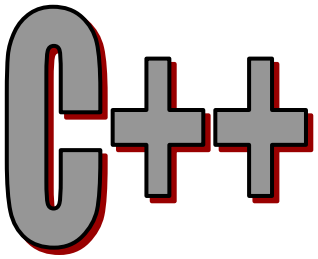

Трофименко О.Г.
Прокоп Ю.В.
Швайко І.Г.
Бута Л.М.
Шаповаленко В.А.
Леонов Ю.Г.
Яснинський В.В.
C++ теорія та практика
За ред. Трофименко О.Г.
C++
Теорія та практика
підручник для вузів
Ректорська серія

За ред. канд. техн. наук, доцента
О.Г. Трофименко



Теорія та практика

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів
напряму підготовки «Телекомунікації»
вищих навчальних закладів

Ректорська серія

ББК 32.973(076)
УДК 004.43

Гриф надано Міністерством освіти і науки України,
лист № 1/11-2645 від 04.04.2011 р.

Р е ц е н з е н т и:
В. М. Тонконогий, д.т.н., професор, зав. каф. інформаційних технологій проектування в машинобудуванні Одеського національного політехнічного університету;
М. П. Мусієнко, д.т.н., професор кафедри інформаційних технологій і програмних систем Чорноморського державного університету ім. Петра Могили;
С. Л. Ємельянов, к.т.н., доцент, зав. каф. правової інформатики Національного університету "Одеська юридична академія";
І. В. Стрелковська, д.т.н., професор, декан факультету інформаційних мереж Одеської національної академії зв'язку ім. О. С. Попова

А в т о р с ь к и й к о л е к т и в:
Трофименко О. Г., Прокоп Ю. В., Швайко І.Г., Буката Л. М.,
Шаповаленко В. А., Леонов Ю. Г., Ясинський В. В.

С++. Теорія та практика : Навч. посібник / [О. Г. Трофименко, Ю. В. Прокоп, І. Г. Швайко, Л. М. Буката та ін.] ; за ред. О. Г. Трофименко. – 587 с.

Навчальний посібник містить опис мови програмування С++ стандарту ANSI у поєднанні із засобами візуального програмування та об'єктно-орієнтованого підходу C++ Builder. Розглянуто засоби програмування як базових алгоритмів, так і опрацювання структурованих типів, робота з покажчиками, засоби динамічного керування пам'яттю тощо. Значну увагу приділено опрацюванню динамічних структур даних, програмуванню списків, дерев, автоматів. Окремий розділ присвячено засобам об'єктно-орієнтованого програмування. Кожна з розглянутих тем супроводжується значною кількістю прикладів програм.

Начальний посібник підготовлено до видання на кафедрі "Інформаційні технології" в ОНАЗ ім. О.С. Попова. Призначено для студентів технічних вузів і може бути корисною для кола читачів, які бажають навчатися програмуванню мовою С++.

ISBN 978-966-7598-59-4

ЗМІСТ

ПЕРЕДМОВА

Найбільш поширеною мовою програмування упродовж кількох останніх десятиліть, поза жодним сумнівом, є мова С++, на підставі якої “виросло” багато сучасних мов програмування і програмних середовищ. Цьому сприяли такі її властивості, як лаконічність, потужність, гнучкість, мобільність, можливість доступу до всіх функціональних засобів системи. Програмувати на С++ можна як для Windows, так і для Unix, причому для кожної з операційних систем існує значна кількість засобів розробляння: від компіляторів до потужних інтерактивних середовищ, як, приміром, Borland C++ Builder, Microsoft Visual C++ чи Visual Studio.NET.

Щоб не звужувати вибір читача щодо технічних засобів, а зосередити його мовою С++, її можливостях та особливостях, навчальний посібник подано як ґрунтовний і водночас доволі легкий для сприйняття матеріалу стосовно вивчення класичних методів програмування. Видання призначене як для початківців, які прагнуть навчатися програмувати мовою С++, так і для тих, хто має базові знання з програмування і прагне їх поглибити та освоїти більшість методів і прийомів, які має “на озброєнні” програміст. У посібнику містяться не лише тривіальні алгоритми, але й задачі з опрацювання структурованих типів даних, робота з вказівниками, засоби динамічного керування пам’яттю тощо. Значну увагу приділено опрацюванню динамічних структур даних, а саме, програмуванню списків, дерев, автоматів. Окремий розділ присвячений засобам об’єктно-орієнтованого програмування.

Навчальний посібник може використовуватись як джерело прикладів, вправ та програм для розглядання. Автори намагались показати специфіку основних засобів та можливостей потужної й гнучкої мови С++ на значній кількості прикладів працездатних програм і програмних фрагментів, реалізованих у середовищі Borland C++ Builder, які можуть бути використані у різних версіях С++. Автори сподіваються, що численна кількість прикладів програм викличе у читача бажання не лише реалізувати їх на комп’ютері та перевірити їхню працездатність, але й змінити та удосконалити ці програми.

Посібник складається з п’ятнадцяти розділів.

На початку, в розд. 1, можна ознайомитись зі структурною складовою комп’ютера та призначенням його основних пристроїв, вивчити специфіку програмного забезпечення, особливості зберігання інформації в комп’ютері й правила переведення чисел з однієї системи числення до іншої. Крім того, у першому розділі описані етапи розв’язування обчислювальних задач на комп’ютері та різновиди складання алгоритмів.

У розд. 2 описані інструментальні засоби інтерактивного середовища С++ Builder – однієї із найпотужніших систем, яка дозволяє на найсучаснішому рівні створювати як окремі прикладні програми Windows, так і розгалужені комплекси, призначені для роботи в корпоративних мережах і в Інтернеті. Середовище С++ Builder поєднує засоби мови програмування С++ та компонентно-орієнтований підхід до створення програм. Поєднання простоти освоєння візу-

ального середовища та підтримки широкого спектра технологій робить C++ Builder універсальним інструментом створення програмних проектів якого завгодно рівня складності. Навіть початкові програмісти зможуть швидко створювати програмні проекти в C++ Builder, які матимуть професійний віконний інтерфейс найрізноманітнішої спрямованості, від суто обчислювальних та логічних, до графічних та мультимедійних. Поряд з цим, значну увагу в цьому розділі приділено створюванню консольних програм, оскільки їхній програмний код є універсальним і майже не відрізнюється від програм інших засобів розробляння програм мовою C++.

Інформацію стосовно основних елементів мови C++, тобто щодо типів даних, правил оголошування змінних та інших об'єктів, записування арифметичних виразів, наведено в розд. 3. Цей розділ є надто важливим, оскільки для створювання програм треба знати синтаксис мови, тобто правила записування команд та використовування лексичних одиниць мови.

У розд. 4 приділено увагу організації трьох основних видів обчислювальних процесів (лінійних, розгалужених та циклічних) та застосуванню різного роду операторів для їхньої програмної реалізації. Специфіку їхнього програмування ілюструє значна кількість прикладів програм та програмних фрагментів.

Широкого застосовування для зберігання та опрацювання однорідної інформації, приміром таблиць, векторів, рядків, матриць та багато ін., мають масиви, організації та опрацюванню яких присвячено розд. 5.

У розд. 6 йдеться про переваги та особливості організації динамічної пам'яті для структур даних, які змінюють розміри, з метою ефективного використовування оперативної пам'яті. Така можливість пов'язана з наявністю та широким використовуванням у C++ особливого типу даних – вказівників.

Оскільки на практиці, крім чисел, досить часто доводиться працювати з символьною інформацією, цій темі присвячений окремий розділ. Так, у розд. 7 докладно, з наведенням значної кількості наочних прикладів, розглянуто особливості опрацювання різних видів організації символьної інформації: і як символьних масивів певної довжини, і за допомогою вказівників на початок символьного масиву, і як рядки.

Незамінну роль відіграють вказівники при передаванні параметрів до функції. У C++ підтримуються функції як окремі логічні одиниці (програмні конструкції) для виконування конкретних завдань. Наприклад, процес розробляння програмного забезпечення передбачає поділ складного завдання на набір більш простих завдань, кожне з яких розв'язується в окремій функції. Висвітленню специфіки організації функцій присвячено розд. 8.

Доволі часто функції організовують в окремі файли, особливості такого процесу описано у розд. 9.

У розд. 10 розглянуто спеціальні засоби C++ та C++ Builder для опрацьовування даних типу дата-час. Проілюстровано значну кількість функцій та методів опрацьовування такого роду даних.

Значну увагу в розд. 11 приділено складовому типу даних – структурі, яка, на відміну від масивів, може поєднувати дані різних типів. За допомогою структур зручно зберігати списки даних, що складаються з полів різних типів.

Окрім того, в цьому розділі вивчаються можливості створювання типів користувачів на базі стандартних типів, особливості організації й застосування об'єднань, перераховного типу та множин.

Значну увагу в навчальному посібнику приділено специфіці організації та опрацювання файлів у С++, оскільки на практиці часто виникає потреба, щоб вся опрацьовувана інформація зберігалася в окремих файлах. Так, у розд. 12 описано й продемонстровано на низці переконливих прикладів використовування потужних та гнучких можливостей С++ для різних підходів щодо організації файлів. Зазначено, що С++ припускає роботу з текстовими й бінарними файлами і має принципово різні способи їх опрацювання. У розділі висвітлюється специфіка роботи з файлами як з потоками у стилі С++. Окрім того, описано можливості роботи з файлами в С++ Builder за допомогою бібліотечних компонентів.

У розд. 13 обговорюються питання організації динамічних структур даних. Докладно висвітлюється матеріал про роботу зі списками, стеками, чергами, бінарними деревами та автоматами. Цей матеріал стане у пригоді тим, хто прагне освоїти щонайбільше прийомів та методів структурного програмування.

Оскільки С++ є мовою об'єктно-орієнтованого програмування (ООП), висвітленню цього питання присвячено окремий розд. 14. В основі підходу лежить ідея моделювання об'єктів за допомогою ієрархічно пов'язаних класів. Встановлення чіткого взаємозв'язку поміж даними й операціями зумовлює цілісність даних і значно підвищує надійність програм.

У розд. 15 приділено увагу налагодженню програм, наведено перелік поширених помилок і зауважень, як на етапі компіляції, так і на етапі виконування програми.

Щоб забезпечити більш ефективне сприйняття матеріалу та формування практичних навиків наприкінці кожного розділу розміщені питання та завдання для самоконтролю засвоєння знань.

У додатки винесено таблиці ASCII-кодів, перелік операцій мови С++ у порядку зменшення пріоритету та перелік функцій стандартної бібліотеки С++. До предметного покажчика, який призначений полегшити користування посібником, включено основні терміни і поняття, що зустрічаються у цьому виданні.

В основу навчальний посібника покладено курс лекцій, який читається в Одеській національній академії зв'язку ім. О.С. Попова на кафедрі інформаційних технологій.

Автори сподіваються, що запропонований до Вашої уваги посібник стане корисним при вивченні мови С++, і будуть щиро вдячні за доброзичливу та конструктивну критику й відгуки.

E-mail: kafedra.it@onat.edu.ua

Розділ 1

Апаратні та програмні засоби ЕОМ

1.1 Основні відомості про будову ЕОМ

Термін *"архітектура комп'ютерів"* – це система принципів, покладених в основу проектування ЕОМ (електронно-обчислювальної машини) певного типу. Більш вузьке значення цього терміну пов'язане з системою основних апаратних пристроїв (hardware) ЕОМ та засобів організації взаємодії між цими пристроями. У цьому контексті можна вважати, що *архітектура комп'ютерів* – це сукупність апаратних ресурсів (hardware) комп'ютера.

В основу архітектури переважної більшості комп'ютерів покладено принципи, які було сформульовані ще 1945 року американським математиком Джоном фон Нейманом. Усі комп'ютери, побудовані згідно з цими принципами, відомі тепер як комп'ютери з фоннеймановською архітектурою.

Основні принципи архітектури комп'ютерів фон Неймана є такі:

- ✓ використання двійкової системи числення для кодування інформації у комп'ютері;
- ✓ програмне керування роботою комп'ютера;
- ✓ зберігання програм у пам'яті комп'ютера;
- ✓ адресація пам'яті.

Ще на початку створення перших ЕОМ фірма IBM вперше зробила комп'ютер не як єдиний нероз'ємний пристрій, а забезпечила можливість його збирання з незалежно виготовлених частин аналогічно до дитячого конструктора. Способи з'єднання пристроїв у комп'ютері є доступні для удосконалювання його окремих частин, тобто для долучання нових чи заміни існуючих на більш досконалі пристрої. Такий підхід називається *принципом відкритої архітектури.*

Незважаючи на еволюцію обчислювальної техніки і незалежно від відмінностей стосовно способів фізичної реалізації, типовий персональний комп'ютер (ПК) містить у своєму складі такі основні пристрої, наведені ще у класичній роботі фон Неймана (рис. 1.1):

- ✓ *мікропроцесор* (МП чи CPU – central processor unit) – є центральним пристроєм, мозком, який безпосередньо здійснює координування усієї роботи, опрацювання, взаємозв'язок та обмін даними поміж функціональними вузлами через системну шину. МП включає в себе: 1) арифметико-логічний пристрій (АЛП) для виконання арифметичних і логічних обчислень; 2) пристрій керування для координації й синхронізації роботи усіх пристроїв комп'ютера; 3) мікропроцесорну пам'ять у вигляді регістрів та кеш-пам'яті; 4) інтерфейсну систему, яка реалізує зв'язок з іншими пристроями через системну шину. Зазвичай CPU і співпроцесор (FPU – floating point unit), який здійснює опрацювання дійсних чисел з рухомою крапкою, функціонально об'єднані в одну мікросхему (chip);

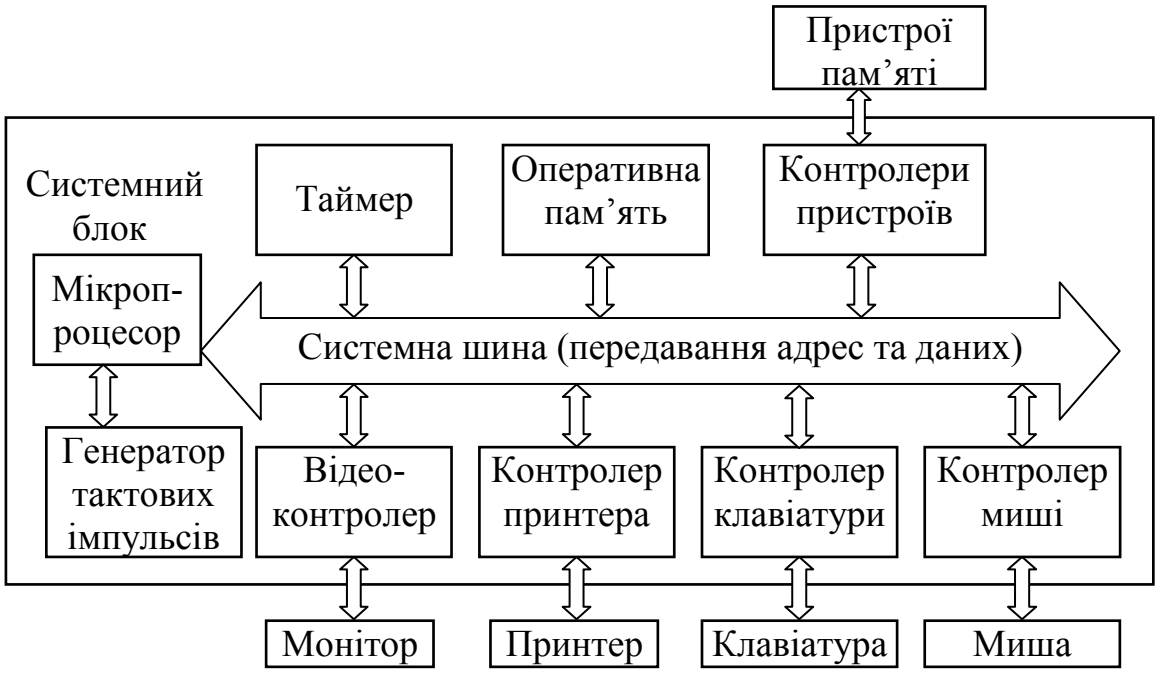


Рис. 1.1. Типова архітектура комп'ютера

✓ *пристрої швидкої пам'яті* – це оперативна пам'ять (ОП чи RAM – read access memory) та надшвидка кеш-пам'ять (cache), потрібні для зберігання опрацьовуваних даних. Вони є енергозалежними, тобто при вимиканні живлення інформація в цих пристроях втрачається;

✓ *пристрої пам'яті для довготермінового зберігання інформації* є енергонезалежними "сховищами" великої ємності. Доступ до інформації на таких носіях є набагато повільніший, аніж в ОП, але вартість одиниці пам'яті є набагато менша. Найпоширенішими сьогодні є жорсткі диски (вінчестери, HDD – hard disk driver), CD, DVD, USB флеш-накопичувачі (flash USB drive), картки памяті (memory cards) тощо;

✓ *пристрої введення інформації* передають інформацію від користувача до комп'ютера для її подальшого опрацювання. Більша частина інформації надходить сьогодні до комп'ютера через клавіатуру та мишу. Окрім того, інформація може вводитися до комп'ютера зі сканера, сенсорних екранів, з голосу людини через мікрофонні акустичні системи, з пристроїв зберігання інформації, з мережі, наприклад Internet;

✓ *пристрої виведення інформації* приймають опрацьовану інформацію і розміщують її на різноманітних фізичних пристроях виведення, щоб вона стала придатна для подальшого використання поза комп'ютером. Більшість вихідної інформації комп'ютера відображується на екрані, друкується на папері чи використовується для керування іншими пристроями. Також інформація може виводитися до мережі, приміром до Internet. Але найбільш зручним і поширеним пристроєм виведення даних є екран. Основними характеристиками, які забезпечують якість зображення, є фізичні розміри екрана і його елементів зображення (пікселів чи точок), частота розгортки, кольорові характеристики тощо. Кількість точок (пікселів) по горизонталі та вертикалі екрана називають його роздільною здатністю.

Для узгодження функціонування різних пристроїв ПК та системної шини використовують спеціальні мікросхеми – контролери (controller) та адаптери.

1.2 Програмне забезпечення

Програмне забезпечення (ПЗ) (software) – сукупність програм та службових даних, призначених для керування роботою комп'ютера. ПЗ комп'ютерів можна поділити на такі основні класи:

- ✓ операційна система та сервісні програми;
- ✓ інструментальні мови та системи програмування;
- ✓ прикладні системи.

Операційна система (ОС) (operating system) – це сукупність програмних засобів, яка здійснює розподіл ресурсів ПК та керування роботою усієї обчислювальної системи. Відомими операційними системами сьогодні є сімейство ОС Windows, Linux, Unix тощо. Операційна система виконує такі функції: керування пам'яттю, введенням-виведенням даних, файловою системою, взаємодією процесів; диспетчеризацію процесів; захист інформації; облік використовування ресурсів; опрацювання командної мови, – тобто забезпечує функціонування комп'ютера.

До складу сучасних операційних систем входять кілька підсистем, основні з них:

- ✓ підсистема управління процесами;
- ✓ файлова підсистема;
- ✓ драйвери – спеціальні програми, які забезпечують роботу з апаратурою;
- ✓ функції для організації взаємодії програм із користувачем;
- ✓ служба безпеки – розмежовування прав доступу.

Інструментальні мови та системи програмування використовують для переведення алгоритмів до комп'ютерних програм, тобто для розробляння програм системного та прикладного призначення.

Інструментальні *мови програмування* поділяють на дві основні категорії:

- ✓ мови низького (машинного) рівня – асемблери, близькі за структурою до інструкцій процесора. Вони орієнтовані на конкретні процесори, а тому набори їхніх інструкцій для різних груп комп'ютерів відрізнюються;
- ✓ мови високого рівня, призначені для того, щоб полегшити процес створювання програм, а тому їхні інструкції багато в чому нагадують мови спілкування людей. Здебільшого кожна з команд поєднує в собі одразу кілька машинних команд. Розрізнюють чотири різновиди мов високого рівня: 1) імперативні (процедурні), наприклад Pascal, C; 2) функціональні – Lisp; 3) логічні – Prolog; 4) об'єктно-орієнтовані – C++, C#, Java, Object Pascal тощо.

Програми, створені мовою високого рівня, переводяться на машинну мову спочатку інтерпретаторами, а потім компіляторами. Інтерпретатор автоматично у міру введення програми перетворює її команди (чи код) на команди машинної мови. А компілятор вже транслює (переводить) усю введену програму за командою програміста. Результатом компілювання є згенерований об'єктний код. Слово код ще доволі часто використовується для позначання тексту про-

грами чи її частини. Системи програмування надають зручний інтерфейс для створювання та налагоджування програм тією чи іншою мовою, приміром C++ Builder, Visual C++, Delphi, Visual Basic тощо.

Прикладні системи призначені для розв'язування задач певних класів. Приміром для виконування математичних інженерних обчислень використовують спеціальні математичні пакети MathCAD, Mathematika, Matlab; для введення й редагування текстової інформації застосовують текстові редактори Microsoft Word, Блокнот (Notepad) та інші; опрацьовувати табличні дані зручно в Microsoft Excel, а для опрацювання графічних даних існують пакети Adobe Photoshop, Paint тощо. Існує чимало інших прикладних програм для ПК, таких як видавничі системи, програми для анімації, програми-перекладачі, системи автоматизованого проектування та безліч інших.

Робота за сучасним комп'ютером потребує від користувача взаємодії як з апаратною частиною, так і з різноманітним програмним забезпеченням. Аналогічно робота сучасного комп'ютера потребує підключення та взаємодії з різними периферійними пристроями. Здійснюється це за допомогою *інтерфейса*, під яким розуміють сукупність апаратних та програмних засобів, які забезпечують взаємодію, співпрацю різних апаратних пристроїв поміж собою і з людиною.

Отже, кожна ЕОМ має дві основні складові – апаратне (hardware) і програмне (software) забезпечення. Збої у роботі однієї з програм можуть спричинити збої у функціонуванні комп'ютера й одержання помилкових результатів його роботи. Помилки апаратури призводять до неможливості реалізації команд програмного забезпечення.

1.3 Файлова система

Файлова система – це сукупність каталогів та файлів, які зберігаються на носіях зовнішньої пам'яті довготермінового зберігання інформації ПК. Файлова система є основним інформаційним об'єктом ОС.

Файл – це іменована область зовнішньої пам'яті для зберігання програм та даних. Кожний файл має такі характеристики: ім'я, розмір, дату останнього зберігання, певне місце розташування на диску та атрибути доступу. Імена файлів складаються з власного імені, крапки і розширення:

<ім'я>.*<розширення>*,

де *ім'я* – набір із символів алфавіту, цифр і спеціальних символів, а *розширення* визначає тип файла і містить, зазвичай, три символи. Наприклад, текстові файли мають розширення txt, документи Microsoft Word – `doc` чи `rtf`, таблиці Microsoft Excel – `xls`, бази даних Microsoft Access – `mdb`, графічні файли – `bmp`, `psd`, `jpg`, `gif` тощо. Інформація про всі атрибути файла міститься у каталогах.

Каталог (тека, директорія) – це логічна одиниця організації диска, яка має власне ім'я і може містити в собі файли та інші каталоги (підкаталоги). Головний каталог диска називають кореневим. Ім'я кореневого каталогу складається з імені диска та символу двокрапки. Інформація про всі атрибути файлів та підкаталогів використовується ОС для визначання повного місцеперебування

файла, яке записується у вигляді послідовності імен каталогів, розпочинаючи з кореневого, наприклад: `C:\Program Files\Microsoft Office\Clipart\A16.gif`. Каталоги, які входять до кореневого каталогу, називаються *підкаталогами 1-го рівня*. Каталоги, які входять до складу підкаталогу 1-го рівня, називаються *підкаталогами 2-го рівня* і т. д. Ієрархічну побудову диска можна подати у вигляді дерева підкаталогів (рис. 1.2).

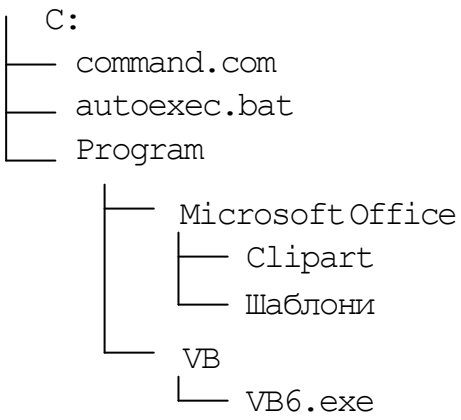


Рис. 1.2. Фрагмент дерева каталогів диска C

Для роботи з файлами існує кілька стандартних операцій, які підтримують усі операційні системи: створювання, копіювання, переміщування, перейменовування, вилучання.

Спеціальне призначення мають файли-ярлики. У такому файлі міститься посилання на інший файл (каталог, програму, документ тощо). Запуск ярлика відкриє той об'єкт, на який він посилається.

1.4 Системи числення

1.4.1 Різновиди систем числення

Під *системою числення* розуміють набір правил записування і позначання чисел за допомогою певного набору знаків (цифр). Залежно від способу використання цих знаків системи числення поділяються на непозиційні, змішані та позиційні.

