залежний перемикач? Для чого він використовується?

17. Що таке незалежний перемикач? Для чого він використовується?

18. Опишіть компонент класу `TOpenDialog`.
19. Опишіть компонент класу `TSaveDialog`.
20. Опишіть компонент класу `TOpenPictureDialog`.
21. Опишіть компонент класу `TSavePictureDialog`.
22. Опишіть основні стандартні діалоги.
23. Опишіть компонент класу `TMainMenu`.
24. Опишіть компонент класу `TPopupMenu`.
25. Що таке компонент класу `TImageList`? Для чого він використовується?
26. Що таке компонент класу `TActionList`? Для чого він використовується?
27. Що таке однодокументний інтерфейс?
28. Як у проекті підключати допоміжні форми?
29. Що таке інтерфейс MDI?
30. Як у проекті можна реалізувати зв'язок між батьківською та дочірніми формами?
31. Що таке таймер? Для чого він використовується?
32. Що таке компонент класу `TImage`? Для чого він використовується?
33. Що таке компонент класу `TShape`? Для чого він використовується?
34. Що таке компонент класу `TPaintBox`? Для чого він використовується?

Завдання для програмування

Завдання 12.1. Створити віконне застосування для табулювання функції, яка вибирається з *завдання 1.1* (стор. 23). Значення функції обчислювати у вузлах сітки, що утворюється розбиттям відрізка $[a, b]$ на n рівних частин, і зберігати на формі у вигляді списку класу `TListBox`, що допускає мультивибір. Після цього реалізувати обчислення агрегованих величин:

1. Найменшого значення функції.
2. Найбільшого значення функції.

3. Середнього арифметичного значення функції.
4. Добуток від'ємних значень функції.
5. Кількість невід'ємних значень функції.

Вимоги до виконання індивідуального завдання:

- обчислювати агреговані величини серед вибраних значень у списку або у всіх точках – за домовленістю;
- при введенні користувачем числових значень здійснювати попереднє цензурування символів;
- передбачити опрацювання можливих помилок при виконанні математичних операцій та обчисленні стандартних математичних функцій;
- вибирати агреговані величини незалежними перемикачами;
- на формі навпроти вибраних перемикачів мають бути відображені відповідні значення агрегованих величин. Поля, що розміщені навпроти невибраних перемикачів, мають бути невидимими.

Завдання 12.2. Завершити *приклад 12.7* (стор. 279), створивши власний варіант електронного мікрокалькулятора. Передбачити також можливість уведення чисел (з попереднім цензуруванням символів) з клавіатури.

Завдання 12.3. Створити віконне застосування для табулювання **функції з розривами** у вузлах сітки, яка утворюється розбиттям відрізка $[a, b]$ на n рівних частин. Вигляд функції вибирати з таблиці згідно з порядковим номером студента у списку студентів групи (у програмі задана функція має бути оформлена як *функція користувача*):

1. $y = 1/\cos^3(2x) - \sqrt{x^2 - 4}$	2. $y = \sin^4(3x) - 2x\sqrt{x^2 - 3}$
3. $y = (\cos^2(2x) - \sin x)\sqrt{x^2 - 2}$	4. $y = \sin(3x) - \operatorname{tg} x\sqrt{(1 + x)x}$
5. $y = \sqrt{x^2 - 5} - 2x \cdot \operatorname{tg} x$	6. $y = \sqrt{x^2 - 3x} + 2\sin x$
7. $y = (x^2 - 2x)^3 - 4x/\sqrt{x^2 - 4}$	8. $y = \sqrt[4]{x^2 + 3x} + \dfrac{1}{x}$
9. $y = 2 + 4x\sqrt{x^2 - 4}$	10. $y = x/\sqrt{x(x + 2)}$
11. $y = \sqrt{x^2 - 3} + 2\operatorname{tg} x$	12. $y = \dfrac{2x}{1 - x^2} + \sqrt{x(x + 3)}$