У *непозиційних* системах числення значення цифр не залежать від їхнього розряду (позиції) у записі числа. Прикладом непозиційної системи є римська система числення, яка має такі числові значення: I – 1, V – 5, X – 10, L – 50, C – 100, D – 500, M – 1000. Отож, число 30 має вигляд: XXX. Тут цифра X в будь-якому місці означає число десять. Запис інших чисел: IV – 4, VI – 6, IX – 9, XI – 11, XL – 40, LX – 60, XC – 90, CX – 110, CM – 900, MC – 1100, MCMLXXXIX – 1989. У записі цих чисел значення кожної літери не залежить від позиції, на якому вона стоїть. Для записування великих чисел доводиться долучати все нові й нові знаки. Непозиційні системи є незручні для записування великих чисел і для виконування арифметичних дій.

У *змішаних* системах числення кількість допустимих цифр для різних розрядів (позицій) є різною. Вага кожного розряду визначається як добуток ваги попереднього розряду на вагу цього розряду. Найвідомішим прикладом змішаної системи числення є представлення часу у вигляді кількості діб, годин, хвилин і секунд. При цьому величина d днів h годин m хвилин s секунд відповідає значенню $d \cdot 24 \cdot 60 \cdot 60 + h \cdot 60 \cdot 60 + m \cdot 60 + s$ секунд.

У *позиційній* системі числення значення кожної цифри залежить від її розряду (позиції) у послідовності цифр, що зображують число. Десяткова система числення, якою ми користуємося у повсякденній практиці, є позиційною систе-

мою. Наприклад, у записі числа 444 цифра 4 повторюється три рази, але при цьому перша означає кількість сотень, друга – кількість десятків, третя – кількість одиниць. У комп'ютері застосовуються позиційні системи числення.

1.4.2 Одиниці інформації

Будь-яка (числова, текстова, графічна, аудіо, відео тощо) інформація в комп'ютері кодується у цифровому вигляді за допомогою двійкової системи числення. Використання двійкової системи числення набагато спрощує апаратну реалізацію пристроїв комп'ютера, оскільки в цій системі числення є лише дві цифри: 0 та 1. Зручність використання двійкової системи числення в обчислювальній техніці зумовлена тим, що електронні перемикачі можуть перебувати лише в одному із двох станів: увімкненому чи вимкненому. Ці стани можна кодувати двома цифрами: 1 чи 0. Так само у двох станах може перебувати канал передавання даних "рівень напруги є високий" чи "рівень напруги є низький". Крім того, чим більше рівнів (станів), які треба розрізнювати, тим менше є відмінностей поміж суміжними величинами і тим менш надійною є пам'ять. Двійкова система числення потребує розрізнювати лише дві величини, а отже, це є самий надійний метод кодування цифрової інформації.

Оскільки у комп'ютерах використовується запис інформації у двійковій системі числення, то кількість інформації вимірюють, підраховуючи кількість двійкових розрядів (комірок), потрібних для її запису. Одна двійкова цифра називається *бітом* (від англ. **bi**nary digi**t** – двійкова цифра). За допомогою одного біта можна закодувати два інформаційних повідомлення, які умовно позначаються символами '0' та '1'. За допомогою n бітів можна закодувати 2^n інформаційних повідомлень. Отже, *біт* є мінімальною одиницею обсягу пам'яті, проте на практиці ніхто не опрацьовує дані розміром в один біт.

Байтом називають послідовність з восьми бітів. *Байт* є мінімальною адресованою одиницею обсягу пам'яті. За допомогою одного байта можна закодувати $2^8 = 256$ різних комбінацій бітів, а отже, змінна розміром в один байт може зберігати числа в межах від 0 до 255. Наприклад, число 1101 0011 – це інформація обсягом в один байт. Два байти становлять слово, чотири байти – подвійне слово.

Ще однією одиницею вимірювання інформації, меншою за один байт, крім біта, є ніббл. *Ніббл* (англ. nibble, nybble), чи полубайт – одиниця вимірювання інформації, яка становить чотири біти й, відповідно, може мати 2^4 різних значень. Ніббл є синонімом "тетради" і є зручний тим, що його можна подати однією шістнадцятковою цифрою.

Приміром, щоб закодувати літери кирилиці, вистачить лише п'ять розрядів (бітів), оскільки $2^5 = 32$. Але, окрім кирилиці, в інформаційних документах також широко використовуються латинські літери, цифри, математичні знаки та інші спеціальні знаки. Тому для кодування усіх згаданих символів використовується восьмирозрядна послідовність 0 та 1, тобто один байт (див. розд. 7). Наприклад: цифра '9' кодується послідовністю бітів 0011 1001, літера латини 'W' – 0101 0111.

Для зручності позначання тисяч та мільйонів байтів використовуються такі одиниці:

- ✓ кілобайт (1 Кб (kB) = 2^{10}байт = 1024 байт);
- ✓ мегабайт (1 Мб (MB) = 2^{20} байт = 1024 Кб = 1 048 576 байт);
- ✓ гігабайт (1 Гб (GB) = 2^{30} байт = 1024 Мб = 1 073 741 824 байт);
- ✓ терабайт (1 Тб (TB) = 2^{40} байт = 1024 Гб = 1 099 511 627 776 байт);
- ✓ петабайт (1 Пб (PB) = 2^{50} байт = 1024 Тб);
- ✓ ексабайт (1 Еб (EB) = 2^{60} байт = 1024 Пб);
- ✓ зетабайт (1 Зб (ZB) = 2^{70} байт = 1024 Еб);
- ✓ йотабайт (1 Йб (YB) = 2^{80} байт = 1024 Зб).

Наприклад, якщо на сторінці тексту розміщується в середньому 2500 знаків, то 1 Мб – це приблизно 400 сторінок, а 1 Гб – 400 тисяч сторінок.

Зауважимо, що одиниці вимірювання інформації ґрунтуються на степенях числа 2. Десяткові префікси (кіло, мега і т. д.) дописуються лише умовно, оскільки 2^{10} = 1024 – число, близьке до 1000. Але різниця є, – і вона зростає зі зростанням ваги префікса. Більш правильно є використовувати двійкові префікси, але на практиці вони наразі не застосовуються, можливо, через неблагозвучність – кібібайт (2^{10}), мебібайт (2^{20}), гібібайт (2^{30}), тебібайт (2^{40}), пебібайт (2^{50}), эксбібайт (2^{60}), зебібайт (2^{70}), йобібайт (2^{80}).

Іноді десяткові префікси застосовують і у прямому значенні, наприклад при зазначенні ємності жорстких дисків чи модулів пам’яті, а також при зазначенні пропускної здатності каналів передавання даних (мереж) виробники використовують число 10 як основу для піднесення до степеня. Один гігабайт у цьому випадку дорівнює 10^9 б. А отже, реальна ємність, наприклад, 250-гігабайтного вінчестера (HDD) становить приблизно 232 Гб.

1.4.3 Основні позиційні системи числення

Кожне число N у позиційній системі числення з основою q можна в єдиний спосіб подати у вигляді полінома:

$$N_q = a_{n-1}\, q^{n-1} + a_{n-2}\, q^{n-2} + \ldots + a_1\, q^1 + a_0\, q^0 + a_{-1}\, q^{-1} + \ldots + a_k\, q^k = \sum_{i=-k}^{n-1} a_i q^i ,$$

де q – основа позиційної системи числення, яка визначає її назву;

a_i – цифри числа відповідного i-го розряду;

n – кількість цифр цілої частини числа;

k – кількість цифр дробової частини числа.

За q = 10 матимемо найпоширенішу десяткову систему числення. У ній кожне число записується поєднанням десяткових цифр, а внесок конкретної цифри залежить від її позиції – розряду. Розряди відлічуються справа наліво. Перший розряд називається розрядом одиниць, другий – десятків, третій – сотень і т. д. Число в десятковій системі числення можна подавати за допомогою операцій додавання, множення та піднесення до степеня, наприклад:

$$7248_{10} = 7 \cdot 10^3 + 2 \cdot 10^2 + 4 \cdot 10^1 + 8 \cdot 10^0 = 7000 + 200 + 40 + 8 = 7248.$$

Для двійкової системи за $q = 2$ вищенаведена формула набуде вигляду

$$N_2 = a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + ... + a_1 \cdot 2^1 + a_0 \cdot 2^0 + \ldots$$

Наприклад:

$$10101.101_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 21.625_{10}.$$

За $q = 8$ в аналогічний спосіб здобувають формулу для вісімкової системи, а за $q = 16$ – для шістнадцяткової системи числення тощо.

Існує чимало різноманітних позиційних систем числення, відмінних від десяткової, але у зв'язку із розвитком обчислювальної техніки, найбільшого поширення набули двійкова, вісімкова й шістнадцяткова системи числення (див. табл. 1.1). У подальшому, щоб явно зазначити використовувану систему числення, в індексі показуватимемо основу системи числення.

Таблиця 1.1

Основні системи числення

Системи числення	q	Базисні цифри системи числення
Двійкова	2	0, 1
Вісімкова	8	0, 1, 2, 3, 4, 5, 6, 7
Десяткова	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Шістнадцяткова	16	0, 1, …, 9, A(10), B(11), C(12), D(13), E(14), F(15)

Розглянемо кілька прикладів записування чисел у різних системах числення та їхнє десяткове подавання:

$194.38_{10} = 1 \cdot 10^2 + 9 \cdot 10^1 + 4 \cdot 10^0 + 3 \cdot 10^{-1} + 8 \cdot 10^{-2} = 194.38_{10}$;

$10011.1_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} = 19.5_{10}$;

$237.2_8 = 2 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 + 2 \cdot 8^{-1} = 159.25_{10}$;

$A1F_{16} = 10 \cdot 16^2 + 1 \cdot 16^1 + 15 \cdot 16^0 = 2591_{10}$.

Для зберігання і опрацювання даних в ЕОМ використовується двійкова система числення. Але на практиці для скорочення запису та зручності користувачів частіш використовується шістнадцяткова система.

1.4.4 Переведення чисел з однієї системи числення до іншої

При виконуванні задач на комп'ютері введення початкових даних і виведення результатів обчислень зазвичай виконується користувачами у звичній для них десятковій системі числення. Але з огляду на те, що переважна більшість комп'ютерів використовує двійкову систему числення, постає потреба у переведенні числа з однієї системи числення до іншої.

Переведення чисел з q-тої системи числення до десяткової безпосередньо виходить з поліноміального виразу конкретного числа, яке було розглянуте у п. 1.4.3.

Переведення цілого десяткового числа в q-ту систему числення відбувається у два етапи: спочатку переводиться ціла частина, потім дробова, після чого ліворуч від крапки записується ціла частина, а праворуч – дробова. Суть переведення полягає у послідовному діленні десяткового числа та його часток на

значення основи системи *q*. Ділення виконується, допоки наступна частка не буде меншою за основу *q*. Остача, обчислена на останньому кроці, є старшою (першою) цифрою переведеного числа. Результатом переведення такого числа до *q*-тої системи числення є запис останньої частки і всіх остач у зворотному порядку. Наприклад, переведення числа 133 з десяткової системи числення до вісімкової виконується у такий спосіб:

133 : 8 = 16 (**5**)
16 : 8 = **2** (**0**)

Результат: $133_{10} = 205_8$. Остачі від ділення записані в дужках після часток. Остання частка є старшою цифрою вісімкового числа, до якого дописуються решта остач у порядку, зворотному до порядку її обчислення.

Переведемо те саме число 133 до двійкової системи числення:

133 : 2 = 66 (**1**)
66 : 2 = 33 (**0**)
33 : 2 = 16 (**1**)
16 : 2 = 8 (**0**)
8 : 2 = 4 (**0**)
4 : 2 = 2 (**0**)
2 : 2 = **1** (**0**)

Результат: $133_{10} = 10000101_2$.

Переведення числа 133 до шістнадцяткової системи числення має вигляд:

133 : 16 = **8** (**5**)

Результат: $133_{10} = 85_{16}$.

Переведення десяткового дробу виконується послідовним множенням дробу і дробових частин результатів на основу системи *q*. Множення виконується аж до віднаходження нульової дробової частини або до обчислення числа із заданою точністю. Запис у прямому порядку всіх цілих частин добутку дає зображення даного дробу в *q*-тій системі числення.

Для прикладу переведемо десяткове число 0.125 почергово до двійкової, вісімкової та шістнадцяткової систем числення:

1) до двійкової:

$0.125 \cdot 2 = \mathbf{0}.25$;
$0.25 \cdot 2 = \mathbf{0}.5$;
$0.5 \cdot 2 = \mathbf{1}$

Результат: $0.125_{10} = 0.001_2$;

2) до вісімкової:

$0.125 \cdot 8 = \mathbf{1}$

Результат: $0.125_{10} = 0.1_8$;

3) до шістнадцяткової:

$0.125 \cdot 16 = \mathbf{2}$

Результат: $0.125_{10} = 0.2_{16}$.

А тепер наведемо приклад переведення десяткового числа з цілою та дробовою частинами 122.6 до двійкової системи числення з точністю у шість значущих цифр дробової частини.

Ціла частина	Дробова частина
122 : 2 = 61 (**0**) ↑	0.6 · 2 = **1**.2 ↓
61 : 2 = 30 (**1**)	0.2 · 2 = **0**.4
30 : 2 = 15 (**0**)	0.4 · 2 = **0**.8
15 : 2 = 7 (**1**)	0.8 · 2 = **1**.6
7 : 2 = 3 (**1**)	0.6 · 2 = **1**.2
3 : 2 = **1** (**1**) →	0.2 · 2 = **0**.4

Результат: $122.6_{10} = 1111010.100110_2$.

Отже, при переведенні цілої частини числа остачі, які залишаються у перебігу послідовного ділення часток, є цифрами цілої частини числа у новій системі числення. Остача, обчислена на останньому кроці, є старшою (першою) цифрою переведеного числа. А при переведенні дробової частини числа цілі частини чисел, які здобувають при множенні, не беруть участі у наступних множеннях. Вони є цифрами дробової частини результату. Значення першої цілої частини є першою цифрою після десяткової крапки переведеного числа тощо. Якщо при переведенні дробової частини виходить періодичний дріб, то здійснюють округлення, керуючись заданою точністю обчислень.

Для перевірки зробимо зворотне переведення здобутого двійкового числа 1111010.100110 до десяткової системи числення за допомогою поліному (див. п. 1.4.1). Невелику точність обчислень дробової частини у цьому прикладі зумовлено малою кількістю значущих цифр у попередньому перетворюванні.

$$1111010.100110_2 =$$
$$= 1\cdot2^6+1\cdot2^5+1\cdot2^4+1\cdot2^3+0\cdot2^2+1\cdot2^1+0\cdot2^0+1\cdot2^{-1}+0\cdot2^{-2}+0\cdot2^{-3}+1\cdot2^{-4}+1\cdot2^{-5}+0\cdot2^{-6}=$$
$$= 64+32+16+8+2+0.5+0.0625+0.03125 = 122.59375 \approx 122.6$$

Переведення десяткових чисел до вісімкової системи числення здійснюється аналогічно. За приклад переведемо вищенаведене число 122.6 і до вісімкової системи числення:

Ціла частина	Дробова частина
122 : 8 = 15 (**2**) ↑	0.6 · 8 = **4**.8 ↓
15 : 8 = **1** (**7**) →	0.8 · 8 = **6**.4
	0.4 · 8 = **3**.2
	0.2 · 8 = **1**.6
	0.6 · 8 = **4**.8

Результат: $122.6_{10} = 172.463146_8$.

Зворотне переведення з вісімкової системи числення до десяткової здобутого вище числа 172.463146_8 має вигляд:

$$172.463146_8 = 1\cdot8^2+7\cdot8^1+2\cdot8^0+4\cdot8^{-1}+6\cdot8^{-2}+3\cdot8^{-3}+1\cdot8^{-4}+4\cdot8^{-5} =$$
$$= 64+56+2+0.5+0.09375+0.0058598+0.00024414+0.00003052\approx$$
$$\approx 122.599884 \approx 122.6$$

Відповідне переведення десяткового числа 122 до шістнадцяткової системи числення здійснюється у такий спосіб:

Ціла частина	Дробова частина
122 : 16 = 7 (10=A)	0.6 · 16 = **9**.6
	0.6 · 16 = **9**.6
	0.6 · 16 = **9**.6
	0.6 · 16 = **9**.6

Результат: $122.6_{10} = 7A.9999_{16}$.

Зворотне переведення з шістнадцяткової системи числення до десяткової здобутого числа $7A.9999_{16}$ матиме вигляд:

$7A.999_{16} = 7\cdot16^{1}+10\cdot16^{0}+9\cdot16^{-1}+9\cdot16^{-2}+9\cdot16^{-3}+9\cdot16^{-4} =$
$= 112+10+0.5625+0.03515325+0.0021972656+0.000137329=$
$=122.599987845 \approx 122.6$

Переведення вісімкового числа до двійкової системи числення й навпаки здійснюються за допомогою табл. 1.2. Для цього треба виписати відповідні двійкові тріади усіх вісімкових цифр числа, розпочинаючи зі старшого розряду, наприклад:

$237_8 = 10\ 011\ 111_2$
$504_8 = 101\ 000\ 100_2$
$145.26_8 = 001100101.010110 = 1100101.01011_2$

Початкові й кінцеві незначущі нулі можна вилучити.

Для переведення двійкового числа до вісімкової системи числення треба виокремити тріади бітів спочатку цілої частини справа наліво, а потім – дробової частини зліва направо, доповнюючи їх, за потреби, незначущими нулями. Потім кожну тріаду слід замінити на відповідну вісімкову цифру, наприклад:

$11000100_2 = 11\ 000\ 100 = 304_8$
$1011.11101_2 = 001\ 011.111\ 010 = 13.72_8$

Алгоритм переведення із вісімкової системи числення до двійкової є зворотнім попередньому:

$304_8 = 11\ 000\ 100_2$
$401.04_8 = 100\ 000\ 001.000\ 1_2$

Переведення шістнадцяткових чисел до двійкової системи числення й навпаки здійснюється за допомогою тетрад (від грецької τετραζ – чотири), тобто кожен шістнадцятковий розряд подається чотирма двійковими. Правила переведення є аналогічні до правил для вісімкової системи числення, наприклад:

$1AF8_{16} = 1\ 1010\ 1111\ 1000_2$
$14.28_{16} = 1\ 1010.0010\ 1011_2$
$1100100001011_2 = 1\ 1001\ 0000\ 1011 = 190B_{16}$
$1011010.01011_2 = 101\ 1010.0101\ 1000 = 5A.58_{16}$

Отже, у різноманітних перетворюваннях головну роль відіграє двійкова система числення, а вісімкова та шістнадцяткова є допоміжними, тобто їх можна розглядати як скорочений запис двійкових чисел. Основами цих систем є ці-

лі степені числа 2: $2^3 = 8$, $2^4 = 16$. Найбільш застосовувана шістнадцяткова система числення широко використовується програмістами, оскільки подавання чисел у цій системі є більш компактним, аніж у двійковій чи вісімковій, і переведення з цієї системи до двійкової та навпаки виконується доволі просто.

Таблиця 1.2

Відповідність чисел різних систем числення

Десяткові числа	Двійкові числа	Вісімкові числа	Шістнадцяткові числа
0.0625	0.0001	0.04	0.1
0.125	0.001	0.1	0.2
0.25	0.01	0.2	0.4
0.5	0.1	0.4	0.8
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	1 0000	20	10

Наведемо приклади усіх можливих перетворювань до наведених систем числення для десяткового цілого числа 263:

1) з десяткової до двійкової:

263 : 2 = 131 (**1**)
131 : 2 = 65 (**1**)
65 : 2 = 32 (**1**)
32 : 2 = 16 (**0**)
16 : 2 = 8 (**0**)
8 : 2 = 4 (**0**)
4 : 2 = 2 (**0**)
2 : 2 = **1** (**0**)

Результат: $263_{10} = 100000111_2$;

2) з двійкової до десяткової:

$100000111_2 = 1 \cdot 2^8 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 256 + 4 + 2 + 1 = 263_{10}$

Результат: $100000111_2 = 263_{10}$;

3) з десяткової до вісімкової:

263 : 8 = 32 (**7**)
32 : 8 = **4** (**0**)

Результат: $263_{10} = 407_8$;

4) з вісімкової до десяткової:

$407_8 = 4 \cdot 8^2 + 7 \cdot 8^0 = 256 + 7 = 263_{10}$

Результат: $407_8 = 263_{10}$;

5) з десяткової до шістнадцяткової:

263 : 16 = 16 (**7**)
16 : 16 = **1** (**0**)

Результат: $263_{10} = 107_{16}$;

6) з шістнадцяткової до десяткової:

$107_{16} = 1 \cdot 16^2 + 7 \cdot 16^0 = 256 + 7 = 263_{10}$

Результат: $107_{16} = 263_{10}$;

7) з шістнадцяткової до двійкової:

Результат: $107_{16} = 1\ 0000\ 0111_2$;

8) з двійкової до шістнадцяткової:

Результат: $100000111_2 = \underline{0001}\ \underline{0000}\ \underline{0111} = 107_{16}$;

9) з вісімкової до двійкової:

Результат: $407_8 = \underline{100}\ \underline{000}\ \underline{111} = 100000111_2$;

10) з двійкової до вісімкової:

Результат: $100000111_2 = \underline{100}\ \underline{000}\ \underline{111} = 407_8$.

1.4.5 Арифметичні операції в різних системах числення

Основними арифметичними операціями над числовими даними в будь-якій системі є додавання, віднімання, множення й ділення. Правила виконування цих операцій в різних системах числення є аналогічними до загальновідомих і застосовуваних у десятковій системі. Відмінністю є лише те, що за потреби перенесення до наступного розряду при додаванні чи позичанні зі старшого розряду при відніманні виступає відповідна основа системи числення: для двійкової – 2, для вісімкової – 8, для шістнадцяткової – 16.

Приклади на *додавання* (крапками над числами позначають перенесення й позичання до старших розрядів):

$\overset{\cdot\cdot}{1100101_2}$	$\overset{\cdot\cdot}{250361_8}$	$\overset{\cdot}{F2A_{16}}$	$\overset{\cdot}{30A67_{16}}$
$+\ 1101110_2$	$+\ 2541714_8$	$+\ 3BC_{16}$	$+\ 2AF824_{16}$
11010011_2	3012275_8	$12E6_{16}$	$2E028B_{16}$

Приклади на *віднімання*:

100_2	100_8	100_{16}	$F05_{16}$
1_2	1_8	1_{16}	$D1_{16}$
11_2	77_8	FF_{16}	$E34_{16}$
1100101_2	5407024_8	$2AF8D4_{16}$	$CF05_{16}$
101010_2	640052_8	$F0A67_{16}$	$F06_{16}$
111011_2	4546752_8	$1BEE6D_{16}$	$BFFF_{16}$

Приклади на *множення*:

$\times 1001011_2$	$\times 47_8$	$+ 41_{16}$	$+ F2_{16}$
1001_2	23_8	AF_{16}	$3C_{16}$
$+ 1001011_2$	$+ 165_8$	$+ 3CF_{16}$	$+ B58_{16}$
1001011_2	114_8	$28A_{16}$	$2D6_{16}$
1010100011_2	1345_8	$2C6F_{16}$	$38B8_{16}$

1.5 Етапи розв'язування обчислювальних задач на комп'ютері

Хоча існують і використовуються сотні тисяч різноманітних прикладних програм, користувачі самі мають можливість розробляти власні програми для розв'язування тих чи інших задач на комп'ютері. Процес розробляння програми може бути подано як послідовність таких кроків:

1) *постановка задачі* – окреслення вимог щодо програми (приміром, визначення початкових даних, їхнього формату, параметрів введення). Результатом цього етапу розробляння є докладний словесний чи математичний опис алгоритму;

2) *побудова математичної моделі задачі* – опис задачі за допомогою математичних формул, визначення переліку початкових даних та шуканих результатів, вихідні умови, точність обчислень;

3) *вибір методу розв'язування*, оскільки одну й ту саму задачу може бути розв'язано за допомогою різних методів. Вибір методу має визначатися багатьма чинниками, основними з яких є точність результатів, час розв'язування, обсяг займаної оперативної пам'яті. У кожному конкретному випадку за критерій вибору методу беруть один зі згаданих критеріїв;

4) *розробляння схеми алгоритму розв'язування задачі*, тобто оптимальної логічної послідовності дій з урахуванням обраного методу розв'язування для здобуття певних результатів;

5) *написання та введення розробленої програми* алгоритмічною мовою програмування до комп'ютера;

6) *налагодження програми*, тобто пошук і виправлення можливих синтаксичних та алгоритмічних помилок у програмі;

7) *тестування* – перевірка правильності роботи програми.

Ця послідовність є притаманна для розв'язування якої завгодно задачі у програмний спосіб. Однак при підготовці задачі кожен етап може мати більш чи менш виражений характер.

1.6 Алгоритм, його властивості й засоби описування. Поняття програми

Алгоритм (algorithm) – це система формальних правил, які чітко й однозначно окреслюють послідовність дій обчислювального процесу від початкових даних до шуканого результату. Алгоритм визначає певні правила перетворювання інформації, тобто він зазначає певну послідовність операцій опрацювання даних, щоб здобути розв'язок задачі.

Походження слова алгоритм: понад тисячу років тому у Багдаді жив Абу Джафар Мухамад ібн Муса аль-Хорезмі, який у своїй праці «Трактат аль-Хорезмі про арифметичне мистецтво індусів» з арифметики та алгебри сформулював правила і окреслив послідовність дій при додаванні та множенні чисел. При переведенні латиною ім'я автора перетворили в Algorithmi, а з часом методи розв'язування задач стали називати алгоритмами.

Основні властивості алгоритмів:

детермінованість (визначеність) – однозначність результату обчислювального процесу за заданих початкових даних;

дискретність – поділ обчислювального процесу на окремі елементарні кроки, виконувані послідовно;

ефективність – простота та швидкість розв'язання задачі за мінімальних потрібних засобів;

результативність – забезпечення здобуття розв'язку через певну кінцеву кількість кроків;

масовість – забезпечення розв'язання якої завгодно задачі з класу однотипних.

Задавати алгоритми можна у різні способи:

- ✓ словесне описування послідовності дій;
- ✓ аналітичне описування у вигляді формул;
- ✓ графічне подавання у вигляді схеми алгоритму (блок-схеми);
- ✓ записування алгоритмічною мовою програмування.

Великого поширення набув графічний спосіб задавання алгоритмів у вигляді схем алгоритмів (блок-схем).

Схема алгоритму (блок-схема) – графічне зображення його структури, в якому кожен етап процесу опрацювання даних подається у вигляді різних геометричних фігур (блоків).

Форма блока зумовлює спосіб дій, а записи всередині – деталі (параметри) відповідного етапу (табл. 1.3). Біля блока можуть бути розміщені певні коментарі.

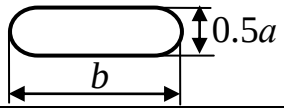
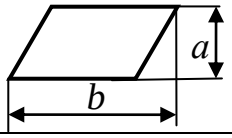
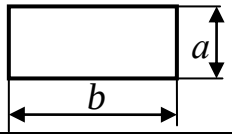
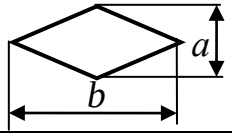
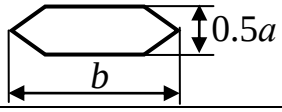
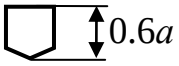
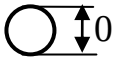
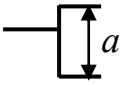
Блоки поєднуються поміж собою лініями потоку, які для кожного етапу визначають напрямок дій. Усім блокам присвоюють порядкові номери, які проставляються у розриві лінії контуру у лівій частині верхньої сторони зображен-

ня блока. Лінії потоку проводять паралельно до ліній зовнішньої рамки схеми. Напрямок ліній потоку зверху вниз і зліва направо прийнято за основний, і, якщо лінії не мають зламів, стрілками їх можна не позначати. В інших випадках їхній напрямок обов'язково позначають стрілкою. Лінію потоку зазвичай підводять до середини блока.

Відстань поміж паралельними лініями потоків має бути не меншою за 3 мм, поміж іншими блоками й символами – не менше за 5 мм. Лінію потоку можна обривати, використовуючи на місці обриву з'єднувачі, якщо схему виконано на двох чи більшій кількості аркушів, чи якщо з'єднувані блоки розташовано на значній відстані один від одного.

Таблиця 1.3

Конфігурація блоків

Назва блока	Графічний вид блока	Призначення у програмі
Початок-кінець		Початок-кінець чи вхід-вихід для підпрограм
Блок введення-виведення		Введення даних та виведення результатів обчислювання
Блок обчислювань		Обчислювання за формулами
Умовний блок (розгалуження)		Вибір напрямку виконування алгоритму залежно від виконання певних умов
Блок модифікацій (заголовок циклу)		Задає початкове й кінцеве значення та крок для параметра циклу
Посилання на іншу сторінку		Задає зв'язки поміж частинами схеми, розташованими на різних сторінках
З'єднувач		Зазначання зв'язку поміж перерваними лініями потоку, що зв'язують блоки
Коментар		Зв'язок поміж елементом схеми і поясненням

Розмір a має бути кратним до 5, наприклад: $a = 20$ мм. Розмір $b = 1.5a$.

Перевагою блок-схем є те, що за їхньої допомоги можна наочно зобразити структуру алгоритму цілковито, окресливши його логічну сутність. Особливо це є важливим для задач економічного характеру і задач управління. Такі задачі містять чималу кількість операцій порівнювання, логічних, арифметичних

та інших операцій, а отже, доволі складно встановити їхню послідовність у перебігу розв'язування задачі.

Розрізняють три базових алгоритмічних структури: лінійна (послідовність), розгалужена та циклічна (повторювання). Алгоритмічні структури будь-якої складності утворюються поєднанням послідовності цих базових алгоритмів. Приклади алгоритмічних структур та їхньої реалізації детально будуть розглянуті у наступних розділах.

Запис алгоритму алгоритмічною мовою програмування має бути зрозумілим не лише людині, а й комп'ютерові, а тому потребує точного дотримання правил цієї мови. Такий спосіб подавання алгоритму буде розглянутий далі при вивченні мови програмування С++.

Програма – упорядкована послідовність команд, які належить виконувати. Принцип програмного керування, який є основним принципом будови всіх сучасних ЕОМ, за Дж. фон Нейманом, полягає у тому, що всі обчислення, відповідно до алгоритму розв'язання задачі, мають бути подані у вигляді програми. Кожна команда програми містить вказівки стосовно конкретної виконуваної операції, місця розташування (адреси) операндів та низки службових ознак. *Операнди* – змінні, значення яких беруть участь в операціях опрацювання даних. Список усіх змінних (початкових даних, проміжних значень та результатів обчислень) є ще одним неодмінним елементом будь-якої програми.

Для доступу до програм, команд та операндів використовуються їхні адреси. За адреси слугують номери комірок пам'яті ПК, призначених для зберігання об'єктів.

Питання та завдання для самоконтролю

1) Назвіть основні принципи архітектури комп'ютерів фон Неймана.
2) В чому полягає принцип відкритої архітектури?
3) Вкажіть основні складові типового персонального комп'ютера.
4) Яке призначення мікропроцесора?
5) Назвіть енергозалежні та енергонезалежні пристрої пам'яті.
6) Назвіть призначення програмного забезпечення та його складові.
7) Які функції призвана виконувати операційна система?
8) Що називається файлом?
9) Чим файл відрізняється від теки (каталогу)?
10) Чи може тека зберігати інформацію?
11) Яке призначення мають файли-ярлики?
12) Чи можна створити файл у середині іншого файла?
13) З чого складається ім'я файла?
14) Які є обмеження на імена файлів? Наведіть приклади правильних і неприпустимих імен файлів.
15) Про що свідчить розширення файла? Які типи файлів Ви знаєте?
16) В який спосіб можна змінити ім'я файла?

17) Наведіть способи створення текстового файла. Як зберегти набраний текст у файлі?

18) Чим схожі й чим відрізняються текстові редактори Notepad (Блокнот) і MS Word? Чи можна відкрити файл з розширенням txt у редакторі MS Word? А у редакторі Paint?

19) Що таке біт і байт? Назвіть відомі Вам одиниці вимірювання інформації.

20) Що таке позиційна система числення і як віднайти значення числа за його записом у певній позиційній системі?

21) Чому інформація в комп'ютері записується у двійковій системі числення?

22) Перевести десяткові числа 50, 127, 255, 25.5 до двійкової та шістнадцяткової систем числення.

23) Перевести двійкові числа 1010, 11111, 1010101 до десяткової та шістнадцяткової систем числення.

24) Подати шістнадцяткові числа 3F, 80, F07 у десятковій та двійковій системах числення.

25) Обчислити у двійковій системі числення: 101101 + 1100, 11001 – 101.

26) Обчислити у шістнадцятковій системі числення: A06 + FB, AC6 – E9.

27) Назвіть призначення та способи подання алгоритмів.

28) Перелічіть блоки, використовуванні в схемах алгоритму.

29) Назвіть базові алгоритмічні структури.

30) Що таке програма? Опишіть процес створювання програм.

Розділ 2

Середовище програмування C++ Builder

2.1 Технологія створювання інтерфейсних програм

2.1.1 Засоби середовища C++ Builder

Кожна програма повинна мати зручний інтерфейс для спілкування з користувачем. Основним елементом інтерфейсу у Windows є вікна. Одним з різновидів вікон є форма, яка може містити кнопки, текстові поля, перемикачі тощо. Тому програми, написані для використання у Windows, зазвичай мають інтерфейс, подібний до вікон та форм. Для швидкого і зручного створювання програм з графічним інтерфейсом використовують спеціальні середовища візуального програмування. Майже кожна сучасна мова програмування має принаймні одне таке середовище: Object Pascal – Borland Delphi, Basic – Visual Basic, C++ – Borland C++ Builder, Microsoft Visual C. Візуальне програмування ще називають Rapid Application Development (RAD), "швидка розробка додатків". Технологія RAD суттєво прискорює створення програм з графічним інтерфейсом.

Інструментальна система Builder, подібно до інших систем візуального програмування (Visual C, Visual Basic, Delphi тощо), насамперед є посередником поміж інтерфейсом прикладного програмування Windows (API – Application Program Interface) та програмістом, надаючи змогу навіть програмістам-початківцям оперативно створювати програмні проекти, які матимуть графічний інтерфейс користувача (GUI – Graphic User Interface) найрізноманітнішої спрямованості, від суто обчислювальних і логічних до графічних і мультимедійних.

C++ Builder – це технологія візуального програмування, де автоматизовано її трудомістку частину – створювання інтерфейсних програм з діалоговими вікнами. Оболонка C++ Builder надає змогу замість повного самостійного написання програми використовувати великий набір готових візуальних об'єктів, так званих компонентів, піктограми яких розміщені на відповідних вкладках палітри компонентів. В C++ Builder існує понад 100 компонентів. Всі компоненти зібрано у бібліотеці візуальних компонентів VCL – Visual Class Library.

C++ Builder призначено для написання програм мовою програмування C++ і поєднує VCL та середовище програмування (IDE – Integrated Development Environment), написані на Delphi з компілятором C++. Цикл створення програмних проектів у C++ Builder є аналогічний до Delphi, але із суттєвими поліпшеннями. Більшість компонентів, розроблених у Delphi, можна використовувати і в C++ Builder без модифікації, але, на жаль, зворотне твердження не слушне.

C++ Builder дозволяє методом drag-and-drop доволі просто розробляти інтерфейсні програми, що зумовлює підвищення ефективності та простоту програмування, оскільки програмістові не треба кожного разу створювати ті еле-

менти власних програм, котрі може бути реалізовано за допомогою вже існуючих об'єктів.

Основним будівельним об'єктом візуального програмування є компонент. *Компонентами* в C++ Builder є об'єкти чи класи об'єктів, які є, у певному розумінні, об'єктами "реального світу". Їх безпосередньо видно на екрані (за винятком групи невидимих компонентів), їх можна пересувати мишею, вони можуть реагувати на клацання клавіш клавіатури і миші тощо. Своєю чергою, компонентам, на відміну від звичайних об'єктів C++, притаманна наявність *властивостей*, *подій* та *методів*, які дозволяють здійснювати різноманітні операції з цими компонентами.

Властивості (Properties) дозволяють легко встановлювати різні характеристики компонентів, такі як назва, розміри, контекстні підказки чи джерела даних. *Методи* (функції-члени) виконують певні операції над компонентним об'єктом, у тому числі й такі складні, як відтворювання чи перемотування пристрою мультимедіа. Події (Ivents) пов'язують зовнішні впливи, на які реагують компоненти, такі як активізація, натиснення кнопок чи редаговане введення – з кодами реакції на ці впливи. Окрім того, події можуть виникати за таких специфічних змінювань стану компонента, як поновлення даних в інтерфейсних елементах доступу до баз даних.

За приклад об'єкта розглянемо компонент `Button`, який за суттю є кнопкою і має низку властивостей: ім'я, розміри (довжина й висота) і розташування на формі, окреслення меж, надпис на кнопці, шрифт, стиль, розмір та колір надпису, видимість (кнопка може існувати, але бути невидимою) тощо. Для кнопки можна створювати різні події, наприклад: клацання кнопкою миші, натискання клавіші клавіатури тощо. Методами для кнопки можуть бути алгоритми, які виконують такі дії: переміщення самої кнопки, змінювання розмірів, створення нової чи знищення існуючої кнопки тощо.

Програмування в C++ Builder складається з двох етапів:

- ✓ конструювання візуального інтерфейсу за допомогою компонентів;
- ✓ написання програмного коду, виконання команд якого забезпечить розв'язок певної задачі.

Вікно середовища C++ Builder при завантаженні складається з таких елементів (рис. 2.1):

- ✓ вікно форми;
- ✓ вікно коду програми;
- ✓ головне меню;
- ✓ "гарячі" кнопки інструментальних панелей;
- ✓ палітра компонентів;
- ✓ вікно інспектора об'єктів.

Вікно форми займає найбільше місце і є прямокутним сірим "контейнером" (рис. 2.1), на який при проектуванні форми розміщують компоненти (кнопки, надписи, панелі, вікна редакторів тощо). Форма сама є компонентом з назвою `Form`. Без додаткових вказівок заголовок компонента (властивість `Caption`) збігається з його назвою (властивість `Name`), до якої додається порядковий номер, розпочинаючи з 1 (приміром `Button1`, `Button2`). Але заголовок

можна змінити за допомогою властивості `Caption`. Розміри форми також можна змінювати чи то за допомогою вікна Object Inspector, чи просто “зачепивши” мишкою за лінію межі форми (цього моменту курсор набуває вигляду двонапрямленої стрілки).

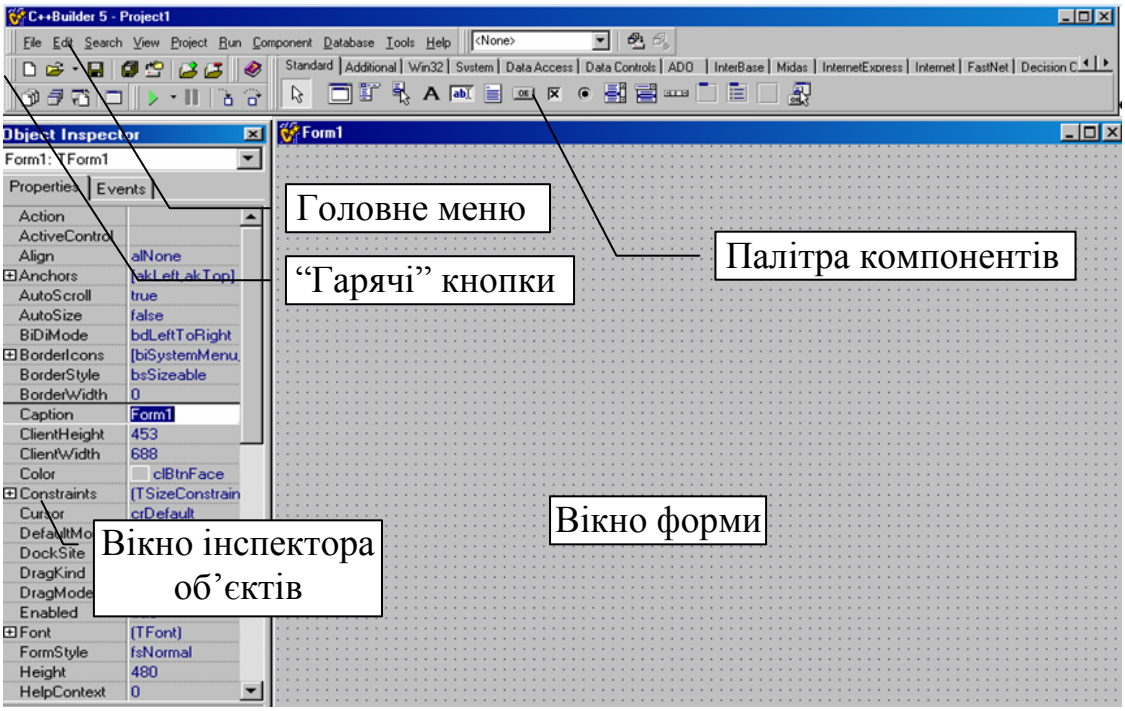


Рис. 2.1. Вигляд головного вікна середовища C++ Builder

Для розміщених на формі компонентів можна викликати контекстну довідку, для чого слід виокремити потрібний компонент і натиснути клавішу <F1>. Якщо клацнути на самій формі і натиснути клавішу <F1>, відкриється довідка по класу форми `TForm`.

Вікно редактора коду програми Unit1.cpp подано на рис. 2.2. Саме у цьому вікні прописують текст програми. Зазвичай вікно редактора коду перекривається вікном форми. Перемикання поміж вікнами форми й редактора можна здійснювати клавішею <F12> чи то натисненням миші на видимій частині обраного вікна.

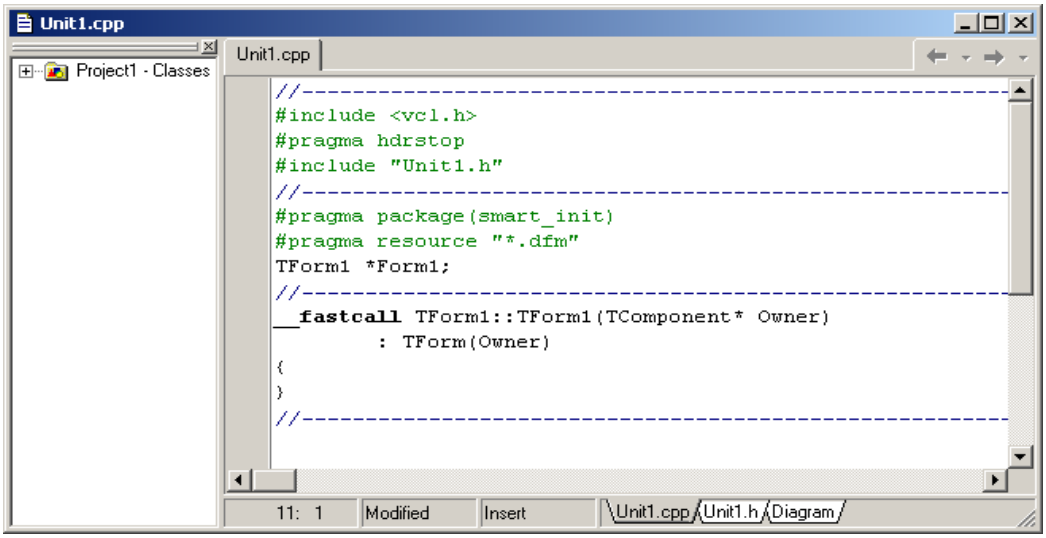


Рис. 2.2. Вигляд вікна редактора коду

З самого початку редактор коду містить мінімальний набір інструкцій, що забезпечує нормальне функціонування порожньої форми у якості Windows-вікна. При створенні проекту програміст вносить до нього потрібний програмний код. Більш докладно структуру програмного модуля розглянуто в п. 2.1.3.

У вікно редактора коду, як і в інші вікна C++ Builder, вбудовано контекстну довідку. Щоб отримати довідку стосовно певного слова (терміну) коду, достатньо встановити курсор на це слово і натиснути клавішу <F1>.

Головне меню міститься у верхній частині основного вікна (див. рис. 2.1). Воно надає можливість доступу до всіх інструментальних засобів створювання, модифікації та керування проектом. Меню містить багато підменю (File, Edit, Search тощо). Слід звернути увагу лише на основні, найбільш використовувані опції:

Підменю **File** (Файл) містить команди для виконування операцій з проектами, модулями та файлами. Його основними опціями є:

New Application – створити новий проект;
New Form* – створити й долучити до проекту нову форму;
Save* – зберегти відкритий активний модуль (unit);
Save As – зберегти активний модуль (unit) під іншим ім’ям;
Save Project As – зберегти проект під іншим ім’ям;
Save All* – зберегти файл проекту та всі його модулі;
Close – закрити поточний модуль і відповідну форму;
Close All – закрити усі відкриті вікна проекту.

Примітка. Опції, позначені символом *, також можна реалізовувати за допомогою “гарячих” кнопок, розглянутих нижче в табл. 2.1.

Підменю **Edit** (Правка) містить команди редагування, роботи з областю обміну даними, відміни дій і керування відображенням компонентів. Його основними опціями є:

Undo, Redo – відмінити чи відновити останню зміну (еквівалентні дії викликаються комбінаціями клавіш: <Ctrl>+<Z> і <Shift>+<Ctrl>+<Z>);
Cut, Copy – вирізати чи лише скопіювати компоненти чи фрагмент тексту з модуля до буфера (<Ctrl>+<X> і <Ctrl>+<C>);
Paste – вставити з буфера дані (<Ctrl>+<V>), які були до цього скопійовані чи вирізані в буфер;
Delete – вилучити виокремлені компоненти форми чи тексту (клавіша <Delete>);
Select All – виокремити всі компоненти форми чи увесь текст поточного модуля;
Aline* – викликати вікно вирівнювання виокремлених компонентів форми;
BringToFront* – перемістити виокремлений компонент на передній план;
SendToBack* – перемістити виокремлений компонент на задній план;

Scale* – викликати вікно для пропорційного змінення масштабу виокремлених компонентів.

Примітка. Опції, позначені символом *, також можна реалізовувати за допомогою допоміжного меню, котре відкривається за натиснення правої кнопки миші на виокремленому компонентові.

Для решти підменю головного меню зазначимо лише область їхньої дії, оскільки при початковому вивченні C++ Builder та створенні нескладних проектів можна користуватися значеннями опцій за замовчуванням чи використовувати “гарячі” кнопки чи то комбінації клавіш клавіатури.

Підменю **Search** (Пошук) містить команди для пошуку (<Ctrl>+<F>) і заміни (<Ctrl>+<R>) фрагментів тексту, а також покрокового пошуку рядків з помилками компілювання. Також можна здійснювати пошук за номером сторінки чи за певною послідовністю символів. Окрім того, існує доволі зручний спосіб пошуку у документі. Просто натиснути <Ctrl>+<E> і почати набирати потрібну фразу для пошуку. По ходу набирання, будуть висвічуватись віднайдені результати. Для пошуку наступного результату, слід натиснути <F3>.

Підменю **View** (Вид) містить команди для виклику менеджера проектів, інспектора об’єктів (<F11>), браузера об’єктів, вікон налагодження та інших інформаційних утиліт.

Підменю **Project** (Проект) містить команди компіляції (<Ctrl>+<F9>) та конструювання проектів, а також встановлення опцій поточного проекту.

Підменю **Run** (Пуск) містить команди для запуску проекта на виконання (<F9>), засобів контролю покрокового виконування і налагоджування програм.

Підменю **Componet** (Компоненти) містить команди для створення та інсталяції нових компонентів на базі вже існуючих, а також для налаштовування конфігурації палітри компонентів.

Підменю **DataBase** (Бази даних) містить команди виклику інструментальних засобів для роботи з базами даних.

Підменю **Tools** (Інструменти) містить команди для налаштування середовища та виклику додаткових утиліт, у тому числі вікно редактора графічних зображень, інструменти обслуговування бази даних і вікно для редагування підменю Tools.

Підменю **Help** (Допомога) містить команди для виклику різних розділів довідкової системи. Зокрема відображує зміст довідкової системи, виконує пошук за ключовим словом, виводить довідку стосовно об’єктів та компонентів, а також містить посилання на адресу сторінок Borland в мережі Інтернет.

“Гарячі” кнопки інструментальних панелей призначені для швидкого запуску найчастіше використовуваних команд головного меню (див. рис. 2.3).

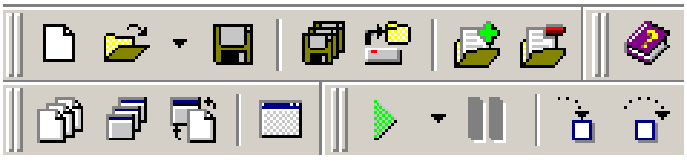


Рис. 2.3. Піктографічні кнопки

Клацання лівою кнопкою миші на якій завгодно “гарячій” кнопці спричинить виконання відповідної функції головного меню. Функції основних “гарячих” кнопок наведені у табл. 2.1.

Кнопки піктографічного меню можна редагувати, тобто вилучати й долучати нові кнопки, за допомогою опції Properties з допоміжного меню, котре відкривається за натиснення на цій панелі правої кнопки миші.

Таблиця 2.1

Функції “гарячих” кнопок

Кнопка	Назва та призначення
	*New* – створити новий елемент проекту (проект, форму, модуль тощо)
	*Open* – відкрити файл
	*Save* – зберегти файл на диску
	*SaveAll* – зберегти усі файли проекту на диску
	*Open project* – відкрити проект
	*Add file to project* – долучити файл до проекту
	*Remove file from project* – вилучити файл з проекту
	*Help* – виклик довідкової системи
	*Select unit from list* – обрати модуль зі списку модулів, пов’язаних з проектом
	*Select form from list* – обрати форму зі списку форм, пов’язаних з проектом
	*Toggle form/unit* – змінити місцями вікно форми й вікно модуля
	*New form* – створити нову порожню форму й долучити її до проекту
	*Run* – компілювати і виконати програму
	*Pause* – зреалізувати паузу в роботі програми
	*Trace info* – покрокове трасування програми із заходом до підпрограм
	*Step over* – те саме, але без заходу до підпрограм

Палітру компонентів сформовано у вигляді сукупності сторінок, на кожній з яких розміщуються компоненти, поєднані спільним функціональним призначенням (див. рис. 2.4). Щоб отримати доступ до певного компонента, слід спочатку клацанням миші обрати потрібну сторінку, а вже на цій сторінці обрати потрібний компонент і розмістити його на формі. Компоненти на палітрі подано у вигляді кнопок.