13. $y=\sqrt{x^3-3x}+3/\sin x$	14. $y=\dfrac{3x}{2-x^2}+\sqrt{x^2-4}$
15. $y=x\sqrt{x^2-4}+\dfrac{1}{x}$	16. $y=\dfrac{x}{1-x^2}+\sqrt{x^2-2}$
17. $y=\sqrt{x^2-3x}+3/\cos x$	18. $y=x+2\sqrt{x^2-4x}$
19. $y=3+4x/\sqrt{x^2-2x}$	20. $y=1/\sin x+\sqrt{x^2+x}$
21. $y=1/\sin(3x)+2\cdot\sqrt{(x+1)x}$	22. $y=\sqrt{x-3x^2}-2/\cos x$
23. $y=\sqrt{x^2-1}+2\,\mathrm{tg}\,x$	24. $y=\sqrt{x^2-3x}+3\cos^2 x$
25. $y=2-4x\sqrt{x^2-1}$	26. $y=\dfrac{x^2}{1-x}+\sqrt{x^2-4}$
27. $y=1+(x-1)\sqrt{x^2+2x}$	28. $y=\sqrt{x-3x^2}+2/\sin x$
29. $y=\sqrt{x^2-5x}-4/x^{\frac{2}{3}}$	30. $y=2\sin x-4x\sqrt{x^2-3}$

Вимоги до виконання індивідуального завдання:

- Інтерфейс проекту має відповідати інтерфейсу *прикл. 12.6* (стор. 268).
- Усі команди, коротку довідку про автора (`About`), коротку інформацію про роботу програми необхідно задати (чи продублювати) у меню та на інструментальній панелі.
- При введенні користувачем числових значень попередньо цензурувати символи. Контролювати сам факт введення даних.
- Реалізуємо контроль за межами табуляції (треба, щоб $a \le b$).
- Окремо розглядати випадок, коли $a = b$.
- Передбачити опрацювання можливих помилок при виконанні математичних операцій та обчисленні стандартних математичних функцій.
- Програма має перевіряти дотримання вимог про область допустимих значень арґументу функції і виводити повідомлення, якщо обидві межі не входять у цю область.

- Програма має корегувати початкові дані відповідно до області допустимих значень аргументу функції (якщо одна з меж не входить у область допустимих значень аргументу функції).
- У програмі передбачити можливість заокруглення результатів обчислень. Точність обчислень задається у діалозі з користувачем.
- Результати обчислень виводити на форму у вигляді таблиці (компонент `StringGrid`) та графічно (компонент `Chart` – вивчити самостійно). Для таблиці передбачити можливість скролінгу.

Завдання 12.4. Переробити проект побудови однодокументного текстового редактора (параграф 12.6), створивши текстовий редактор у стилі інтерфейсу MDI. Додатково ввести нові акції та відповідні компоненти для реалізації *завдань 9.1 – 9.3* (стор. 186 – 188).

Завдання 12.5. Увівши відповідні класи, створити віконне застосування для реалізації операцій над **комплексними** числами (студенти з непарними номерами у списку студентів групи) або над **раціональними** числами (студенти з парними номерами).

Список літератури

1. *Абрамов А.Г., Трифонов Н.П., Трифонова Г.Н.* Введение в язык Паскаль. – М.: Наука, 1988.
2. *Абрамов С.А., Гнездилова Г.Г., Капустина Е.Н., Селюн М.И.* Задачи по программированию. – М.: Наука, 1988.
3. *Архангельский А.Я.* Object Pascal в Delphi 5. – М.: Бином, 1999.
4. *Архангельский А.Я.* Программирование в C++ Builder 5. – М.: Бином, 2000.
5. *Архангельский А.Я.* Object Pascal в Delphi. – М.: Бином, 2002.
6. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979.
7. *Вирт Н.* Алгоритмы и структуры данных. – М.: Мир, 1989.
8. *Голуб Б.М., Вовк В.Д.* Бібліотечка Delphi. Програмування тривимірних зображень засобами OpenGL: Тексти лекцій. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2000.
9. *Голуб Б.М.* Бібліотечка Delphi. DLL: Методичні вказівки. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2000.
10. *Гудзь Р.В., Ярошко С.А.* Використання динамічних структур даних у програмах на Borland Pascal: Тексти лекцій. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2000.
11. *Дантеманн Джефф*, *Мишел Джим, Тейлор Дон*. Программирование в среде Delphi. – Киев: ДиаСофт, 1995.
12. *Дудзяний І.М., Черняхівський В.В.* Програмування мовою асемблера: Навчальний посібник. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2002.
13. *Епанешников А., Епанешников В.* Программирование в среде Turbo Pascal 7.0. – М.: Диалог-Мифи, 1993.
14. *Зеленяк О.П.* Практикум программирования на Turbo Pascal. Задачи, алгоритмы и решения. – Киев: ДиаСофт, 2001.
15. *Зуев Е.А.* Система программирования Turbo Pascal. - М.: Радио и связь, 1992.
16. *Зуев Е.А.* Программирование на языке Турбо-Паскаль 6.0,7.0. – М.: Радио и связь; Веста, 1993.
17. *Кандзюба С.П., Громов В.Н.* Delphi 6. Базы данных и приложения. Лекции и упражнения. – Киев: ДиаСофт, 2001.
18. *Кетков Ю.Л.*, *Кетков А.Ю.* Практика программирования: Visual Basic, C++ Builder, Delphi. – СПб.: БХВ – Санкт-Петербург, 2002.