а

б

Рис. 2.4. Палітра компонентів:
а – сторінки Standard; б – сторінки Additional

Палітру компонентів можна налаштовувати відповідно до вимог розробника. Виклик відповідного діалогового вікна можна здійснити чи то командою **Component / Configure Palette**, чи опцією Properties з контекстного меню.

Спочатку розглянемо стандартні компоненти, які дозволяють створювати візуальний інтерфейс у програмних додатках.

Сторінка Standard

На цій сторінці палітри компонентів містяться базові для Windows елементи, наведені в табл. 2.2.

Таблиця 2.2

Компоненти сторінки Standard

Кнопка	Назва	Призначення
	`Frame` (Фрейм – рамка)	Окремий контейнер у вигляді окремого вікна для компонентів з можливостями наслідування
	`MainMenu` (Головне меню)	Панель команд головного меню. Ідентифікатори всіх команд меню визначаються властивістю `Items`, яка має доступ до кожної команди меню
	`PopupMenu` (Допоміжне меню)	Контекстне меню для форми чи для іншого компонента, котре зазвичай з'являється у вигляді окремого вікна після натискання правої кнопки миші. Саме для цього кожен компонент має властивість `PopupMenu`, в якій можна задати посилання на пов'язане з ним меню
	`Label` (Надпис – позначка)	Надпис на формі, текст якого задається властивістю `Caption`. Зазвичай використовується для пояснювальних надписів. Властивість `AutoSize` є ознакою того, що розмір поля визначається його вмістом. Значення `true` властивості `WordWrap` надає можливість перенесення тексту надпису по словах. Властивість `Alignment` визначає спосіб вирівнювання тексту

Продовження табл. 2.2

Кнопка	Назва	Призначення
	Edit (Однорядковий редактор)	Відображення, введення та виведення лише одного рядка тексту; цей рядок зберігається у властивості Text. Параметри шрифту: назва, розмір, колір, стиль визначається за допомогою властивості Font
	Memo (Багаторядковий текстовий редактор)	Багаторядкове текстове вікно для відображення, введення чи виведення тексту та значень даних програми; текст зберігається у властивості Lines
	Button (Кнопка)	Прямокутна кнопка, натискання якої ініціює задані дії у програмі; надпис на кнопці задається властивістю Caption
	CheckBox (Індикатор з прапорцем)	Квадратний індикатор дозволяє користувачу вмикати/вимикати опції програми, оскільки компонент передбачає два стани: “ввімкнено” і “вимкнено” (властивість State)
	RadioButton (Радіокнопка)	Кругла кнопка з двома станами і текстом опису, який специфікує її призначення. Радіокнопки – набір взаємовиключних варіантів вибору: лише одна кнопка може бути обрана на даний момент часу (позначається внутрішнім чорним кружечком), а з попередньо обраної кнопки вибір автоматично знімається. При натисканні радіокнопки властивість Checked змінюється і виникає подія OnClick
	ListBox (Список вибору)	Список текстових варіантів для вибору. Елементи списку містяться у властивості Items, а номер обраного під час виконання програми елемента – у властивості Itemlndex. Можна динамічно долучати, вилучати, вставляти і переміщувати елементи списку за допомогою методів Add, Delete, Insert та Append об’єкта Items. Значення true властивості Sorted встановлює сортування елементів списку за абеткою
	ComboBox (Комбінований список)	Список елементів (властивість Items) розкривається (випадає) для вибору. Значення обраного елемента надається властивості Text. Компонент поєднує в собі два компоненти: ListBox та Edit. Значення true властивості Sorted встановлює сортування елементів списку за абеткою
	Panel (Панель)	Платформа для розміщення компонентів. Може використовуватися для побудови смуги стану, панелей інструментів

Закінчення табл. 2.2

Кнопка	Назва	Призначення
	ScrollBar (Смуга прокручування)	Смуга прокручування з бігунком для керування переглядання видимої частини форми чи вмісту іншого компонента, приміром для переміщування всередині заданого інтервалу значень певного параметра. Значення властивостей Min та Max встановлюють інтервал припустимих переміщувань бігунка. Можна програмно встановлювати бігунок у позицію, задану значенням властивості Position
	RadioGroup (Група радіокнопок)	Є комбінацією групового вікна GroupBox і групи залежних радіокнопок RadioButton. Лише одну кнопку з групи може бути обрано. Перелік кнопок задається властивістю Items. Значення властивості ItemIndex зумовлює, яку саме радіокнопку обрано на цей момент. Можна групувати радіокнопки у кілька стовпчиків за допомогою властивості Columns
	GroupBox (Група елементів)	Контейнер, який поєднує на формі групу логічно пов'язаних компонентів
	ActionList (Список дій)	Містить список дій компонентів, таких як пункти меню чи кнопки

Сторінка Additional

На цій сторінці палітри компонентів розміщено додаткові компоненти, за допомогою яких до програмного додатка можна долучати спеціалізовані інтерфейсні елементи Windows, наведені в табл. 2.3.

Таблиця 2.3

Основні компоненти сторінки Additional

Кнопка	Назва	Призначення
	BitBtn (Кнопка з графікою)	Командна кнопка із зображенням бітового образу (піктограмою) і надписом. Властивість Kind містить набір стандартизованих стилів кнопки з відповідними надписами та графікою: OK, Cancel, Hello тощо. Властивість Glyph дозволяє обрати для зображення на кнопці який завгодно файл формату bmp
	SpeedButton (Кнопка з фіксацією та графікою)	Графічна швидка кнопка для створювання у програмі панелей з "гарячих" кнопок
	MaskEdit (Вікно маскованого редагування)	Дозволяє вводити текст (властивість Text) відповідно до заданого шаблона (властивість EditMask)

Закінчення табл. 2.3

Кнопка	Назва	Призначення
	`StringGrid` (Таблиця рядків)	Таблиця для відображення текстової інформації в її рядках та стовпчиках. Властивість `Cells` дозволяє програмне звертання до комірок компонента. Властивість `ColCount` задає кількість стовпчиків, а `RowCount` – кількість рядків
	`DrawGrid` (Таблиця рисунків)	Таблиця для відображення графічної інформації в її рядках та стовпчиках
	`Image` (Зображення)	Контейнер для відображення графіки: піктограм, рисунків та метафайлів
	`Shape` (Фігура)	Прості геометричні фігури – коло та еліпс, квадрат та прямокутник (можна із заокругленими кутами). Вигляд обраної геометричної фігури задається властивістю `Shape`, а колір та спосіб її забарвлення – двома вкладеними до `Brush` властивостями: `Color` та `Style`
	`Bevel` (Кромка)	Рамки та лінії з об'ємним виглядом. Властивість `Shape` визначає вигляд компонента, а значення властивості `Style` задає опуклу чи угнуту форму об'єкта
	`ScrollBox` (Вікно з прокручуванням)	Контейнер для відображення зі смугами прокручування за потреби
	`CheckListBox` (Список з індикаторами)	Компонент є комбінацією списку `ListBox` та індикаторів `CheckBox` і забезпечує вибір кількох опцій водночас
	`Splitter` (Відокремлювач)	Відокремлювач використовується для створення панелей зі змінними розмірами для відокремлення компонентів один від одного
	`StaticText` (Надпис з бордюром)	Надпис є подібний до компонента `Label`, але забезпечує можливість задавати стиль бордюру. Основна властивість – `Caption`
	`ControlBar` (Інструментальна панель)	Застосовується для розміщення компонентів інструментальної панелі
	`ApplicationEvents` (Події додатка)	Компонент перехоплює події на рівні додатка
	`Chart` (Діаграми та графіки)	Різноманітні тривимірні діаграми й графіки

Палітри компонентів відрізняються у різних версіях C++ Builder і можуть бути налаштовані відповідно до вимог розробника. Кількість компонентів вимірюється сотнями. У табл. 2.4 наведено ще деякі широко поширені

у застосуванні компоненти з різних сторінок палітри компонентів, на більшість з яких є посилання у наступних розділах підручника.

Таблиця 2.4

Деякі додаткові компоненти

Кнопка	Назва	Призначення	Сторінка
	`TabControl` (Сторінка із закладкою)	Сторінки із закладками, котрі може обирати користувач. Кількість закладок і надписи на них задаються властивістю `Tabs`. Автоматичне оновлення вмісту сторінок при перемиканні не відбувається, тому програміст має робити це самостійно	Win32
	`PageControl` (Багато-сторінкове вікно)	Сторінки із закладками для економії місця, якими можна керувати. Створення нових сторінок, редагування та вилучення виконується через контекстне меню клацанням правої кнопки миші та обранням відповідної команди. Властивість `Caption` кожної сторінки задає надпис на закладці	Win32
	`ImageList` (Список зображень)	Список зображень однакового розміру форматів BMP та ICO. Долучення нових зображень і редагування списку зображень здійснюється через відповідне вікно, відкриване подвійним клацанням по компонентові	Win32
	`RichEdit` (Вікно редактора у форматі RTF)	Багаторядкове вікно для форматування тексту в форматі RTF, яке дозволяє задавати різне форматування різних фрагментів тексту	Win32
	`DateTimePicker` (Введення дати і часу)	Календар, який розкривається і дозволяє обирати дату чи час. Властивості `Date` й `Time` містять значення дати й часу відповідно. Властивість `Kind` дозволяє обрати режим роботи чи то з датою, чи з часом	Win32
	`MonthCalendar` (Введення дати)	Дозволяє обирати дату з календаря. Властивість `Date` містить обрану дату	Win32
	`ProgressBar` (Індикатор)	Прямокутний індикатор, який зафарбовується зліва направо заданим кольором у міру виконування певного процесу в програмі. Властивості `Min` та `Max` задають інтервал значень індикатора. Властивість `Step` задає крок змінювання значень позиції індикатора (властивість `Position`)	Win32

Продовження табл. 2.4

Кнопка	Назва	Призначення	Сторінка
	TrackBar (Полоса з регулятором)	Шкала з позначками й регулятором поточної позиції (варіант лінійки прокручування). Властивості Min та Max задають інтервал значень шкали, причому властивість Position показує поточну позицію регулятора всередині заданого інтервалу. Кількість зображених позначок специфікує властивість Frequency. Те, на скільки позначок має пересунутись регулятор, коли користувач клацне мишею на самій шкалі (по обидва боки від регулятора) чи натисне клавіші PageUp та PageDown, задає значення властивості PageSize. Вигляд шкали можна змінювати за допомогою властивостей TickStyle та TickMarks	Win32
	StatusBar	Набір панелей у вигляді смуги стану у стилі Windows. Зазвичай ця смуга розміщується внизу форми. Властивість SimplePanel визначає наявність декількох панелей. Значення true цієї властивості означає, що уся смуга є єдиною панеллю, текст якої задається властивістю SimpleText. Значення false зазначає набір панелей, які задаються властивістю Panels. Текст таких панелей задається командою: `StatusBar1->Panels->Items[0]->Text="текст";` де 0 – номер самої першої панелі	Win32
	ToolBar	Контейнер для створення інструментальних панелей. Для розміщення на панелі інструментів кнопок ToolButton треба з контекстного меню обрати команду New Button (Нова кнопка) чи NewSeparator (Новий роздільник). Піктограми на кнопках можна задавати зі списку зображень компонента ImageList	Win32
	Timer (Таймер)	Використовується для програмування дій, які мають виконуватись з певною періодичністю чи то за заданого відтинку часу. Властивість Interval задає частоту, з якою має відбуватися подія Timer, значення 1000 цієї властивості відповідає одній секунді	System
	OleContainer (контейнер OLE)	OLE-контейнер для вбудовування і розміщення на формі OLE-об’єктів, створених в інших програмних додатках	System

Закінчення табл. 2.4

Кнопка	Назва	Призначення	Сторінка
	OpenDialog (Відкрити файл)	Виклик діалогового вікна Windows "Відкрити файл". Властивість FileName зберігає ім'я файла, а властивість InitialDir – обраний шлях до теки з файлом	Dialogs
	SaveDialog (Зберегти файл)	Виклик діалогового вікна "Зберегти файл". Властивості є подібні до компонента OpenDialog	Dialogs
	OpenPictureDialog (Відкрити рисунок)	Реалізує спеціальне вікно вибору графічних файлів. Властивості є подібні до компонента OpenDialog	Dialogs
	SavePictureDialog (Зберегти рисунок)	Реалізує спеціальне вікно зберігання графічних файлів. Властивості є подібні до компонента OpenDialog	Dialogs

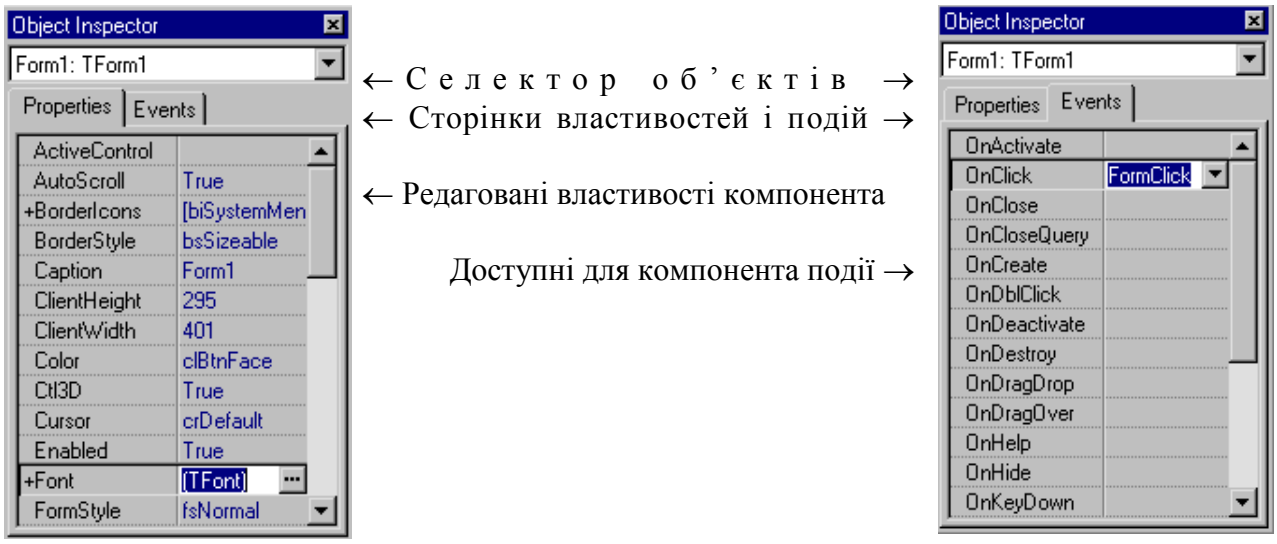
Усі компоненти можна поділити на дві групи: видимі (візуальні) і невидимі (невізуальні). Під час виконання програми візуальні компоненти (наприклад, кнопки Button, надписи Label тощо) видно на формі точно так само, як і при проектуванні форми. Невізуальні компоненти (наприклад, діалоги роботи із файлами OpenDialog та SaveDialog, системний таймер Timer, компоненти для роботи з базами даних: Table, DataSource, Query, ADOConnection та багато інших) є видимі як піктограми на формі під час проектування, але їх не видно під час виконання. Ці компоненти мають певну функціональність (наприклад, забезпечують доступ до даних, викликають стандартні діалоги Windows та ін.).

Якщо обрати будь-який компонент з палітри та розмістити на формі, його властивості та події автоматично покаже інспектор об'єктів.

Вікно інспектора об'єктів (Object Inspector) розташоване ліворуч вікна форми. Інспектор об'єктів забезпечує простий і зручний інтерфейс для змінювання значень властивостей компонентів і керування подіями, на які реагують об'єкти. Інспектор об'єктів має дві сторінки: *Properties* (властивості) і *Events* (події). Властивості компонентів (назва, колір, розмір тощо) задаються на сторінці Properties, а події (натискання клавіш миші чи її рух, натискання клавіш клавіатури, вибір пункту меню тощо) – на сторінці Events у вікні Object Inspector (рис. 2.5). Для того щоб у вікні інспектора об'єктів було відображено атрибути певного об'єкта, слід виокремити цей об'єкт, клацнувши на ньому лівою кнопкою миші, або обрати потрібний об'єкт із наявних на формі зі списку у верхній частині інспектора об'єктів.

Сторінка властивостей Properties

На цій сторінці (рис. 2.5, а) відображаються для виокремленого об'єкта основні властивості та їхні значення. *Властивості* є атрибутами компонента, що визначають його зовнішній вигляд і поведінку.



а **б**

Рис. 2.5. Вікна інспектора об'єктів:
а – сторінка властивостей Properties; б – сторінка подій Events

Значення властивостей можна змінювати. Приміром, якщо клацнути на формі, а тоді обрати її властивість `Caption`, то у вікні зі значенням цієї властивості `Form1` можна ввести новий текст, який і стане заголовком форми. Властивість `Color` має список можливих значень, який можна відкрити, клацнувши на кнопці у вікні цієї властивості. Якщо обрати колір `clRed`, то форма чи виокремлений об'єкт набуде червоного кольору. Усю палітру кольорів можна побачити, двічі клацнувши мишею у полі значення кольору.

Деякі з властивостей мають складну структуру. Клацання миші у полі значень властивості призводить до появи у цьому полі кнопки . Після натиснення цієї кнопки відкриється діалогове вікно, котре і дозволяє встановлювати нові значення цієї властивості. Наприклад, властивість `Font` дозволяє задавати атрибути тексту: шрифт, розмір, стиль та колір літер.

Усі властивості мають за замовчуванням певні значення при створенні компонента. Приміром, для компонента `Label` властивість `AutoSize` (Автомасштаб) має значення `true` (істина) без додаткових вказівок. Це означає, що розмір компонента підлаштовується під розмір його вмісту. Але можна задавати потрібний розмір за допомогою властивостей `Width` (довжина) та `Height` (висота), якщо при цьому змінити значення властивості `AutoSize` на `false` (хибність).

Якщо на формі треба розмістити кілька однотипних компонентів, то за замовчуванням їхніми іменами буде назва відповідного компонента, наприкінці якого зазначається його порядковий номер на формі. Наприклад, `Button1`, `Button2`, … чи `Memo1`, `Memo2`, …

Інспектор об'єктів відображає опубліковані (published) властивості компонентів. Крім published-властивостей, компоненти можуть і найчастіше мають загальні (public), опубліковані властивості, які доступні лише під час виконання проекту. Наприклад, для компонента `StringGrid` властивість звертання до комірок `Cells` є доступною лише програмно. Задавати нові значення властивостям можна не лише під час проектування форми, а й програмно, написавши відповідний код для видозміни властивостей компонента під час виконання проекту.

Сторінка подій Events

Ця закладка вікна інспектора об'єктів показує список подій, розпізнаваних компонентом, і дозволяє визначати реакцію виокремленого компонента на ту чи іншу подію (програмування для операційних систем з графічним користувальницьким інтерфейсом, зокрема, для різних версій Windows передбачає опис реакції проекту на ті чи інші події, а сама операційна система займається постійним опитуванням комп'ютера з метою виявлення настання якоїсь події). Кожен компонент має свій власний набір можливих подій.

Найбільш поширеними у застосуванні подіями є такі:

- ✓ `OnClick` – клацання лівою кнопкою миші на об'єкті;
- ✓ `OnDblClick` – подвійне клацання лівою кнопкою миші на об'єкті;
- ✓ `OnChange` – внесення змін до вмісту об'єкта;
- ✓ `OnCreate` – подія відбувається при створенні форми;
- ✓ `OnActivate` – подія відбувається при активації форми;
- ✓ `OnMouseDown` – натиснення кнопки миші над компонентом;
- ✓ `OnMouseUp` – відпускання кнопки миші над компонентом;
- ✓ `OnMouseMove` – переміщення миші над компонентом;
- ✓ `OnKeyPress` – натиснення символьної клавіші клавіатури;
- ✓ `OnKeyDown` – натиснення будь-якої клавіші клавіатури;
- ✓ `OnKeyUp` – відпускання будь-якої клавіші клавіатури;
- ✓ `OnEnter` – подія відбувається в момент набуття компонентом фокуса;
- ✓ `OnExit` – подія відбувається в момент втрати компонентом фокуса;
- ✓ `OnStartDrag` – подія відбувається, коли користувач розпочинає перетягування компонента;
- ✓ `OnDragDrop` – подія відбувається в момент відпускання перетягуваного компонента над даним компонентом;
- ✓ `OnEndDrag` – подія відбувається в момент завершення перетягування компонента.

Назви подій наведені у лівій колонці сторінки Events. Для того, щоби створити функцію відгуку на певну подію, треба обрати на формі за допомогою миші необхідний компонент, відкрити сторінку подій інспектора об'єктів і двічі клацнути лівою клавішею миші на колонку поруч з цією подією. При цьому C++ Builder автоматично згенерує шаблон функції відгуку на подію, її прототип запише до відповідного заголовного файла, відкриє редактор коду, в якому покаже текст порожньої функції, і зпозиціонує курсор усередині її операторних дужок { ... } для подальшого введення програмного коду. Наприклад, для події `OnClick` компонента `Button1`, розміщеного на формі `Form1`, шаблон відповідної функції в Unit1.cpp матиме вигляд:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
}
```

а її заголовок в Unit1.h:

```
void __fastcall Button1Click(TObject *Sender);
```

Багато властивостей і методів є притаманні більшості компонентів, приміром: ім'я, розміри, доступність, видимість тощо. У табл. 2.5 наведено найчастіше уживані властивості, а в табл. 2.6 – методи функціонування компонентів.

Що стосується методів, вони є функціями, пов'язаними з компонентами, і по суті є частиною цих об'єктів. Виклик методів і звертання до властивостей компонентів здійснюють за допомогою операції вибору елемента "->", наприклад:

```
Edit1->Show();
```

Таблиця 2.5

Основні властивості стандартних компонентів

Властивість	**Пояснення**	**Компоненти з такою властивістю**
AutoSize	Автомасштаб: якщо встановити в 1 (true), розмір компонента автоматично адаптується до розміру символів тексту	Label, Edit, Panel, Image та інші
Caption	Надпис	Button, Label та інші
Color	Колір компонента	Більшість видимих
Count	Кількість рядків списку	Memo, ListBox
Enabled	Доступність до компонента. Визначає, чи реагує компонент на події, пов'язані з мишею, клавіатурою і таймером. Значення false блокує доступ, а сам компонент набуває переважно більш світлого кольору	Усі видимі
Font	Параметри шрифта	Button, Label, Edit, Memo та інші
Glyph	Зображення у форматі bmp, яке можна винести на компонент	BitBtn, SpeedButton
Height	Висота компонента в пікселях	Усі видимі
ItemIndex	Номер обраного елемента списку. Нумерація розпочинається з 0	ListBox
Items	Список можливих значень списку	ListBox, ComboBox, MainMenu та інші
Left	Розміщення лівого краю компонента відносно форми в пікселях. Верхній лівий кут форми має координати (0, 0)	Усі видимі
Lines	Рядки вікна для введення чи редагування тексту. Нумерація рядків розпочинається з 0	Memo, RichEdit
Name	Ім'я компонента	Усі
ScrollBars	Встановлення лінійок прокручування у вікні Memo. Можливі значення цієї властивості: None, Vertical, Gorizontal, Both (відсутні, вертикальна, горизонтальна, обидві)	Memo, StringGrid та інші

Закінчення табл. 2.5

Властивість	Пояснення	Компоненти з такою властивістю
`Sorted`	Якщо встановити в 1 (`true`), усі елементи списку буде відсортовано за зростанням	`ListBox`, `ComboBox`
`Text`	Текст у вікні редагування	`Edit`, `ComboBox` та інші
`Top`	Розміщення верхнього краю компонента відносно форми в пікселях. Верхній лівий кут форми має координати (0, 0)	Усі видимі
`Visible`	Видимість: якщо встановити в 0 (`false`), компонент стане невидимим на формі	Усі видимі
`Width`	Ширина компонента в пікселях	Усі видимі

Таблиця 2.6

Основні методи стандартних компонентів

Метод	Пояснення	Для яких компонентів застосовується
`int` **`Add`** `(AnsiString S)`	Долучення рядка `S` у кінець списку. Метод повертає індекс цього рядка, наприклад: `int n=Memo1->Lines->Add(S);`	`Memo`, `ListBox`, `RadioGroup`, `RichEdit` та інші
`Clear``()`	Очищення вікна	`Edit`, `Memo` та інші
`Delete``(int n)`	Вилучення рядка з номером `n`, наприклад: `Memo1->Lines->Delete(n);`	`Memo`, `ListBox`, `RadioGroup`, `RichEdit` та інші
`Hide``()`	Приховання компонента	Більшість видимих
`Insert``(int n, AnsiString S)`	Вставлення рядка `S` у позицію `n`, наприклад: `Memo1->Lines->Insert(n, S);`	`Memo`, `ListBox`, `RadioGroup`, `RichEdit` та інші
`LoadFromFile` `(AnsiString fname)`	Вивантаження вмісту із зазначеного файла до поля компонента	`Memo`, `RichEdit` та інші
`SaveToFile` `(AnsiString fname)`	Зберігання вмісту компонента у файлі із зазначеним ім'ям `fname`	`Memo`, `RichEdit` та інші
`SelectAll``()`	Виокремлення всього тексту	`Memo`, `RichEdit` та інші
`SetFocus``()`	Встановлення фокуса (курсора) на цей компонент	Більшість видимих
`SetText``(char *S)`	Долучення тексту `S` у поточну позицію курсора, наприклад: `Memo1->Lines->SetText(S);`	`Memo` та інші
`Show``()`	Показ прихованого компонента	Більшість видимих

2.1.2 Структура C++ Builder-проекту

Програма на C++ складається з оголошень (змінних, констант, типів, класів, функцій) та опису функцій. З-посеред функцій головною є функція `main` для консольних додатків (яка працює з WIN32) чи то `WinMain` – для додатків Windows. Головна функція виконується після початку роботи програми. Щоб побачити текст головного файла, слід виконати команду **Project/View Source**.

Програму в C++ Builder частіше називають проектом чи додатком. Зазвичай кожен проект складається з кількох файлів, основними з яких є:

✓ головний файл програми, який поєднує всі файли проекту. Він створюється автоматично і, за замовчуванням, його ім'я – Project1.bpr;

✓ файли форми з інформацією про властивості усіх компонентів на цих формах. Їхні імена, за замовчуванням, – Unit1.dfm, Unit2.dfm, …;

✓ файли реалізації модуля з програмним кодом, так звані модулі. Це файли, в яких прописуються тексти функції опрацювання подій для компонентів форми. Імена файлів реалізації, за замовчуванням, – Unit1.cpp, Unit2.cpp, …;

✓ заголовний файл с розширенням .h, який зберігає оголошення класу цієї форми. Весь основний текст цього файла формується автоматично, але іноді виникає потреба долучати до нього оголошення функцій, типів, констант та змінних користувача. Відкрити цей файл у редакторі кодів можна, клацнувши у вікні з файлом реалізації модуля правою кнопкою миші та обравши з контекстного меню команду **Open Sourse/Header File** або скориставшись клавіатурною комбінацією <Ctrl> + <F6>.

Імена файла реалізації (модуля) і заголовного файла збігаються. Якщо модуль пов'язаний з формою, то його ім'я збігається і з ім'ям файла форми (з розширенням dfm). Це ім'я задають за першого зберігання проекту. За замовчуванням C++ Builder пропонує ім'я Unit1.

Заголовний файл при створенні проекту має вигляд:

```
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
// Тут можуть розміщуватися додаткові директиви препроцесора
// (зокрема include), які долучають потрібні заголовні файли
//---------------------------------------------------------
class TForm1 : public TForm// Оголошення класу форми TForm1
{   __published:     // IDE-managed Components
// Розташовані на формі компоненти
        TLabel *Label1;
        TButton *Button1;
        void __fastcall Button1Click(TObject *Sender);
private:             // User declarations
// Закритий розділ класу. Тут можуть розміщуватися оголошення типів, змінних,
// функцій, які входять до класу форми, але є недоступні для інших модулів
```

```
public:                 // User declarations
// Відкритий розділ класу. Тут можуть розміщуватися оголошення типів, змінних,
// функцій, які входять до класу форми, доступні для інших модулів
__fastcall TForm1(TComponent* Owner);
}; //---------------------------------------------------
extern PACKAGE TForm1 *Form1;
// Тут можуть розміщуватися оголошення типів, змінних, функцій,
// котрі не долучаються до класу форми; доступ до них з інших блоків
// є можливий лише за дотримання певних додаткових умов
#endif
```

Розпочинається заголовний файл рядками, перший символ яких є #. З цього символу розпочинаються директиви препроцесора. З-посеред них найбільш важливою директивою є `include`, яка долучає до модуля потрібний файл. Більш докладно заголовні файли розглянуто у розд. 9. Приміром, якщо в програмі використовуватимуться математичні функції, слід долучити відповідну математичну бібліотеку, тобто власноруч прописати директиви:

```
#include <math.h>    // Долучення заголовного файла math.h
#include <Math.hpp>  // Долучення заголовного файла Math.hpp
```

Після директив препроцесора у заголовному файлі йде опис класу форми – `TForm1`, в якому розділ `__published` заповнюється автоматично при проектуванні форми. У вищенаведеному прикладі оголошено вказівники на два компоненти: `Label1` класу `TLabel` та `Button1` класу `TButton`. Також оголошено функцію `Button1Click` – програмний відгук на подію натискання кнопки `Button1`. Розділи `private` та `public` заповнюються за потреби автором програми. Всі об'єкти, оголошені у розділі `public`, будуть доступними для інших класів та модулів. Те, що оголошено у розділі `private`, є доступним лише у межах відповідного модуля.

Нижче за розділом PACKAGE можна розмістити оголошення типів, змінних, функцій, доступ до яких буде можливим і з інших модулів, але вони не долучатимуться до класу форми.

Файли реалізації (cpp-файли) є основою проектів у C++. Вони містять програмний код усіх функцій, оголошених у відповідному заголовному файлі. Окрім того, у cpp-файлі можна записувати функції, прототипи яких є відсутні у h-файлі, однак їхня видимість у такому разі обмежується цим cpp-файлом.

Розпочинається cpp-файл директивою препроцесора `#include <vcl.h>`, яка долучає відповідний h-файл з описом класів бібліотеки компонентів. За нею розміщена директива `#pragma hdrstop`, призначена для обмеження списку заголовних файлів, доступних для попередньої компіляції.

Розглянемо текст *файла реалізації*:

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
```

```
// Тут можуть розміщуватися додаткові директиви препроцесора
// (зокрема include), які не долучаються до файла автоматично.
//---------------------------------------------------
TForm1 *Form1; // Оголошення класу форми TForm1
//---------------------------------------------------
// Виклик конструктора форми Form1
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{ // Тут можна розмістити команди,
  // які мають виконуватися при створенні форми.
}
// Тут можуть розміщуватися глобальні оголошення типів, змінних,
// доступ до яких є можливий з будь-якої функції цього модуля.
// Окрім того, тут має бути прописано реалізації всіх функцій,
// оголошених у заголовному файлі, а також може бути
// прописано реалізації функцій, не оголошених раніш.
//---------------------------------------------------
// Приклад реалізації функції, яка здійснюватиме закриття форми,
// при натисненні лівої кнопки миші на компоненті Button1.
void __fastcall TForm1::Button1Click(TObject *Sender)
{ Close();
}
```

Звернімо увагу на те, що фігурні та круглі дужки мають у C++ різне призначення. По-перше, у круглих дужках записуються параметри функцій, а, по-друге, круглі дужки використовуються за потреби при записуванні арифметичних виразів. Фігурні дужки називають операторними; у C++ вони є аналогом begin/end інших мов. Відкритою фігурною ({) дужкою розпочинається кожна функція, у тому числі й функції відгуку на події, а закрита фігурна дужка (}) ставиться після останньої команди функції. Окрім того, фігурні дужки використовуються всередині функцій для поєднування групи команд відносно заданої команди, приміром якщо в циклі слід виконувати понад одну команду.

Примітка. C++ Builder надає зручну можливість перевірити чи є закритою та чи інша фігурна дужка. Для цього треба підвести до неї курсор і натиснути <Alt> + <[>. Якщо вона є закритою, то курсор переміститься до відповідної закритої дужки.

2.1.3 Послідовність створювання програмного проекту в C++ Builder

Створювання програмних проектів у C++ Builder передбачає створення форми, розміщування на ній потрібних компонентів, визначення їхніх властивостей та створювання програмних відгуків на задані дії.

Послідовність роботи може бути такою:

✓ запустити C++ Builder. При запуску автоматично створюється новий проект. Якщо C++ Builder вже працює для створення нового проекту слід з головного меню обрати опцію **File/New Application**;

✓ винести на порожню форму з палітри компонентів усі потрібні компоненти для створення діалогового вікна програми;

✓ задати на вкладці Properties (властивості) інспектора об'єктів усі потрібні властивості для компонентів, розташованих на формі;

✓ створити потрібні відгуки на події для розташованих на формі компонентів. Це можна зробити за допомогою вкладки Events (події) інспектора об'єктів чи то безпосередньо з форми, клацнувши двічі по компонентові. Не можна самостійно прописувати заголовки відгуків чи вилучати автоматично створені;

✓ у вікні редактора кодів заповнити відгуки на події кодом, а також написати текст усіх потрібних програм. Перехід поміж вікном форми і вікном редактора можна здійснити клавішею <F12>;

✓ зберегти створений проект за допомогою команди головного меню **File/Save Project All** (рис. 2.6);

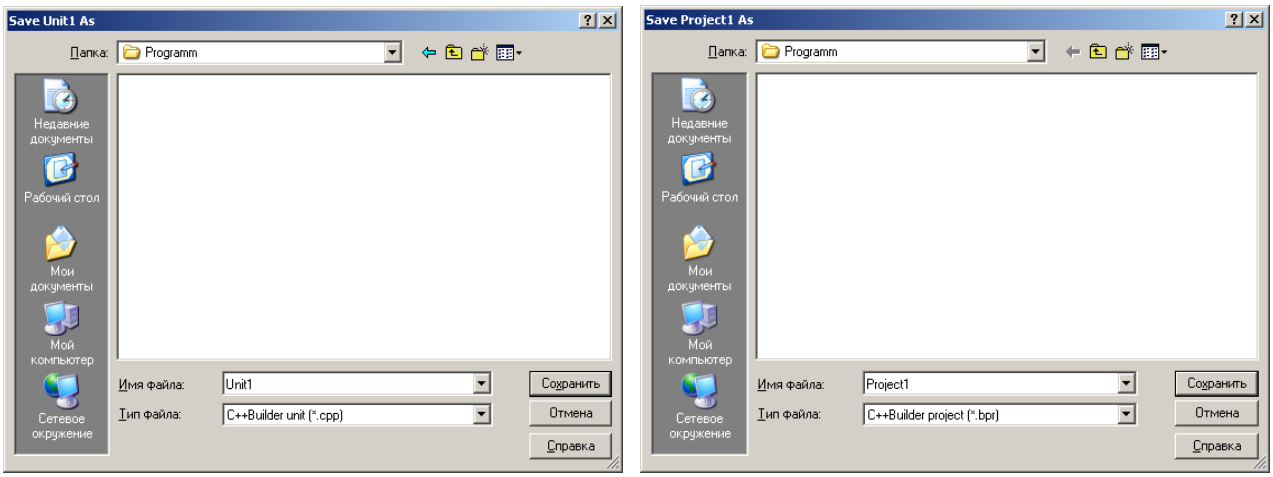


Рис. 2.6. Вікна зберігання файлів проекту

✓ запустити проект на виконання одним зі способів: натиснути клавішу <F9>, виконати команду меню **Run / Run**, натиснути відповідну кнопку на панелі інструментів;

✓ якщо помилок немає, з'явиться форма без координатної сітки. Якщо ж помилки є, то рядок з помилкою висвічуватиметься червоним кольором, а повідомлення компілятора, розташоване під вікном редактора коду, повідомить, якої саме помилки було припущено. Характерні помилки компіляції розглянуто у розд. 15. Згадані помилки слід виправити, зберегти проект і знову запустити проект на виконання;

✓ провести тестові розрахунки, для чого виконати відповідні дії (вводити дані, натискати потрібні кнопки тощо). Якщо при виконанні програми були виявлені алгоритмічні помилки їх, зрозуміло, слід виправити. Для довчасного припинення виконання програми, приміром, за її зависання, слід виконати команду **Run / Program Reset**;

✓ доволі часто на практиці при налагодженні програми виникає потреба продивитися проміжкові результати обчислень, приміром за обчислень у циклах сум доданків. Для цього існують спеціальні "гарячі" кнопки покрокового

виконання програми (див. табл. 2.1), дію яких можна також відтворити відповідними командами меню **Run** чи то функціональними клавішами клавіатури <F7> (покрокове трасування програми із заходом до підпрограм ) і <F8> (те саме, але без заходу до підпрограм, – ). Для переглядання результатів покрокового виконання програми слід відкрити відповідне віконце командою **Run / Add Watch**. Щоб покрокове виконання програми здійснювати, розпочинаючи з певної команди програми, цю команду слід позначити так званою точкою зупину (Breakpoint) за допомогою команди меню **Run / Add Breakpoint** чи клавішею <F5>;

✓ після налагодження й тестового виконання програми здійснити потрібні обчислення і вийти з C++ Builder. Сеанс завершено.

Рекомендуємо взяти за правило: перед кожним запуском програми обов'язково її зберігати.

Приклади створювання програмних проектів

Приклад 2.1 Створити програму, в якій можна буде встановлювати колір форми натисканням кнопки. Слід передбачити можливість установлення чотирьох кольорів: жовтого, синього, зеленого, червоного.

Розв'язок. На вікно форми слід встановити такі компоненти: один надпис (`Label`) і чотири кнопки (`Button`). Окрім того, створимо ще одну кнопку для виходу з програми. Дані компоненти розташовано на палітрі компонентів на вкладці Standard (див. табл. 2.2).

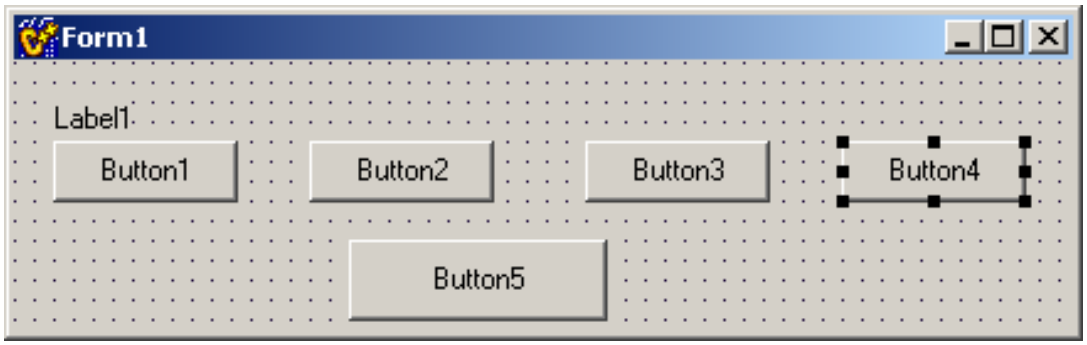
Для розміщення кожного компонента на форму треба:

1) на палітрі компонентів обрати потрібний компонент (клацанням лівої кнопки по його піктограмі);

2) клацнути лівою кнопкою миші у тому місці форми, де компонент має міститися;

3) за потреби відкоригувати більш точно місце розташування та розміри компонентів і самої форми. Це можна здійснити чи то за допомогою миші, чи задавши точні значення властивостей `Top`, `Left`, `Width` та `Height` компонентів у інспекторі об'єктів.

Після того як ці дії повторити для розміщення всіх потрібних компонентів, форма набуде вигляду



Примітка. Зауважимо, що за потреби розміщення на формі декількох компонентів одного типу, є зручна можливість зробити це швидко, скористува-

вшись таким методом. На панелі компонентів обрати потрібний компонент та утримуючи <Shift> клацнути на нього. Далі при кожному клацанні всередині форми, на ній з'являтиметься новий компонент.

Надписи, розмір, місце розташування на формі – все це властивості (Properties) компонентів, які можна встановити за допомогою інспектора об'єктів (Object Inspector). Як вже зазначалось, щоб змінити властивості компонента, його треба виокремити, клацнувши по компонентові лівою кнопкою миші, потім перейти до вікна інспектора об'єктів, на закладці Properties обрати потрібну властивість і встановити її нове значення. Отже, встановимо властивості компонентів згідно до таблиці, наведеної нижче.

Ім'я компонента	Властивість	Значення властивості	Пояснення
Button1	Caption	Жовтий	Надписи на кнопках
Button2	Caption	Синій	
Button3	Caption	Зелений	
Button4	Caption	Червоний	
Button5	Caption	Вихід	Надпис на кнопці
	Height	50	Висота кнопки
	Width	170	Ширина кнопки
Label1	Alignment	taCenter	Горизонтальне вирівнювання тексту
	AutoSize	false	Автоматичне змінювання розміру компонента заборонено
	Caption	"Змінення кольору форми"	Текст надпису
	Color	clBlack	Колір фону компонента
	Font->Color	clWhite	Колір шрифту надпису
	Font->Size	14	Розмір шрифту надпису
	Height	50	Висота компонента
	Layout	tlCenter	Вертикальне вирівнювання тексту
	Width	400	Ширина компонента
Form1	Caption	Приклад програми № 1	Надпис на формі

Після створення форми можна перейти до програмування. Колір форми має змінюватись при клацанні (Click) по відповідних кнопках. Для цього треба створити програми відгуків на подію OnClick для відповідних кнопок подвійним клацанням по цих кнопках. Після цього C++ Builder *автоматично* створить шаблон-заготовку в редакторі коду. Всі команди, які мають виконуватись при клацанні по кнопці, треба записати поміж операторних дужок { }. Програмний відгук на подію OnClick для кнопки Button1 остаточно матиме вигляд:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{  Form1->Color = clYellow;
}
```

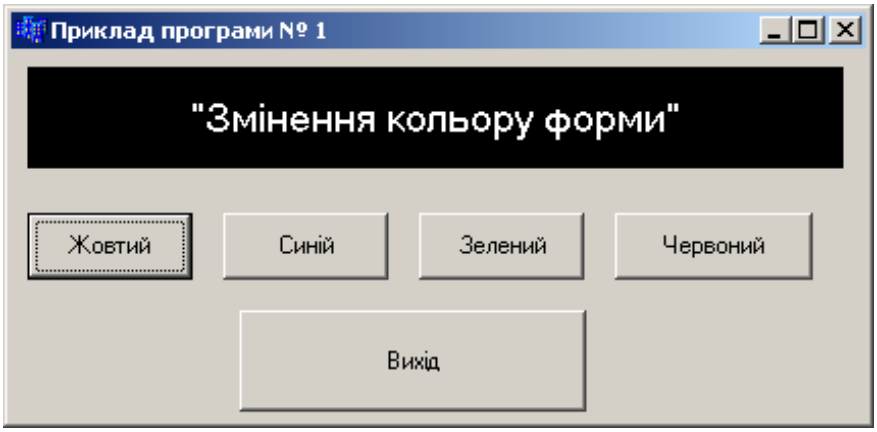
Програмний код для кнопок `Button2`, `Button3`, `Button4` відрізнятиметься від вищенаведеного лише кольором. Синій колір задається як `clBlue`, зелений – `clGreen`, червоний – `clRed`.

Для кнопки `Button5` (Вихід) програмний код міститиме команду закриття активної форми `Close()`:

```
void __fastcall TForm1::Button5Click(TObject *Sender)
{  Close();
}
```

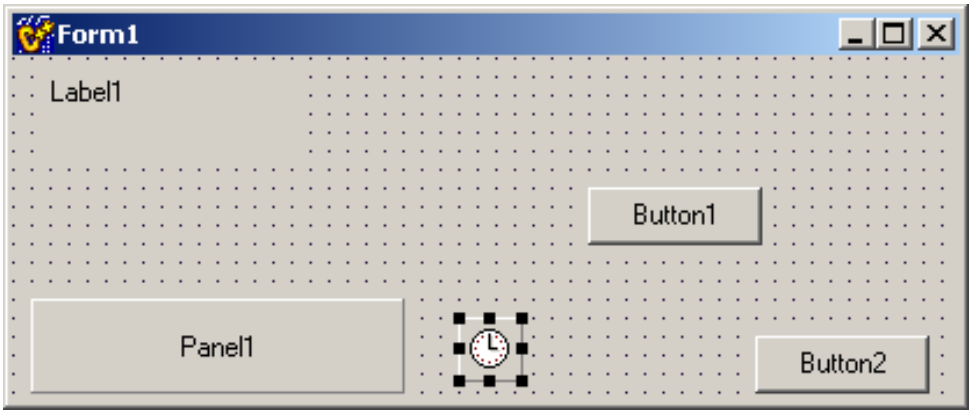
Перед запуском на виконання програмний проект слід зберегти в *окремій* теці командою **File/SaveAll** (чи то через ярлик, чи клавішами <Shift> + <Ctrl> + <S>). Внаслідок таких дій почергово з'являться два діалогових вікна, в яких треба у власноруч створеній новій теці зберегти і Unit1 і Project1.

Для запуску проекту слід виконати команду меню **Run/Run** чи то натиснути на клавіатурі кнопку <**F9**>. Унаслідок цього має з'явитися наведена нижче форма проекту (це означає, що програму написано без помилок).



Приклад 2.2 Створити програму, в якій зреалізувати кнопку-"втікача", тобто кнопку, яка "втікатиме" за намагання натиснути її, окрім того, вивести поточну дату і час на форму.

Розв'язок. Розташуємо на формі один надпис (`Label`), одну панель (`Panel`), дві кнопки (`Button`) і компонент таймер (`Timer`). Перші три компоненти містяться на палітрі компонентів на вкладці Standard (див. табл. 2.2), а компонент `Timer` – на вкладці System (див. табл. 2.4). Після розташування всіх компонентів форма матиме вигляд



Задамо властивості компонентів, наведені в таблиці.

Ім'я компонента	Властивість	Значення властивості	Пояснення
Form1	Caption	Приклад програми № 2	Надпис на формі
	Height	210	Висота форми
	Width	695	Ширина форми
Panel1	Caption	–	Надпис на панелі
Button1	Caption	Натисни!	Надпис на кнопці
Button2	Caption	Вихід	Надпис на кнопці
Timer1	Interval	1000	Інтервал таймера, заданий у мілісекундах
Label1	Alignment	taCenter	Горизонтальне вирівнювання тексту
	AutoSize	false	Автоматичне змінювання розміру заборонено
	Caption	Спробуй натиснути кнопку	Текст надпису
	Color	clBlue	Колір компонента
	Font->Color	clWhite	Колір шрифту надпису
	Font->Size	14	Розмір шрифту надпису
	Height	50	Висота компонента
	Layout	tlCenter	Розміщення тексту по вертикалі
	Width	500	Ширина компонента

У віконці інспектора об'єктів на вкладці подій (Events) задамо шаблон для відгуку на подію OnMouseMove (подія виникає за наведення на кнопку покажчика миші) для кнопки Button1. Для цього треба такі кроки:

1) виокремити компонент;
2) перейти до вікна Object Inspector;
3) відкрити вкладку Events;
4) двічі клацнути лівою кнопкою миші напроти назви потрібної події.

Після виконання наведених чотирьох кроків C++ Builder *автоматично* створить шаблон-заготовку в редакторі коду. Після того як впишемо поміж операторних дужок наступний програмний код, програма відгуку набуде вигляду

```
void __fastcall TForm1::Button1MouseMove(TObject *Sender,
                                TShiftState Shift, int X, int Y)
{ if (Button1->Left <= 0)
     Button1->Left = Form1->Width - Button1->Width - 10;
   else  Button1->Left = Button1->Left - 5;  }
```

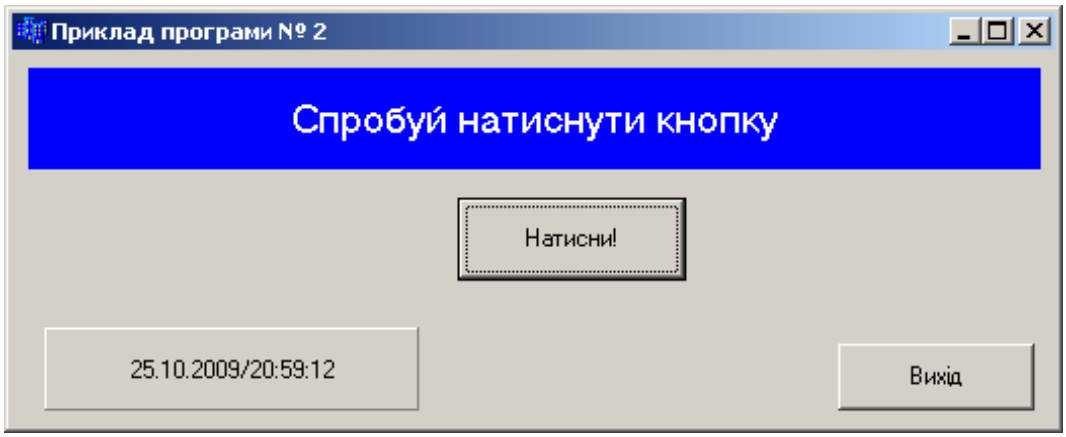
Отже, кнопка "втікає", залишилось вивести дату й час на панель. Задамо для компонента Timer шаблон для події OnTimer (подія відбувається із заданим у властивості Interval інтервалом часу), в якому запишемо програмний код:

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{ Panel1->Caption=Now().DateString() + "/" + Now().TimeString();}
```

Пояснення. Функція `Now()` повертає поточні дату й час. Метод `DateString()` перетворює дату на текст, метод `TimeString()` – час на текст. Докладніше ці функції розглянуто в розд. 10.

Перед запуском на виконання програмний проект слід зберегти в *окремій* теці командою **File/SaveAll** (чи то через "гарячу" кнопку, чи клавішами <Shift>+<Ctrl>+<S>). Внаслідок таких дій почергово з'являться два діалогових вікна, в яких треба у власноруч створеній новій теці зберегти і Unit1 і Project1. Для запуску проекту на виконання слід здійснити команду меню **Run/Run** чи натиснути на клавіатурі кнопку <F9> чи "гарячу" кнопку.