19. *Кнут Д.* Искусство программирования на ЭВМ. – М.: МИР. Т.1. – 1976. Т.2. – 1977. Т.3. – 1978.
20. *Конопка Рей.* Создание оригинальных компонент в среде Delphi. – Киев: ДиаСофт, 1996.
21. *Костів О.В., Ярошко С.А.* Методи розробки алгоритмів: Тексти лекцій. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2002.
22. *Культин Н.Б.* Программирование в Turbo Pascal 7.0 и Delphi. – СПб.: BHV – Санкт-Петербург, 1997.
23. *Культин Н.Б.* Программирование на Object Pascal в Delphi 5. – СПб.: БХВ – Санкт-Петербург, 1999.
24. *Липский В.* Комбинаторика для программистов. – М.: Мир, 1988.
25. *Марченко А.И.* Программирование на языке Object Pascal 2.0. – Киев: ЮНИОР, 1998.
26. *Марченко А.И., Марченко Л.М.* Программирование в среде Turbo Pascal 7.0. – Киев: Век+, 1998.
27. *Немнюгин С.*А. Turbo Pascal: практикум. – СПб.: Питер, 2001.
28. *Новиков Ф.*, *Яценко А.* Microsoft Office 97 в целом. – СПб.: BHV – Санкт-Петербург, 1998.
29. *Петров В.Н.* Информационные системы. – СПб.: Питер, 2002.
30. *Пильщиков В.Н.* Сборник упражнений по языку Паскаль.– М.: Наука, 1989.
31. *Рейнгольд Э., Нивергельт Ю., Део Н.* Комбинаторные алгоритмы. – М.: Мир, 1980.
32. *Фаронов В.В.* Турбо Паскаль 7.0. Начальный курс. – М.: Нолидж, 1999.
33. *Фаронов В.В.* Турбо Паскаль 7.0. Практика программирования. – М.: Нолидж, 1997.
34. *Фаронов В.В.* Delphi 4. Учебный курс. – М.: Нолидж, 1999.
35. *Фаронов В.В., Шумаков П.В.* Delphi 5. Руководство разработчика баз данных. – М.: Нолидж, 2000.
36. *Федоров А.Г.* Delphi 3.0 для всех. – М.: КомпьютерПресс, 1998.
37. *Черняхівський В.В.* Delphi-4: сучасна технологія візуального програмування. – Львів: БаК, 1999.
38. *Шамис В*. Borland C++ Builder 5: учебный курс. – СПб.: Питер, 2002.
39. *Шумаков П.В.* Delphi 3 и разработка приложений баз данных. – М.: Нолидж, 1998.