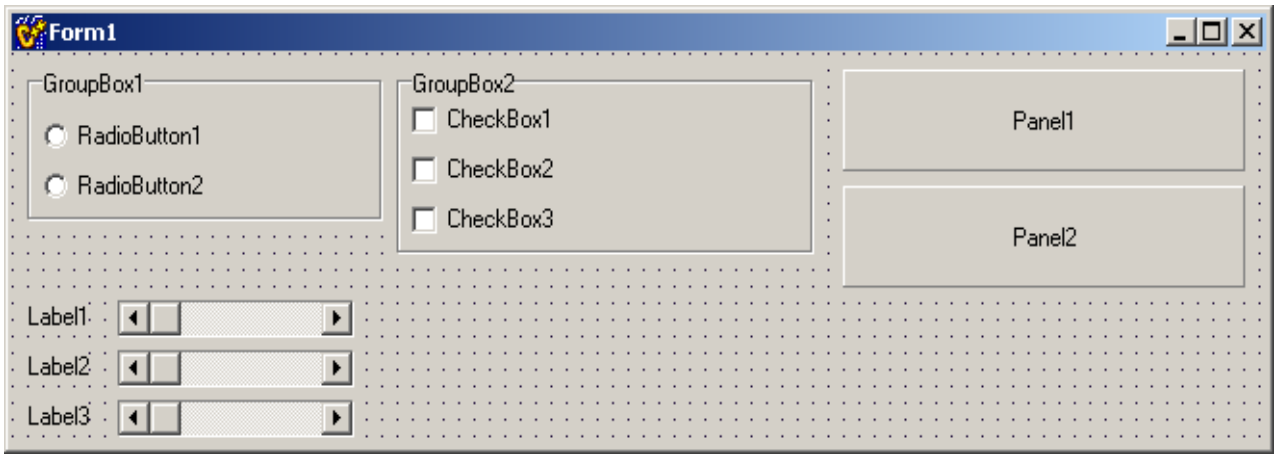
Вікно форми під час роботи проекта матиме вигляд



Приклад 2.3 Створити програму, в якій, залежно від вибору користувача, можна встановлювати колір першої чи другої панелі. Здійснити можливість змішувати кольори: задавати складові трьох основних кольорів: червоного, зеленого, синього. Надати можливість вибору дозволу/заборони змінювання певної складової кольорів.

Розв'язок. Із вкладки Standard на вікно форми слід встановити такі компоненти: три надписи (`Label`), два компоненти `GroupBox`, дві панелі (`Panel`), три смуги прокручування (`ScrollBar`). На компонент `GroupBox1` розмістимо два компоненти `RadioButton`, а на `GroupBox2` – три компоненти `CheckBox`.

Після розташування всіх компонентів форма матиме вигляд



Задамо властивості компонентів згідно таблиці, поданої нижче.

Компонент	Властивість	Значення властивості	Пояснення
Form1	Caption	Приклад програми № 3	Надпис на формі
Label1	Caption	R	Текст надпису
Label2	Caption	G	Текст надпису
Label3	Caption	B	Текст надпису
GroupBox1	Caption	Вибір панелі	Надпис компонента
	Height	90	Висота компонента
GroupBox2	Caption	Дозволити складову кольору	Надпис компонента
	Height	90	Висота компонента
RadioButton1	Caption	Панель № 1	Надпис компонента
	Checked	true	Стан вибору
RadioButton2	Caption	Панель № 2	Надпис компонента
ScrollBox1, ScrollBox2, ScrollBox3	Max	255	Верхня межа прокручування
	Width	550	Ширина смуги
CheckBox1	Caption	Червоний	Надпис компонента
	Checked	true	Стан вибору
CheckBox2	Caption	Зелений	Надпис компонента
	Checked	true	Стан вибору
CheckBox3	Caption	Синій	Надпис компонента
	Checked	true	Стан вибору
Panel1	Caption	Панель № 1	Надпис панелі
Panel2	Caption	Панель № 2	Надпис панелі

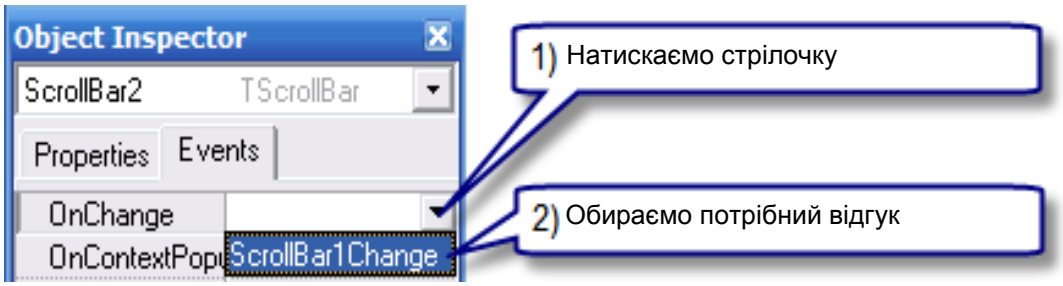
При змінюванні позиції повзунка смуги прокручування виникатиме подія OnChange. У нашому прикладі треба, щоб у відгук на цю подію комп'ютер змінював колір панелі Panel відповідно до вибору перемикача RadioButton. Новий колір визначається всіма трьома смугами прокручування: позиція повзунка кожної впливає на кількість відповідної складової частини кольору у результуючому кольорі при змішуванні. Для змішування трьох базових кольорів використовується функція RGB (Red-Green-Blue), параметрами якої є позиції повзунків кожного з трьох кольорів.

Задамо шаблон для відгуку на подію OnChange для смуги прокручування ScrollBar1. Коли виконаємо чотири кроки, наведені у прикладі 2.2, матимемо шаблон у редакторі коду. Впишемо поміж операторних дужок такий програмний код:

```
void __fastcall TForm1::ScrollBar1Change(TObject *Sender)
{ if(RadioButton1->Checked)
   Panel1->Color = RGB(ScrollBar1->Position,
                       ScrollBar2->Position,ScrollBar3->Position);
  else Panel2->Color = RGB(ScrollBar1->Position,
                       ScrollBar2->Position,ScrollBar3->Position);
}
```

Програми для відгуку на подію `OnChange` для компонентів `ScrollBar1` та `ScrollBar2` є аналогічні, але щоб не писати зайвого коду, зазначимо для `ScrollBar2` вже наявну програму-відгук `ScrollBar1Change`. Для цього:

- ✓ виокремимо компонент `ScrollBar2`;
- ✓ у вікні Object Inspector перейдемо до вкладки Events;
- ✓ оберемо для події `OnChange` зі списку `ScrollBar1Change`:



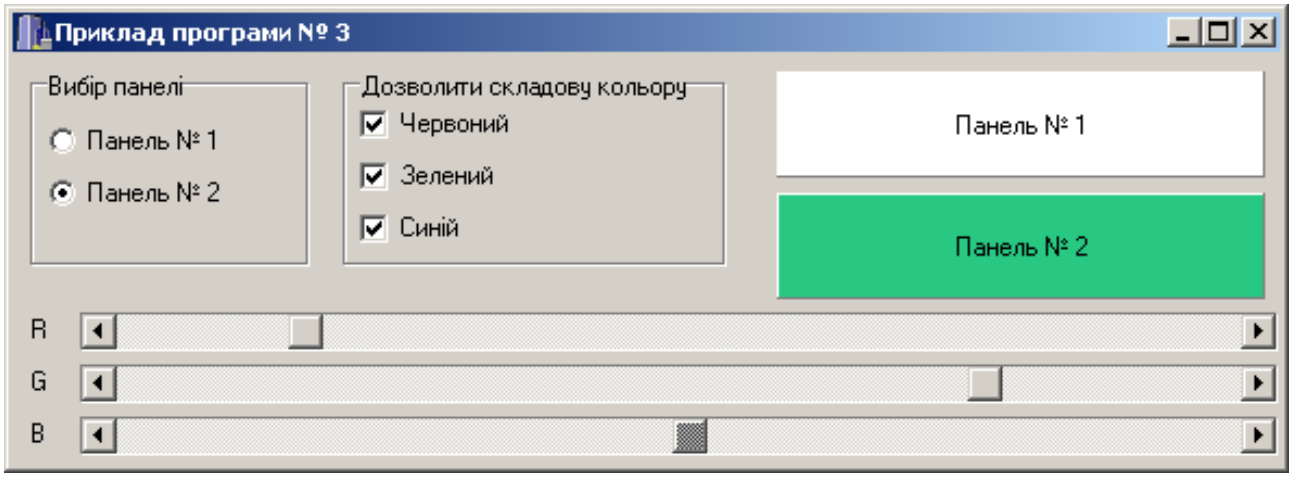
Аналогічно встановимо для смуги прокручування `ScrollBar3` програму-відгук для події `OnChange`.

Щоб дозволити чи заборонити змінення будь-якої складової кольорів, на формі є три відповідні "прапорці" (`CheckBox`). Якщо "прапорець" обрано (властивість `Checked`), то змінювати складову кольору можна, в іншому разі – ні. Стан "прапорця" змінюється за його натискання. Отже, для компонентів `CheckBox1`, `CheckBox2`, `CheckBox3` створимо програми опрацювання події `OnClick`, в яких запишемо:

```
для CheckBox1:  ScrollBar1->Enabled = CheckBox1->Checked;
для CheckBox2:  ScrollBar2->Enabled = CheckBox2->Checked;
для CheckBox3:  ScrollBar3->Enabled = CheckBox3->Checked;
```

Після написання коду програми слід зберегти (**File/SaveAll** ) і запустити (<**F9**>) проект.

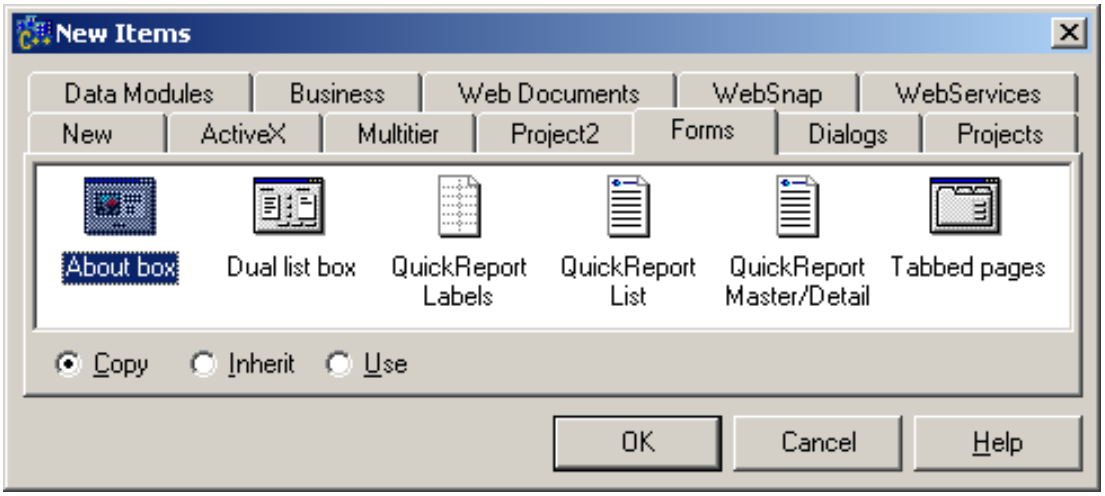
Вигляд форми під час роботи проекту:



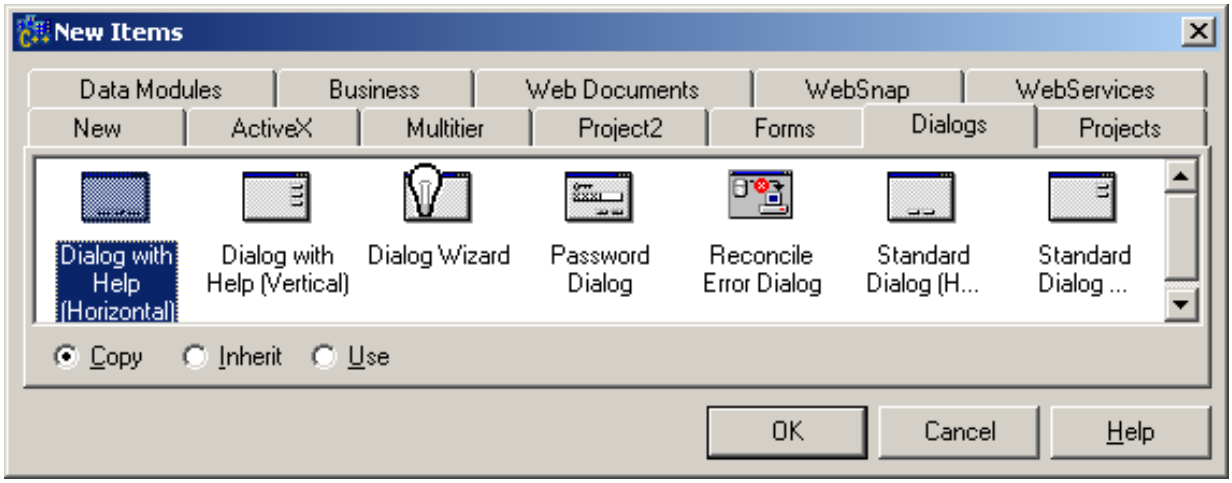
2.1.4 Створення проектів з декількома формами

За замовчуванням проект містить файли для однієї форми та вихідного коду одного модуля. Однак більшість більш-менш серйозних проектів містять декілька форм і модулів. Щоби створити новий модуль чи то форму у проекті, треба скористатись командою **File / New…** та обрати відповідний ярличок створюваного елемента проекту у вікні New Items (див. далі рис. 2.7).

Крім стандартної форми на закладці **Forms** цього вікна пропонується декілька різновидів створюваних форм:



Більше того, на закладці **Dialogs** існують зручні засоби створення форм-діалогів з готовим інтерфейсом:



Крім команди **File / New…** для створення нових форм чи модулів проекту можна скористатись відповідними “гарячими” кнопками або відкрити командою **View / Project Manager** менеджер проектів, в якому скористатись кнопкою **New**. Також вікно менеджера проектів надає можливість за потреби видалити елемент проекту (проект, форму, модуль тощо) за допомогою кнопки **Remove**.

Залучити заголовний файл нової форми чи модуля можна виконавши команду **File / Include Unit Hdr** чи то самостійно прописавши директиву `include`.

Коли у подальшій роботі виникає потреба відкрити файл долученого модуля, можна скористатись клавішами <Ctrl> + <Enter>, попередньо встановив-

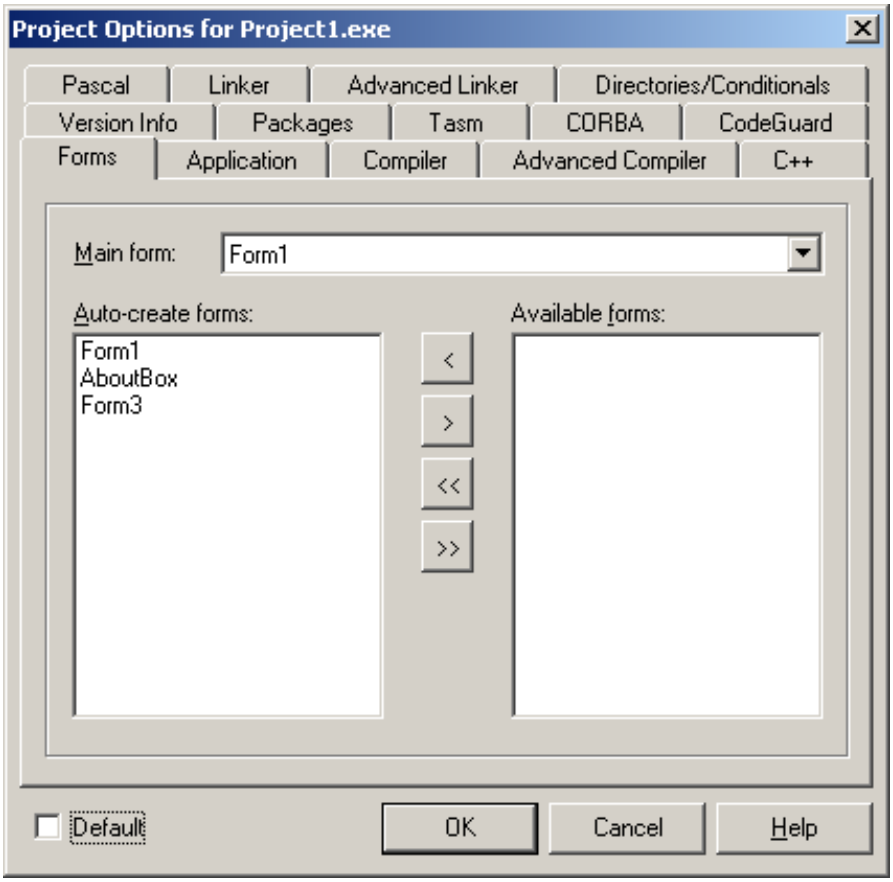
ши курсор на ім'я цього модуля у директиві `include`. Аналогічну дію можна виконати командою контекстного меню **Open File at Cursor**.

Нагадаємо, що комбінацією <Ctrl> + <F6> чи то командою контекстного меню редактора **Open Source/Header File** користуються для перемикання поміж cpp- та h-файлами, а для переходу поміж відповідною формою та її cpp-файлом можна скористатись клавішею <F12>.

При застосуванні клавіш <Shift> + <F12> відкривається вікно зі списком форм проекту для зручного перемикання поміж ними особливо при великій кількості форм у проекті.

Оскільки вікно редактора коду при створенні програмного проекту потребує на екрані значного місця, то воно, зазвичай, перекриває собою більш дрібні вікна IDE, приміром вікно інспектора об'єктів, вікно treeview, редактор форми чи то інші. Щоби побачити "приховані" вікна, можна скористатись поєднанням клавіш <Alt> + <0> для виведення списку відкритих вікон IDE.

Якщо виконати команду **Project / Options**, відкриється діалогове вікно різноманітних опцій проекту, в якому можна обрати головну форму проекту, визначити, які форми будуть створюватися динамічно, задавати параметри компіляції модулів і компонування тощо.



Приклад програмного проекту з трьома формами, у тому числі форми `AboutBox`, наведено у прикладі 9.3.

2.2 Робота у консолі

2.2.1 Основні функції роботи у консольному режимі

Консольний додаток – це програма, для якої пристроєм введення інформації є клавіатура, а пристроєм виведення – монітор у символьному режимі.

Консольний режим − це режим створювання програм користувача без використання графічного інтерфейсу, а з відображенням лише символьної інформації. В операційній системі робота у консольному режимі здійснюється засобами командного рядка.

Виконування консольного додатка відбувається у чорному консольному екрані. Для введення даних слід використовувати спеціальні команди. Під час виконування вони скомандують програмі зупинитися й очікувати, допоки користувач уведе значення і натисне клавішу <Enter>. Виведення даних виконується у теж само чорне вікно і завжди потребує вказівок та пояснень для користувача.

Виведення означає, що певний текст чи то значення певних змінних висвітяться на екрані і користувач зможе їх прочитати. Введення означає, що користувач задає значення змінної, яке буде використано у програмі.

Перед тим як розпочати створювання консольного програмного додатка, слід розглянути функції, які забезпечують введення даних з клавіатури та виведення результатів на екран.

Для виведення тексту у C++ найчастіш використовується команда `cout<<` з текстом, який записується у подвійних лапках, наприклад:

```
cout << "Текст ";
```

Для виведення значення змінної також використовується команда `cout<<` із зазначенням імені змінної без лапок. Наприклад, щоб вивести значення змінної `x`, слід написати:

```
cout << x;
```

Для виведення значення змінної x з коментарем слід написати:

```
cout << "Значення змінної x: " << x;
```

Після кожного виведення курсор залишається на тому ж самому рядку і наступне виведення виконується поряд з попереднім. Якщо треба перемістити курсор на новий рядок після виведення даних, слід написати `<<endl` наприкінці команди:

```
cout << " Значення змінної x: " << x << endl;
```

Для введення значення змінної зазвичай використовується команда `cin>>`:

```
cin >> x;
```

Після того як програма завершила свою роботу, чорне вікно закривається. Зазвичай це трапляється настільки швидко, що користувач не встигає побачити результати виконання програми. Щоб зупинити програму і надати користувачу можливість прочитати результати, використовується функція `getch()`, яка очікує на натиснення будь-якої клавіші, наприкінці програми перед `return 0`:

```
getch();
```

Аналогічну дію виконує функція `cin.get()`.

Зверніть увагу! Щоб у програмі можна було використовувати команди `cin` та `cout`, на початку програми слід написати:

```
#include <iostream.h>
```

Для використання функції `getch()` слід на початку програми написати:

```
#include <conio.h>
```

Наступна програма демонструє консольне виведення:

Текст програми:

```
1  //Програма використовує команду cout
2  #include <iostream.h>
3  int main()
4  {
5  cout<<"Привіт!\n";
6  cout<<"Це 5: " << 5 << "\n";
7  cout<<"Маніпулятор endl переводить курсор на новий рядок"<<endl;
8  cout<<"Велике число:\t" << 70000 << endl;
9  cout<<"Сума 8 + 5:\t" << 8+5 << endl;
10 cout<<"Дріб:\t\t" << (float) 5/8 << endl;
11 cout<<"Дуже велике число:\t" << (double) 7000*7000 <<endl;
12 cin.get();
13 return 0;
14 }
```

Результати виконання програми:

```
Привіт!
Це 5: 5
Маніпулятор endl переводить курсор на новий рядок
Велике число:        70000
Сума 8 + 5:          13
Дріб:                0.625
Дуже велике число:   4.9e+07
```

Розглянемо детальніше наведену програму.

У рядку 2: інструкція `#include <iostream.h>` долучає файл `iostream.h` до коду програми. Це потрібно для використання `cout` та пов'язаних з ним функцій.

Рядок 3 містить заголовок функції `main()`. Доволі часто у консолі функція `main()` не має аргументів, однак, якщо треба при викликанні програми передати їй якісь параметри, синтаксис функції матиме вигляд

```
int main(int argc, char* argv[])
```

Тут перший аргумент `argc` задає кількість передаваних параметрів, а другий, `argv`, є вказівником на масив символьних рядків, які містять ці аргументи. За замовчуванням у консольному додаткові C++ Builder функція `main()` має саме такий синтаксис. Для наведених у посібнику прикладів консольних програм синтаксис цієї функції є несуттєвим, тобто обидва синтаксиси є працездатними.

У рядку 5 показано найпростіший спосіб виведення рядка – `cout`. Символ `\n` є спеціальним форматним знаком. Він вказує `cout` перейти на новий рядок (аналогічно до `endl`).

У рядку 6 команді `cout` передаються три значення і кожне з них відокремлюється знаками <<. Перше значення – рядок `"Це 5: "`. Зверніть увагу на пробіл після двокрапки. Цей пробіл є часткою рядка. Далі йде значення 5 і символ переведення рядка (завжди у подвійних чи одинарних лапках). Внаслідок цього на екран виводиться:

```
Це 5: 5
```

У рядку 7 виводиться на екран інформаційне повідомлення (рядок), після чого використано маніпулятор `endl` для переведення курсора на новий рядок.

У рядку 8 використовується ще один форматний символ: `\t`. Він вставляє табуляцію (пропуск у рядку до певної позиції). Рядок 9 показує, що можна виводити не лише цілі, а й довгі цілі числа.

Рядок 9 ілюструє можливість виконання обчислення суми при виведенні за допомогою `cout`. Обчислюється результат суми `8+5`, і `cout` виводить на екран значення 13.

У рядку 10 виконується обчислення і виведення виразу `5/8`. Перед цим написано `(float)`, яке повідомляє `cout`, що йдеться про дійсне число з десятковою крапкою.

У рядку 11 обчислюється і виводиться результат виразу `7000*7000` і `(double)` повідомляє `cout`, що результат слід вивести в експоненціальному вигляді. У наведеній програмі `4.9e+07` означає число $4.9*10^7=49000000$.

Окрім специфічних команд C++: `cin` та `cout` (вони мають назву команд *потокового введення/виведення*), – існують команди для введення/виведення, які прийшли до C++ з C: `scanf` та `printf`. Ці команди мають назву *команд форматного введення/виведення*.

Для того щоб програма могла використовувати згадані команди, на початку програми слід долучити директиву `#include <stdio.h>`.

У загальному вигляді функція `scanf` для введення значення однієї змінної виглядає як

scanf(<*формат*>, &<*змінна*>);

де: *формат* − рядок специфікаторів формату у подвійних лапках. Найбільш поширеними специфікаторами є: `%i` − для введення цілих чисел, `%f` − для введення дійсних чисел, `%s` − для введення рядка символів;

змінна − це ім'я змінної, значення якої вводиться.

Знак & є елементом синтаксису і означає операцію отримання адреси змінної у пам'яті. Докладніше про адреси змінних див. у розд. 6.

Наприклад, функція `scanf("%i", &kol)` вводить значення цілої змінної `kol`, а функція `scanf("%i %f", &kol, &vart)` − значення цілої змінної `kol` і дійсної змінної `vart`.

При виконуванні функції `scanf()` відбувається таке. Програма призупиняє роботу і очікує, допоки користувач набере на клавіатурі рядок символів та

натисне клавішу <Enter>. Після натиснення клавіші <Enter> функція `scanf()` перетворює введений рядок відповідно до специфікаторів формату на дані й записує їх до змінних, адреси яких зазначено. Приміром, після виконання функції

```
scanf("%f", &cena);
```

і введення з клавіатури рядка 34.76 змінна `cena` набуде значення 34.76.

Для форматного *виведення* інформації в консольному режимі є функція `printf()`. Загальний вигляд функції є такий:

printf(<*формат*>, <*список_змінних*>);

Тут *формат* задає пояснювальний текст та вигляд значень змінних, імена яких задає параметр *список_змінних*. Параметр *список_змінних* не є обов'язковим і являє собою послідовність розділених комами змінних, значення яких виводяться. Наприклад, функція `printf("x=%7.3f", x)` виводить дійсне число `x` у заданому форматі з трьома цифрами дійсної частини числа після десяткової крапки.

Специфікатор формату задає вигляд виведеного результату. В табл. 2.7 наведено найбільш поширені специфікатори формату. Необов'язковий параметр специфікатора `n`, замість якого слід підставити десяткове число, задає розмір поля виведення; параметр `m` – розмір поля для виведення цифр дійсної частини.

Таблиця 2.7

Специфікатори формату

Специфікатор	Тип змінної	Формат виведення
`%nd`	`int`	десяткове зі знаком
`%n.mf`	`float` чи `double`	дійсне з фіксованою крапкою
`%ne`	`float` чи `double`	дійсне з рухомою крапкою
`%nc`	`char`	символ
`%ns`	`char*`	рядок

Існує ще одна пара функцій С для введення-виведення: `gets()` і `puts()`. Використовуються ці функції лише для рядків.

Функцію `puts()` часто використовують для виведення на екран повідомлень. Вона після виведення автоматично переводить курсор на початок наступного рядка. Функція `puts()` має лише один параметр – текст повідомлення. Наприклад, функція

```
puts("Введіть значення, \n змінної x");
```

виведе два рядки:

```
Введіть значення
змінної x
```

і переведе курсор на початок наступного рядка. Для обмеження рядків застосовується спеціальна послідовність символів.

Докладніше про введення/виведення рядків див. у розд. 7.

2.2.2 Послідовність створювання консольного програмного додатка

Щоби створити в C++ Builder консольний програмний додаток, слід виконати такі дії. Спочатку в меню **File** слід обрати команду **New…** і на багаторядковій панелі **New Items** діалогового вікна (див. рис. 2.7) клацнути по значкові Console Wizard.

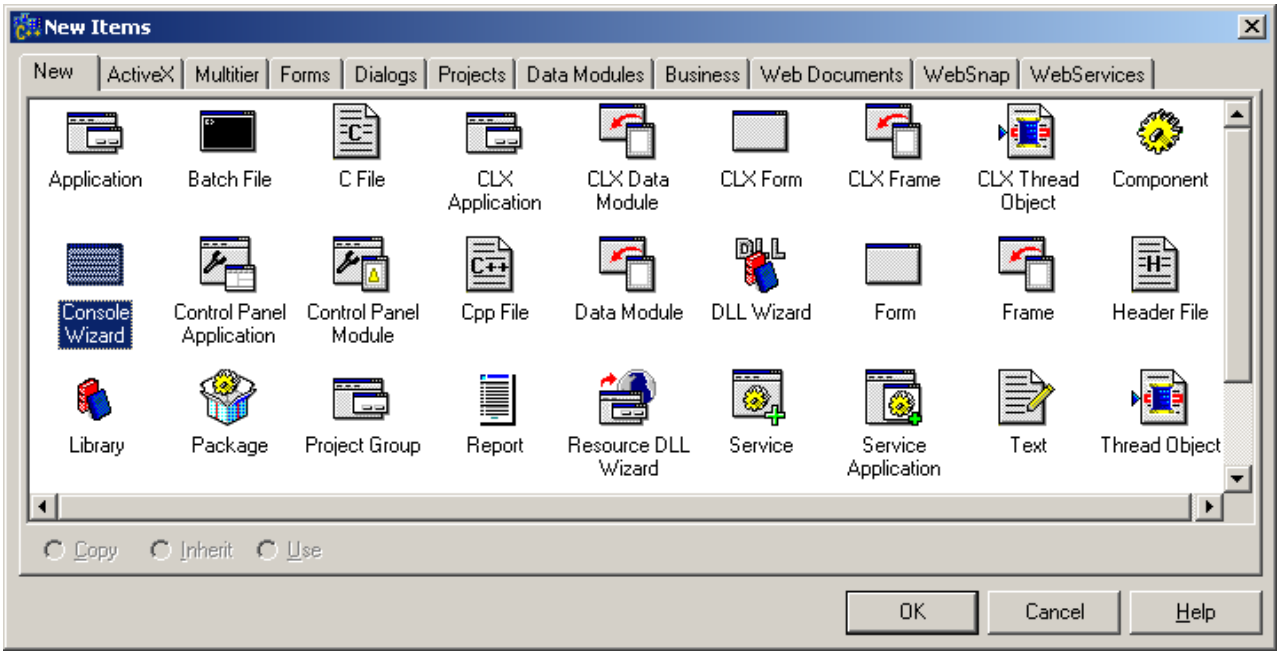


Рис. 2.7. Діалогове вікно New Items

Унаслідок виконаних дій на екрані з'явиться вікно Console Wizard (див. рис. 2.8). У цьому вікні можна обрати мову програмування і зазначити, чи буде використовуватись та чи інша бібліотека. Після того як буде задано параметри створюваного консольного додатка, слід клацнути по кнопці OK.

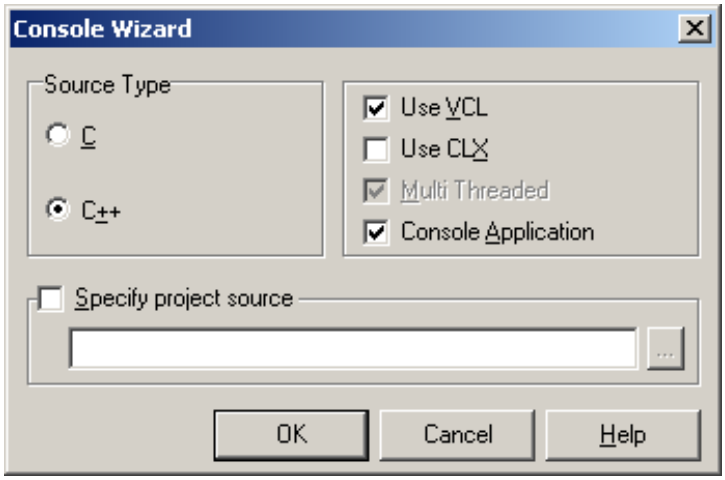


Рис. 2.8. Вікно Console Wizard

Як наслідок C++ Builder створить проект консольного додатка – і на екрані з'явиться вікно редактора коду Unit1.cpp (див. рис. 2.9), в якому буде створено шаблон консольного додатка чи то функція `main()`.

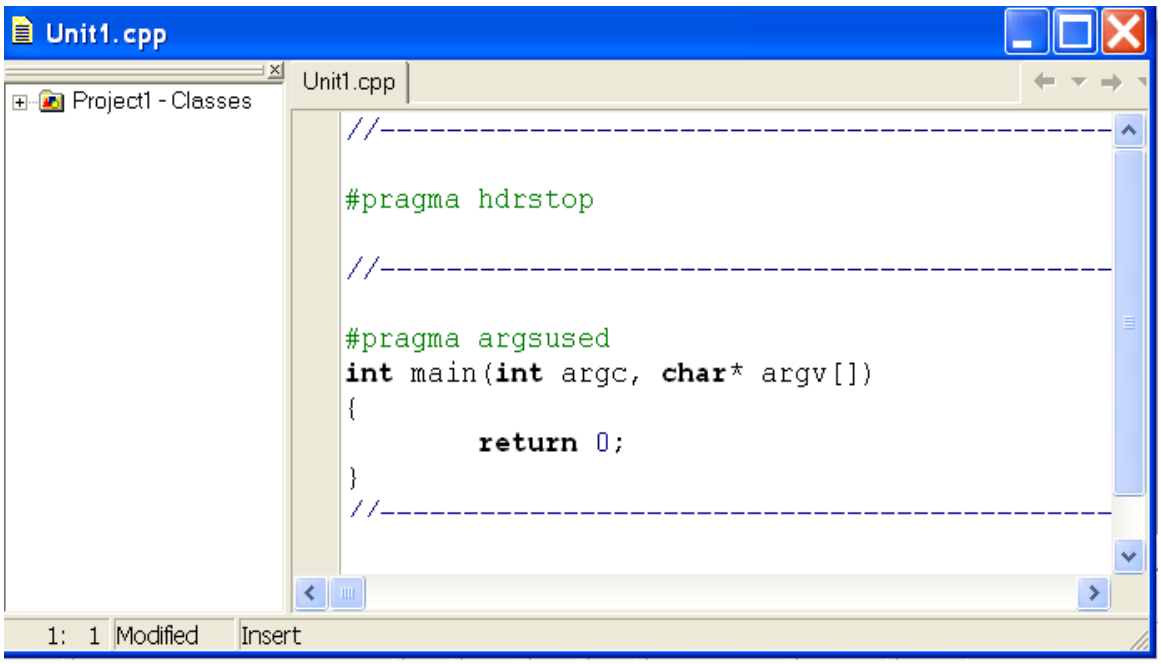


Рис. 2.9. Вікно редактора коду Unit1.cpp

Розпочинається консольний додаток директивою `#pragma hdrstop`, яка забороняє виконування попередньої компіляції долучених файлів.

Після цієї директиви можна вставити директиви `#include`, які забезпечують долучення потрібних заголовних файлів. Наприклад `#include <stdio.h>` долучає заголовний файл, який містить прототипи функцій введення/виведення, у тому числі `printf()`. Файл `conio.h` потрібен, оскільки для очікування натиснення клавіші було застосовано функцію введення символу `getch()`.

Директива `#pragma argsused` скасовує попередження компіляції про те, що аргументи, зазначені у заголовку функції, не використовуються.

Функція `main()` є присутня у кожній програмі, саме через неї передається керування після завантаження та ініціалізації програми.

Слід звернути увагу на те, що консольний додаток розробляється у Windows, а виконується як програма DOS. Оскільки в DOS та Windows літери кирилиці мають різні коди, це призводить до того, що консольний додаток замість повідомлень на кирилиці виводить “абракадабру”.

Для зберігання проекту слід обрати у головному меню **File/Save Project as…** і зберегти файли проекту в окремому каталозі (теці).

Для компіляції та запуску програми на виконання треба натиснути кнопку – **Run** на інструментальній панелі чи функціональну клавішу <F9>, після чого на екрані з’явиться прототип вікна MS DOS.

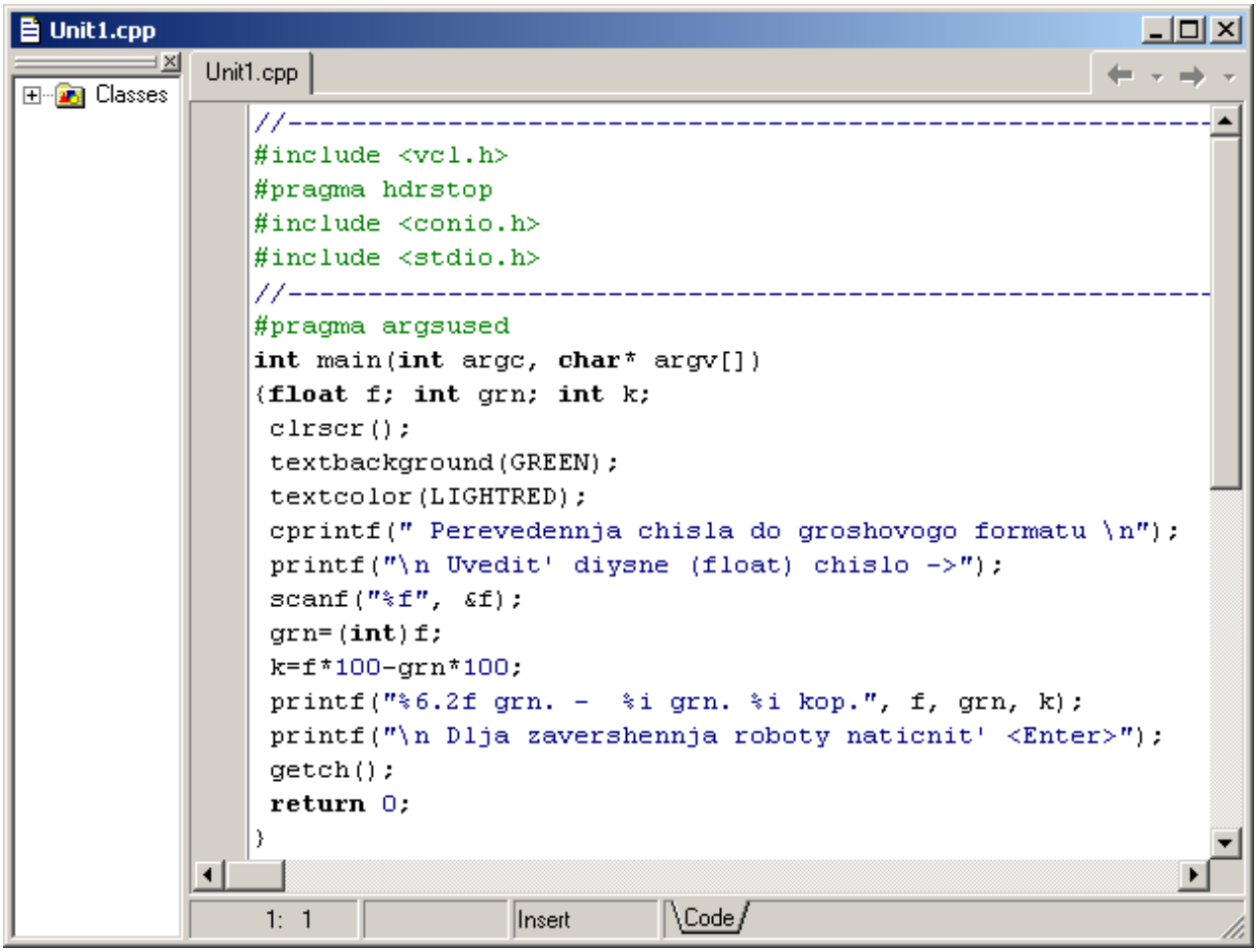
Якщо натиснути будь-яку клавішу на клавіатурі, – програма завершиться й її вікно закриється.

Приклад програми в консольному режимі

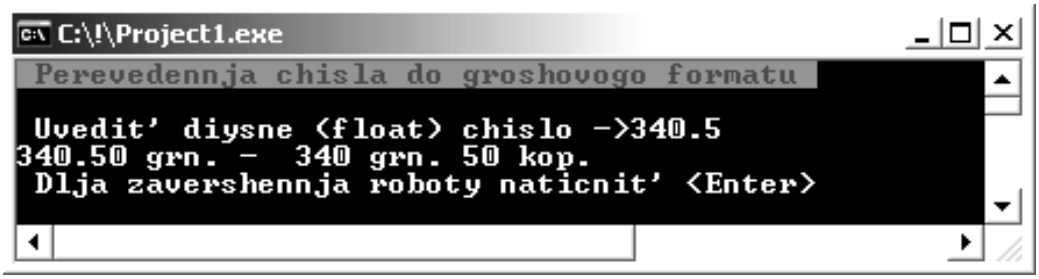
Приклад 2.4 Написати консольну програму для переведення певного введеного дійсного числа з клавіатури до грошового формату. Наприклад, число 23.5 має бути приведено до вигляду 23 грн. 50 коп.

Розв'язок. Для виведення повідомлень використовуватимемо функції `printf` та `cprintf`; для введення даних з клавіатури – функцію `scanf`, а для виведення результатів – функцію `printf`. Функція `cprintf`, на відміну від `printf`, дозволяє задавати колір літер попередньо заданою функцією `textcolor` і колір фону літер – функцією `textbackground`. Функція `clrscr();` очищує екран.

Вигляд вікна редактора коду з текстом програми:



Результати роботи програми у консольному додатку:



Питання та завдання для самоконтролю

1) Поясніть поняття RAD, GUI та IDE.

2) З яких двох етапів складається створення програм у C++ Builder?

3) Перелічіть елементи вікна середовища C++ Builder.

4) Для чого потрібна форма проекту?

5) Яке призначення властивостей, подій та методів компонентів?

6) Яке вікно C++ Builder призначене для записування текстів програм?

7) В який спосіб можна здійснювати перемикання поміж вікнами форми й редактора коду?

8) Для чого потрібна палітра компонентів?

9) Перелічіть компоненти, які Ви вже знаєте.

10) Для чого потрібний інспектор об'єктів?

11) В якому вікні задають властивості компонентів?

12) Назвіть основне призначення сторінки подій Events.

13) Перелічіть події, які Ви вже знаєте.

14) Назвіть основні властивості, притаманні більшості компонентів.

15) Вкажіть основні методи стандартних компонентів.

16) З яких основних файлів складається програмний проект у C++ Builder?

17) Яке ім'я за замовчуванням надається головному файлу програми?

18) Назвіть основне призначення заголовного файла та файла реалізації.

19) За допомогою якої директиви препроцесора до програми долучають бібліотечні модулі (заголовні файли)?

20) В яких бібліотеках містяться математичні функції?

21) Яке призначення мають фігурні та круглі дужки в C++?

22) Назвіть послідовність створювання програмного проекту у C++ Builder.

23) В який спосіб можна зберегти програмний проект?

24) В які способи можна запустити проект на виконання?

25) В які способи можна здійснити перехід поміж вікном форми і вікном редактора коду?

26) Що таке консольна програма?

27) Які функції введення-виведення даних у консольному режимі Вам відомі?

28) Які заголовні файли слід долучити для використання функцій введення-виведення?

29) Вкажіть призначення і можливий синтаксис функції `main()`.

30) Назвіть послідовність створювання консольного проекту в C++ Builder.

Розділ 3

Елементи мови C++

3.1 Історія C++

Мова програмування − формалізована мова, призначена для описування алгоритмів розв'язування задач на ЕОМ. Мова програмування потрібна, щоб записувати алгоритм у термінах, зрозумілих компіляторові, який, своєю чергою, перекладе його на зрозумілу процесорові машинну мову. Як і кожна людська мова, мова програмування складається з алфавіту (вживаних символів), слів (елементарних конструкцій, які називають лексемами), словосполучень (виразів) та речень (операторів). Лексеми утворюються із символів, вирази – із лексем та символів, а оператори – із символів, виразів та лексем.

C++ є результатом еволюції мови C, створеної Денісом Рітчі та Брайаном Керніганом. Ще на початку 70-х років минулого сторіччя C набула широкої популярності як мова розробляння операційної системи UNIX. Сьогодні більшість операційних систем написано мовою C та/чи C++. Мова C є доступною для більшості систем, до того ж вона є машинонезалежною. За ретельного проектування можна писати C-програми, які можна переносити на більшість комп'ютерів.

Широке використання C на комп'ютерах різних апаратних платформ призвело до появи різноманітних варіацій мови. Це зумовило потребу стандартизації мови і 1990 року відповідний документ стандарту ANSI/ISO 9899: 1990 було розроблено.

Мову C++ розробив Бьєрн Страуструп на початку 80-х років минулого сторіччя. Порівняно з C, мова C++ набула багато "прикрас", головною з яких є можливість об'єктно-орієнтованого програмування. Об'єкти є за своєю суттю програмними компонентами, які моделюють предмети реального світу. Модульний, об'єктно-орієнтований підхід до проектування та реалізації програмного забезпечення дозволили полегшити написання складних потужних програмних проектів. Об'єктно-орієнтовані програми є набагато простіші для розуміння, виправлення та модифікації.

1991 року на основі C і C++ фірмою Sun, а саме Джеймсом Гослінгом, було створено мову Java. Метою Java на початку було керування інтелектуальною побутовою технікою, а згодом, завдячуючи шаленому вибухові популярності Word Wide Web, її швидко було перекваліфіковано і представлено 1995 року як потужний засіб розробляння програм для розв'язування різноманітних сучасних задач. Сьогодні Java використовується для створювання динамічного й інтерактивного вмісту Web-сторінок, розширювання можливостей Web-серверів, створювання програм для мобільних телефонів та побутових пристроїв, розробляння широкомасштабних програмних додатків на підприємствах тощо. Подібні можливості мають сучасні версії C++, як-от Microsoft® Visual C++®.NET та Borland® C++ Builder™. На основі C++ і Java для платформи .NET

було розроблено мову C#, яка, подібно до Microsoft® Visual C++®.NET та Visual Basic.NET, дозволяє інтегрувати в програмні додатки можливості Internet та Web, писати компоненти програмного забезпечення мовою, яку знає розроблювач, а потім будувати додатки, комбінуючи ці компоненти з компонентами, написаними будь-якою іншою .NET мовою.

Еволюція C++ Builder розпочинається з 1997 року, і майже кожного року випускаються оновлені й оптимізовані версії. Приміром, 2002 року вийшла версія C++ Builder Builder Enterprise 6.0 for Windows NT\2000\XP + Update, як потужний і якісний інструмент реалізації RAD-стилю (Rapid Application Development – швидке розробляння додатків) стосовно самих різних класів задач, включаючи роботу з Internet, ActiveX, з проектами рівня підприємств, і все це – з використанням стандартного C++. Borland C++ Builder тісно інтегрований з обома базовими технологіями створення розподілених систем – CORBA та COM. Крім того, Borland C++ Builder є сумісний з Visual C++ та підтримує більшість технологій сучасної розробки. 2005 року Borland випустила Borland Developer Studio 2006, який включає у себе Borland C++ Builder 2006 з оптимізованим керуванням конфігурацією та налагодження. Borland Developer Studio 2006 є повноцінним комплектом, який містить Delphi, C++Builder та C#Builder. 2007 року CodeGear випустила C++ Builder 2007, в якому реалізувала повну підтримку API Microsoft Windows Vista, збільшила повноту відповідності стандартові ANSI C++, прискорила розробку до 5-ти разів, долучила підтримку MSBuild, архітектур баз даних DBX4 та “VCL для Web”, яка підтримує AJAX. Підтримка API Microsoft Windows Vista включила додатки, оформлені у стилі Vista, і підтримку VCL для Aero та Vista Desktop. CodeGear RAD Studio 2007 містить C++ Builder 2007 та Delphi. Також 2007 року CodeGear відродила марку “Turbo” і випустила дві “Turbo” версії C++ Builder: Turbo C++ Professional та Turbo C++ Explorer (безкоштовна), основаних на Borland C++ Builder 2006. Наприкінці 2008 року компанія CodeGear випустила нову версію RAD Studio, до якої увійшли Delphi 2009 та C++ Builder 2009. У складі RAD Studio 2009 року вийшов C++ Builder 2010, а 2010 року вийшла RAD Studio XE, яка дозволяє швидко і наочно розробляти додатки з графічним інтерфейсом для Windows, .NET, PHP та веб-додатків. У C++ Builder XE можна створювати динамічні додатки для будь-яких задач: від активної взаємодії з базами даних до проектування додатків з графічним інтерфейсом для керованих СУБД багатоланкових систем, настільних систем, сенсорних екранів, веб-додатків, веб-служб та багато ін.

Зауважимо, що C++ Builder спочатку створювалася тільки для платформи Microsoft Windows. Пізні версії, які містять кросплатформову компонентну бібліотеку Borland, підтримують і Windows і Linux.

Отже, C++ – потужна, лаконічна, гнучка, мобільна і жива мова, яка розвивається і в різних своїх версіях набуває нових сучасних можливостей.

3.2 Алфавіт мови С++

Алфавіт мови – набір символів, які можна використовувати для записування програмного коду: це великі та малі літери латиниці, цифри, знаки операцій та спеціальні символи. Кожен зокрема чи в комбінаціях *знаки операцій* і *спеціальні символи* дозволяють задавати лексеми, вирази і оператори, серед яких найбільш використовувані є такі:

= – присвоювання. Наприклад, якщо `x` і `y` – змінні, то запис `x = y` означає, що змінна `x` набуває значення `y` (докладніше див. п. 3.5.3);

+, **–**, *****, **/** – знаки операцій додавання, віднімання, множення та ділення;

++, **––** – знаки операцій збільшення та зменшення на одиницю значення операнда;

; – символ завершення команди;

< (менше), **<=** (менше чи дорівнює), **>** (більше), **>=** (більше чи дорівнює), **==** (дорівнює), **!=** (не дорівнює) – знаки операцій порівняння (див. п. 4.3.3);

+=, **–=**, ***=**, **/=**, **%=**, **<<=**, **>>=**, **?=**, **^=**, **|=** – знаки для записування операцій присвоювання (див. п. 3.5.3);

% – залишок від цілочисельного ділення двох чисел. Наприклад у виразі `c = a % b` значення залишку від ділення `a` на `b` присвоюється змінній `c`;

&&, **||**, **!** – знаки логічних операцій, які докладно описано в п. 4.3.3;

&, **|**, **^**, **<<**, **>>**, **~** – знаки побітових логічних операцій (див. п. 4.3.3);

?: – знаки умовної операції (див. п. 4.3.7);

*****, **&** – знаки адресних операцій. Хоча такі символи вже зустрічались у цьому переліку, однак їхнє специфічне застосовування у відповідних виразах дозволяє виключити усіляку неоднозначність (див. розд. 6);

/*, ***/**, **//** – сукупність знаків, використовуваних для записування коментарів;

, – знак коми, використовуваний для перелічування об'єктів;

. , **–>** – знаки використовувані при роботі зі структурованими типами і класами;

{ } – операторні дужки, які поєднують *блок* команд;

[ ] – дужки для запису індексів масивів;

( ) – дужки для встановлення черговості виконування операцій і записування аргументів функції;

' ' – знаки для записування значення символьних констант;

" – знак для записування значення символьної інформації .