Зміст

Вступ 3

1. Основні конструкції мови Object Pascal 5

1.1. Алфавіт мови. Ідентифікатори 5

1.2. Операції та вирази 8

1.3. Прості типи даних 12

1.4. Порожній оператор. Оператор присвоєння 17

1.5. Процедури введення/виведення даних у консольному режимі 18

1.6. Структура консольної програми 19

1.7. Перетворення типів даних 21

Запитання для самоперевірки 23

Завдання для програмування 24

2. **Галуження і цикли** 25

2.1. Складений оператор. Оператори галуження та вибору 25

2.2. Оператор циклу з параметром 28

2.3. Оператори циклу з умовою 29

2.4. Безумовна передача керування 31

2.5. Програми з простим повторенням 32

2.6. Програмування задач цілочислової арифметики 33

2.7. Сумування рядів 36

Запитання для самоперевірки 37

Завдання для програмування 38

3. Робота з рядками символів 41

3.1. Рядки символів у Паскалі 41

3.2. Типові задачі опрацювання рядків символів у Паскалі 43

3.3. Рядки символів у Object Pascal 46

3.4. Основні процедури/функції опрацювання рядків символів типу AnsiString 49

Запитання для самоперевірки 52

Завдання для програмування 53

4. Структуровані типи даних 55

4.1. Масив. Опис масиву 55

4.2. Одновимірні масиви (вектори) 57

4.3. Алгоритми пошуку в одновимірному масиві 67

4.4. Алгоритми сортування в одновимірному масиві 70

4.5. Матричні задачі 72

4.6. Записи (комбіновані типи) 76

4.7. Множини 78

4.8. Типізовані константи 82

4.9. Тип Variant у Object Pascal 85

Запитання для самоперевірки 86

Завдання для програмування 86

5. Процедури і функції 89

5.1. Загальні положення 89

5.2. Опис процедури. Оператор процедури 90

5.3. Опис функції. Виклик функції 91
5.4. Механізм взаємозв'язку між параметрами і аргументами 92
5.5. Відкриті параметри-масиви 97
5.6. Параметри без типів. Процедурні типи 98
5.7. Рекурсія 99
5.8. Параметри за домовленістю 100
5.9. Перевантаження підпрограм 101
Запитання для самоперевірки 101
Завдання для програмування 102

6. Покажчики. Динамічні структури даних 103
6.1. Статичне і динамічне розміщення даних у пам'яті 103
6.2. Покажчики 106
6.3. Операція рознайменування 108
6.4. Динамічні змінні 109
6.5. Нетипізовані покажчики 114
6.6. Динамічні масиви 117
6.7. Динамічні структури даних 120
6.8. Реалізація лінійних списків 124
Запитання для самоперевірки 133
Завдання для програмування 133

7. Система візуального програмування Delphi 6 135
7.1. Інтегроване середовище Delphi 6 135
7.2. Побудова проекту віконного застосування 138
7.3. Структура проекту віконного застосування 143
Запитання для самоперевірки 148
Завдання для програмування 148

8. Модулі 149
8.1. Означення модуля і його застосування у Паскалі 149
8.2. Модулі в Object Pascal 155
8.3. Приклад модуля Object Pascal (*виділення підрядків*) 156
8.3.1. Постановка завдання 156
8.3.2. Реалізація модуля виділення підрядків 158
8.3.3. Тестування модуля виділення підрядків 160
8.3.4. Опрацювання виняткових ситуацій 164
8.4. Модуль глобальних оголошень 165
Запитання для самоперевірки 170
Завдання для програмування 170

9. Файли 173
9.1. Загальні положення 173
9.2. Процедури і функції роботи з файлами 175
9.3. Текстові файли 176
9.4. Компонентні файли 177
9.5. Безтипові файли 178
9.6. Послідовний і прямий доступ 180

9.7. Приклади роботи з файлами (режим консолі) 181
9.8. Робота з файлами у Object Pascal 183
Запитання для самоперевірки 185
Завдання для програмування 186

10. Об'єктно-орієнтоване програмування 189

10.1. Основні концепції об'єктно-орієнтованого програмування 189
10.2. Оголошення класу 190
10.3. Створення і використання об'єктів 195
10.4. Наслідування та поліформізм 203
10.5. Властивості та події 208
10.5.1. Властивості 208
10.5.2. Події 210
10.5.3. Властивості-масиви 213
Запитання для самоперевірки 216
Завдання для програмування 216

11. Бібліотека візуальних компонентів (VCL) 217

11.1. Базові класи компонентів 217
11.2. Клас `TComponent` 220
11.3. Клас `TControl` 221
11.3.1. Батьківський і дочірній компоненти 221
11.3.2. Основні властивості класу `TControl` 223
11.3.3. Основні події, описані в класі `TControl` 226
11.3.4. Деякі методи класу `TControl` 228
11.3.5. Приклад опрацювання однієї події одночасно для декількох компонентів 229
11.4. Клас `TWinControl` 231
11.4.1. Основні властивості класу `TWinControl` 231
11.4.2. Деякі методи класу `TWinControl` 232
11.4.3. Основні події, описані в класі `TWinControl` 233
11.4.4. Приклади опрацювання подій натискання клавіш 234
11.5. Класи – прямі нащадки класу `TPersistent` 238
11.5.1. Клас `TCanvas` 239
11.5.2. Клас `TGraphicsObject` 241
11.5.3. Клас `TStrings` 243
Запитання для самоперевірки 244

12. Практичне використання компонентів 245

12.1. Компоненти введення і відображення текстових даних 246
12.1.1. Перелік основних компонентів 246
12.1.2. Позначки, панель 248
12.1.3. Властивості та методи класу `TCustomEdit` 251
12.1.4. Однорядкові текстові редактори 254
12.1.5. Багаторядкові текстові редактори 254
12.1.6. Компоненти відображення списків рядків 256

12.1.7. Компонент класу TListBox 258
12.1.8. Компонент класу TComboBox 262
12.1.9. Компонент класу TStringGrid 264
12.1.10. Уведення цілих чисел (TUpDown і TSpinEdit) 266
12.2. Перемикачі та кнопки 273
12.2.1. Компонент класу TButton (командна кнопка) 273
12.2.2. Командна кнопка з написом і піктограмою 273
12.2.3. Компонент класу TSpeedButton (швидка кнопка) 275
12.2.4. Компонент класу TRadioButton 276
12.2.5. Компонент класу TRadioGroup 276
12.2.6. Компонент класу TCheckBox 277
12.2.7. Компонент класу TCheckListBox 278
12.2.8. Приклади використання кнопок і перемикачів 279
12.3. Діалогові вікна 285
12.3.1. Загальні положення 285
12.3.2. Компоненти класів TOpenDialog і TSaveDialog 287
12.3.3. Компоненти TopenPictureDialog/TSavePictureDialog 290
12.4. Стандартні діалоги 291
12.5. Компоненти меню 295
12.5.1. Компонент класу TMainMenu 295
12.5.2. Компонент класу TPopupMenu 296
12.5.3. Компонент класу TImageList 296
12.6. Використання меню та діалогових вікон на прикладі побудови текстового редактора 297
12.6.1. Початкове розміщення компонентів на формі 297
12.6.2. Компонент класу TActionList 298
12.6.3. Заповнення панелі інструментів кнопками 299
12.6.4. Формування набору зображень 300
12.6.5. Формування головного меню форми 301
12.6.6. Формування контекстозалежних меню 304
12.6.7. Програмування процедур опрацювання подій 305
12.6.8. Однодокументний інтерфейс (SDI) 308
12.6.9. Підключення допоміжних форм. Модальність форми 310
12.7. Багатовіконний інтерфейс 313
12.7.1. Принципи побудови проектів у стилі інтерфейсу MDI 313
12.7.2. Взаємодія батьківської та дочірніх форм MDI 314
12.8. Таймер 315
12.9. Створення і опрацювання графічних зображень 315
12.9.1. Компонент класу TImage 315
12.9.2. Компонент класу TShape 316
12.9.3. Компонент класу TPaintBox 317
Запитання для самоперевірки 318
Завдання для програмування 319
Список літератури 322

Навчальне видання

Ігор Михайлович Дудзяний

Програмування мовою Object Pascal

Редактор Л. Макітринська
Технічний редактор С. Сеник
Комп'ютерне макетування І. Дудзяний
Коректор М.Михалюк

Підп. до друку 12.11.2003. Формат 60×84/16. Папір друк.
Друк на ризогр. Умовн. друк. арк. 19,1. Обл.-вид. арк. 19.8.
Тираж 200 прим. Зам.

Видавничий центр Львівського національного університету
імені Івана Франка
79602 м. Львів, вул. Дорошенка, 41