Список усіх операцій у порядку спадання їхніх пріоритетів наведено у додатку Б.

Коментарі застосовуються лише для пояснень і жодних дій у програмі не спричинюють, тому що ігноруються компілятором. Текст коментарів розміщують поміж знаків `/*` та `*/`. Коментар може бути будь-якої довжини, чи то займати частину рядка, чи декілька рядків. Наприклад:

```
/* Коментар до програми
може займати кілька рядків   */
```

Окрім символів `/*` та `*/` для невеликих коментарів в один рядок в С++ використовують знак `//`.

// Текст після цього символу і до кінця рядка є коментарем.

Також, окрім пояснень, доволі часто коментарі використовують при налагодженні для тимчасового "вимкнення" певного фрагмента програми.

Змінна – іменована ділянка оперативної пам'яті, застосовувана для зберігання даних під час роботи програми. При оголошенні змінної для неї резервується певна ділянка пам'яті, розмір якої залежить від конкретного типу змінної. *Значення змінної* – фактичне значення, яке міститься у цій ділянці пам'яті.

Ім'я (*ідентифікатор*) *змінної* може складатися з літер латиниці, цифр та символу "_" (підкреслювання), але неодмінно має розпочинатися з літери чи символу підкреслювання. Великі й малі літери розрізнюються, тобто мова С++ є чутливою до регістру, наприклад – *a* та *A* – це два різних об'єкти. Ідентифікатор створюється на етапі оголошення змінної, функції, структури тощо. Після цього його можна використовувати в командах розроблюваної програми.

Слід відзначити важливі *особливості при обиранні ідентифікатора*.

По-перше, ідентифікатор не повинен збігатися з ключовими словами, із зарезервованими словами й іменами функцій бібліотек компілятора мови С++.

По-друге, слід звернути особливу увагу на використання символу підкреслювання "_" у якості першого символу ідентифікатора, оскільки побудовані у такий спосіб ідентифікатори, з одного боку, можуть збігатися з іменами системних функцій та змінних, а з іншого, – при використанні таких ідентифікаторів програми можуть стати непереносними, тобто їх не можна буде використовувати на комп'ютерах інших типів.

3.3 Типи даних

У програмі мовою С++ усі змінні мають бути оголошеними, тобто для кожної змінної має бути зазначено її тип. На відміну від інших мов, *у С++ задавати тип змінної можна в будь-якому місці програми до її використання*. При оголошенні змінної для неї резервується ділянка пам'яті, розмір якої залежить від типу змінної.

Тип змінної – вид і розмір даних, які змінна може зберігати. Кожен тип даних зберігається й опрацьовується за певними правилами. Слід зауважити, що розмір одного й того ж типу даних може відрізнятися на комп'ютерах різних платформ, а також залежить від налагоджень компілятора.

Усі типи мови С++ розподіляють на дві групи: основні типи та структуровані. Список основних типів даних С++ Builder із зазначенням діапазону та прикладами можливих значень змінних наведено в табл. 3.1.

До *основних (базових)* типів можна віднести `char`, `int`, `float` та `double`, а також їхні варіанти з специфікаторами `short` (короткий), `long` (довгий), `signed` (зі знаком) та `unsigned` (без знаку).

Структуровані (похідні) типи базуються на основних, до них належать масиви будь-яких типів, вказівники, функції, класи, файли, структури, об'єднання, перерахування тощо.

Таблиця 3.1

Основні типи даних С++

Тип	Назва	Розмір, байт	Діапазон	Приклади можливих значень	Типи чисел
`char`	символьний (знаковий)	1	–128…127	'a', '\n', '9'	цілі
`unsigned char`	беззнаковий символьний	1	0…255	1, 233	
`short`	короткий цілий	2	–32 768…32 767	1, 153, –349	
`unsigned short`	беззнаковий короткий	2	0…65 535	0, 4, 65 000	
`int`	цілий (знаковий)	4*	–2 147 483 648… …2 147 483 647	–30 000, 0, 690	
`unsigned int`	беззнаковий цілий	4	0…4 294 967 295	2 348, 60 864	
`long`	цілий (знаковий)	4	–2 147 483 648… …2 147 483 647	–30 000, 0, 690	
`float`	дійсний одинарної точності	4	$3.4\cdot10^{-38}$…$3.4\cdot10^{38}$	3.23, –0.2 100.23, 12, –0.947, 0.0001, 6.34e–3, 4e5	дійсні
`double`	дійсний подвійної точності	8	$1.7\cdot10^{-308}$… …$1.7\cdot10^{308}$		
`long double`	довгий дійсний	10	$3.4\cdot10^{-4932}$… …$1.1\cdot10^{4932}$		
`bool`	логічний	1	false чи true	false(0), true(>=1)	
`enum`	перераховний	2 чи 4			
`void`	порожній, без значення				

* – залежно від налагоджень компілятора та апаратних характеристик тип `int` може мати 4 чи 2 байти.

Для подання *цілих чисел* використовують типи `char`, `short`, `int`, `long`. Специфікатор `unsigned` застосовують при роботі з додатними числами (без знаку), а специфікатор `signed` – для яких завгодно чисел, як додатних, так і від'ємних. За замовчуванням призначається знаковий тип, а тому специфікатор `signed` зазначати необов'язково.

Типи `float` і `double` визначають *дійсні змінні* розміром у 32 і 64 біти відповідно, а `long double` – 80 бітів. В С++ для відокремлення цілої частини числа від дійсної застосовується десяткова крапка. Окрім звичної форми, дійсні

константи можна записувати у формі з рухомою крапкою. Наприклад: 2.38e3 (яке дорівнює $2.38 \cdot 10^3 = 2380$), 3.61e–4 (яке дорівнює $3.61 \cdot 10^{-4} = 0.000361$). Число перед символом "e" називається *мантисою*, а після символу "e" – *порядком*, тобто замість основи 10 використовується літера "e", після якої ставиться показник степеня, наприклад: 1e3 ($1 \cdot 10^3 = 1000$), –2.7e–4 ($-2.7 \cdot 10^{-4} = 0.00027$). Змінні типу `float` займають в пам'яті 4 байти: 1 біт – знак, 8 бітів – порядок, 23 біти – мантиса.

Отже, на базових типах (`char`, `int`, `float`, `double`) будується решта типів даних за допомогою специфікаторів:

`signed` (знаковий – за замовчуванням), `unsigned` (беззнаковий) – до цілих типів `char` і `int`;

`long` (довгий), `short` (короткий) – до типів `double` і `int`.

Зауважимо, що ключові слова `signed` і `unsigned` є необов'язковими. Вони визначають, як інтерпретується старший біт оголошуваної змінної, тобто, якщо визначено тип `signed`, то нульовий біт числа виділяється під знак (0 – число додаткове, 1 – від'ємне). При визначенні типу `unsigned` усі виділені біти разом зі старшим використовується для зберігання значення числа. Якщо специфікатор типу складається з ключового типа `signed` чи `unsigned` і далі одразу слідує ідентифікатор змінної, то вона розглядатиметься як змінна типу `int`.

Наприклад:

```
unsigned int n;
int c;          // Інтерпретується як  signed int c
short  a;       // Інтерпретується як  signed short int a
unsigned d;     // Інтерпретується як  unsigned int d
signed f;       // Інтерпретується як  signed int f
```

При оголошенні змінних їм можна присвоювати початкові значення, які надалі може бути змінено. У прикладі

```
int i, j = 1, q = 0xFFF;
```

ціла змінна `i` не є ініціалізована, `j` – ініціалізована значенням 1, `q` – ініціалізована шістнацятковим значенням 0xFFF (десяткове 4095).

При ініціалізації змінних їм можна присвоювати арифметичні вирази:

```
long megabute=1024*1024;
```

Наведемо ще кілька прикладів оголошування змінних:

```
char tol='a';    // Символьна змінна tol ініціалізується символом 'a'
char x,c='\n';   // x не є ініціалізований, c – ініціалізований символом <Enter>
AnsiString s="літо";     // Рядок s ініціалізований значенням "літо" *
char st[5]="Одеса";      // Масив з 5-ти символів зі значенням "Одеса"
```

Якщо при оголошенні числові змінні не ініціалізовано, то їхні значення є невизначені (випадкові).

Зауважимо, що тип `char` використовується для подання символу, а для визначення його десяткового коду – тип `unsigned char` чи `signed char`. Значенням змінної типу `char` є відповідний код символу (розміром 1 байт). Для визначення десяткових кодів символів кирилиці специфікатор типу ідентифікато-

ра даних має бути типом `unsigned char`. Окрім того, тип `char` застосовується для оголошення рядків типу `char*` (див. розд. 7).

Змінна *логічного* типу `bool` займає лише 1 байт і може набувати лише одне із двох значень: `true` (істина) чи `false` (хибність). Зазвичай змінні типу `bool` застосовують чи то як перемикачі чи для побудови логічних виразів в умовах. Слід зауважити, що в С++ змінним логічного типу можна присвоювати і цілочисельні значення; при цьому значення 0 розцінюється як `false`, а будь-яке ненульове значення – як `true`.

Перерахування `enum` (від enumerate) призначено для обирання й призначання об'єктів із заданої множини значень. Оголошення перерахувань задає відповідний тип змінної і визначає список логічно пов'язаних іменованих констант, значеннями яких є певні цілі числа. Змінна перераховного типу може набувати значення однієї з іменованих констант списку. Кожен ідентифікатор у списку іменує елемент з переліку і має бути унікальним. Максимальна кількість ідентифікаторів – 65 535. Ідентифікатори списку є еквівалентні до їхніх значень, які можуть бути і додатними і від'ємними. У разі відсутності конкретного значення, першому ідентифікаторові відповідає значення 0, наступному ідентифікаторові – значення 1 і т. д., наприклад:

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

В цьому прикладі оголошено перерахування `days` з відповідною множиною значень, а також змінну `anyday` типу `days`. Імена `sun`, `mon`, `tues`, `wed`, `thur`, `fri`, `sat` є цілими константами, перша з яких `sun` автоматично встановлюється в 0, а кожна наступна має значення на одиницю більше за попередню (`mon` = 1, `tues` = 2, …, `sat` = 6).

Властивості даних типу `enum` є аналогічними до властивостей даних типу `int`, а змінні типу `enum` можуть застосовуватись в індексних виразах, в арифметичних операціях і в операціях порівняння у якості операндів. Більш докладно цей тип розглянуто в розд. 11.

Тип `void` (порожній, без значення) використовується, по-перше, для визначення типу функцій, які не повертають жодного результату, а всі дії та обчислення виконуються всередині цих функцій. По-друге, цей тип застосовують для зазначення порожнього списку аргументів функції (докладно функції розглядаються в розд. 8). По-третє, `void` використовують як базовий тип для вказівників. Окрім того, його застосовують в операціях зведення типів.

Для *визначення розміру пам'яті*, займаної змінною, існує операція `sizeof()`, яка повертає значення довжини зазначеного типу, наприклад:

```
a = sizeof(int);            // a = 4
b = sizeof(long double);    // b = 10
```

Окрім основних типів С++, середовище С++ Builder додатково підтримує власні різновиди типів, подібні до типів середовища Borland Delphi. Деякі з цих додаткових типів повторюють за змістом існуючі типи (табл. 3.2).

Таблиця 3.2

Додаткові типи даних C++ Builder

Тип C++ Builder	Діапазон значень, пояснення	Розмір, байт	Аналог C++	Різновиди
`Byte, byte`	0 … 255	1	`unsigned char`	цілочисельні
`Word`	0 … 65 000	2	`unsigned short`	
`Cardinal`	0 … 4 294 967 295	4	`unsigned int`	
`Comp`	$-2^{63}+1 \ldots 2^{62}-1$, тобто ~ $(-9.2\cdot10^{18} \ldots 9.2\cdot10^{18})$	8	`Comp`	
`ByteBool`	true/false, 0 ... 255	1	`unsigned char`	
`WordBool`	true/false, 0 ... 65 000	2	`unsigned short`	
`LongBool`	true/false, 0 … 4 294 967 295	4	`unsigned int`	
`Single`	$\pm1.5\cdot10^{-45} \ldots \pm3.4\cdot10^{38}$	4	`Float`	дійсні
`Double`	$\pm5.0\cdot10^{-324} \ldots \pm1.7\cdot10^{308}$	8	`Double`	
`Extended`	$\pm3.6\cdot10^{-4932} \ldots \pm1.1\cdot10^{4392}$	10	`long double`	
`Pointer`	вказівник	4	`void *`	
`AnsiChar`	один символ ANSI	1	`Char`	символьні
`Char`	еквівалентний до `AnsiChar`	1	`Char`	
`WideChar`	один символ Unicode	2	`wchar_t`	
`PChar`	вказівник на рядок	4	`unsigned char *`	рядки
`PAnsiChar`	вказівник на ANSI рядок	4	`unsigned char *`	
`AnsiString`	рядок символів ANSI	до 2 Гб		
`String`	еквівалентний `AnsiString`	до 2 Гб		
`WideString`	рядок символів Unicode	до 1 Гб		
`String[n]`	короткий рядок (n=1…255)	n + 1		
`SmallString<n>`	еквівалентний до `String[n]`	n + 1		
`ShortString`	еквівалентний до `SmallString<255>`	256		
`OleVariant`	OLE variant value	16	`OleVariant`	

Оголошення типів. Окрім оголошень змінних різних типів, в C++ є можливість оголошувати власні типи. Це можна здійснювати у два способи. Перший спосіб – використовувати для оголошення типу ключове слово `typedef`. Другий спосіб (буде розглянуто докладніше в розд. 11) – зазначити ім'я тега при оголошенні структури, об'єднання чи перерахування, а потім застосовувати це ім'я при оголошенні змінних та функцій у якості посилання на цей тег.

При оголошенні ключовим словом `typedef` створюється новий тип даних (див. підрозд. 11.1), який надалі може бути використаний у межах видимості цього типу для оголошення змінних, наприклад:

```
typedef int vector[40];
vector x;                    // Змінна x – масив 40-ка цілих чисел
```

Що є еквівалентним до оголошення: `int x [40];`

Використовуючи ключове слово `typedef`, можна оголошувати які завгодно типи, включаючи вказівники, функції чи масиви.

3.4 Константи

Константа – величина, яка не змінюється упродовж виконування програми. Для оголошення константи у програмі використовується специфікатор `const`. При цьому зазначається тип константи, наприклад:

```
const float  Pi = 3.14159;
```

У якості значення константи можна подавати константний вираз, який містить раніш оголошені константи та змінні. Наприклад:

```
const float  Pi2 = 2 * Pi,  k = Pi / 180;
```

Константам при створенні слід неодмінно надавати значення, які в подальшому не можна буде змінювати.

У мові С++ розрізнюють чотири типи констант: цілі, дійсні, символьні константи та рядки.

Для *цілих констант* тип можна не зазначати, оскільки за замовчуванням константам призначається тип `int`, наприклад:

```
const B = 286;       // інтерпретується як const int B=286
```

Окрім десяткових цілих констант, в С++ можна використовувати вісімкові (перед ними обов'язково ставиться 0) і шістнадцяткові (перед ними слід ставити 0x) цілі константи, наприклад:

```
const C=01, Q=0555, W=0777;       // вісімкові константи
const k=0xFF, K=0x1A, v=0x23      // шістнадцяткові константи
```

Значення вісімкових констант утворюються з десяткових цифр без вісімок та дев'яток, оскільки ці цифри не входять до вісімкової системи числення. А шістнадцяткові константи утворюються з набору цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (див. розд 1.4).

Приклади значень цілих констант:

Десяткові константи	Вісімкові константи	Шістнадцяткові константи
16	020	0x10
127	0117	0x2B
240	0360	0xF0

За потреби оголошення від'ємних цілих констант використовують знак "–" перед записом константи, який ще називають унарним мінусом, наприклад: –0x2A, –067, –16.

Для *дійсних констант* слід зазначити тип (`float`, `double` чи `long double`) і записати їхні значення з десятковою крапкою. Окрім звичної форми, дійсні константи можна записувати і в експоненціальній формі з рухомою крапкою, де десяткове дійсне число `N` записують у вигляді мантиси `m` і порядку `p`: $N = m \cdot 10^p$. Наприклад, число 373 можна подати як $0.373 \cdot 10^3$, при цьому мантиса `m = 3`, а порядок `p = 3`. Число 0.0002 можна записати як $0.2 \cdot 10^{-3}$ чи $2 \cdot 10^{-4}$. Запис у С++ цих чисел матиме вигляд 0.373e3 та 2e–4, де замість основи 10 використовується літера "e", після якої ставиться показник степеня. Число у формі

з рухомою крапкою може бути представлене безліччю способів: $2 = 20 \cdot 10^{-1} = 0.2 \cdot 10^{1} = 0.002 \cdot 10^{3}$ і т. д. Для однозначного подання на мантису накладається обмеження: $0 \le |m| < 1$, при дотриманні якого число знаходиться в *нормалізованій формі*.

Приклади оголошення дійсних констант:

```
const float fi = -1.84;
const double eps = 1e-4; // еквівалентно до eps =0.0001
```

Символьна константа – один символ в одинарних лапках ‘’, наприклад:

```
const char d='+', a='ц', c='9';
```

Рядкова константа (літерал) – послідовність символів, включаючи великі та малі літери кирилиці й латиниці, а також цифри і пробіли, розташовані поміж подвійних лапок “ ”, наприклад:

```
const AnsiString S="Одеса – місто-герой!"; //рядок типу AnsiString
const char *SS="ОНАЗ ім. О. С. Попова";     //рядок типу char*
```

Окрім літер і цифр, в символьних та рядкових константах можна задавати і керувальні символи: `'\n'` (Enter), `'\0'` (нуль-символ), `'\v'` – вертикальна табуляція тощо. Значенням символьної константи є числовий код символу.

Зауважимо, що керувальні символи можуть входити до складу рядків, наприклад при виведенні літерала “ОНАЗ \n ім. О. С. Попова” його частину “ОНАЗ” буде виведено в один рядок, а друга частина “ім. О. С. Попова” – в наступний рядок.

Якщо ключове слово `const` стоїть перед оголошенням структурованих типів (масив, структура, перерахування), це призводить до того, що кожен елемент також є константою, тобто значення йому може бути присвоєне лише один раз.

3.5 Правила записування арифметичних виразів

3.5.1 Операнди і вирази

Комбінація знаків операцій та операндів, результатом якої є конкретне значення, називається *виразом*. Знаки операцій визначають дії, які має бути виконано над операндами. Значення виразу залежить від розташування знаків операцій і круглих дужок у виразі, а також від пріоритету виконання операцій.

Операнд – це константа, літерал, ідентифікатор, виклик функції, індексний вираз, вираз вибору елемента чи більш складний вираз, сформований комбінацією операндів, знаків операцій і круглих дужок. Кожен операнд має тип.

Арифметичний вираз складається із операндів, арифметичних операцій (+ додавання, – віднімання, * множення , / ділення, % остача від ділення цілих чисел) і оператора присвоювання (=). Вираз, який завершується крапкою з комою, є *оператором*. Оператор задає закінчений опис певної дії. Виконання оператора виразу полягає в обчисленні виразу. Окремим випадком виразу є *порожній оператор* (;), який використовується, коли за синтаксисом оператор потрібен, а за змістом – ні (див. п. 4.4.2).

3.5.2 Арифметичні операції

Для записування виразів використовують відповідні операції. В С++ існують унарні, бінарні та тернарні операції, залежно від кількості операндів. Більшість операцій є бінарними, тобто мають два операнди, один з яких розміщується перед знаком операції, а другий – після. Наприклад, операція додавання "+" має два операнди: `x` та `y` і обчислює їхню суму. Унарні операції є ті, які мають лише один операнд. Наприклад, вираз `x++` означає застосування до операнда `x` операції унарного інкремента (збільшення на 1). Тернарну умовну операцію (`?:`), яка має три операнди, буде розглянуто докладно у п. 4.3.7.

Вирази можуть бути арифметичними, символьними та логічними.

Арифметичні вирази − це сукупність числових констант, змінних і функцій, пов'язаних знаками арифметичних операцій і круглих дужок. Значенням арифметичних виразів є число. С++ підтримує арифметичні операції, наведені в табл. 3.3.

Таблиця 3.3

Арифметичні операції

Позначення	Операція	Типи операндів і результату	Приклади
+	додавання	арифметичний, вказівник	`x + y`
−	віднімання та унарний мінус	арифметичний, вказівник	`x − y`
*	добуток	арифметичний	`x * y`
/	ділення	арифметичний	`x / y`
%	остача від ділення цілих чисел	цілий	`i % 6`
++	збільшення на одиницю (інкремент)	арифметичний, вказівник	`i++; ++i`
−−	зменшення на одиницю (декремент)	арифметичний, вказівник	`i--; --i`

Результатом операції обчислення остачі від ділення (%) є залишок від ділення першого операнда на другий. Операндами цієї операції мають бути цілі числа. Знак результату залежить від конкретної реалізації, переважно знак результату збігається зі знаком діленого. Якщо другий операнд дорівнює нулю, видається відповідне повідомлення.

```
int n = 49, m = 10, i, j, k, l;
i = n % m;          // i = 9
j = n % (–m);       // j = 9
k = (–n) % m;       // k = –9
l = (–n) % (–m);    // l = –9
```

Операції ++ (інкремент) та −− (декремент) є унарними, тобто мають лише один операнд. Операція ++ додає одиницю до операнда, операція −− віднімає одиницю від операнда. Ці операції можуть бути записані як праворуч, так і ліворуч операнда. Залежно від місця розміщення операції відносно операнда розрізняють дві форми цих операцій: префіксну та постфіксну. У префіксній формі, в якій операцію розміщують перед операндом, наприклад: `++i`, `--j`, спочат-

ку збільшується чи зменшується на одиницю значення змінної, а вже потім ця змінна з її новим значенням бере участь в арифметичному виразі. У постфіксній формі цих операцій, навпаки, операцію розміщують після операнда, наприклад: `i++`, `j--`, і у виразі спочатку використовується поточне значення цієї змінної, а потім збільшується чи зменшується її значення.

Три нижченаведені оператори дають одинакові результати, але мають різницю при використанні у виразах:

```
i = i + 1;  i += 1;  ++i;  i++;
```

Наприклад, в результаті виконання операторів

```
int  j, i = 1;
j = i++ * i++;
```

значення змінної `i` дорівнюватиме 3, а змінної `j` – 1.

Якщо змінювати ці оператори в такий спосіб:

```
int  j, i = 1;
j =++ i * ++i;
```

то результат буде іншим: значення `i` дорівнюватиме 3, а `j` дорівнюватиме 9.

Наведемо ще два приклади використання операції інкременту:

```
int k = 1, m, n;
n = k++; //порядок обчислень: 1) n=1, 2) k=k+1=2. Результат n=1, k=2
m = ++k; //порядок: 1) k=k+1=2+1=3, 2) n=k=3. Результат k=3, m=3;
int t = 1, s = 2, z, f;
z = (t++)*5; //порядок обчислень: 1)t*5, 2)t=t+1=2. Результат z=5, t=2
f = (++s)/3; //порядок: 1) s=s+1=3, 2) f=3/3=1. Результат s=3, f=1
```

Обчислення в арифметичних виразах виконуються зліва направо згідно з таким *пріоритетом* операцій:

1) стандартні функції, ++, --;

2) множення (*), ділення (/), остача від ділення (%);

3) додавання (+) й віднімання (–).

Вирази у круглих дужках виконуються першочергово.

Пріоритет усіх операцій наведено в додатку Б.

Для здобуття правильного результату слід дотримуватися таких правил записування арифметичних виразів в операторах С++:

✓ кожна команда (інструкція) має завершуватись крапкою з комою (;);

✓ мова С++ є чутлива до регістру, тобто `x` та `X` – це дві різні змінні;

✓ аргумент функції завжди записують у круглих дужках;

✓ знаки множення не можна пропускати (`3ab` –> `3*a*b`);

✓ якщо знаменник чи чисельник має операції (+, –, *, /), то його слід записувати у круглих дужках;

✓ для записування раціональних дробів, у чисельнику чи знаменнику яких є числові константи, хоча б одну з цих констант слід записати як дійсне число із зазначенням десяткової крапки, наприклад, $\frac{2}{k}$ записують як `2.0/k` чи `2./k`;

✓ радикали (тобто корінь кубічний і вище) замінюють на дробові степені,

наприклад, $\sqrt[3]{x+1}$ записують як `pow(x+1, 1./3);`

✓ слід враховувати правила зведення типів, оскільки в арифметичних виразах можуть брати участь різнотипні дані й відбувається зведення типів.

3.5.3 Оператори присвоювання

Оператор присвоювання – основний оператор програмування – має формат:

<*ім'я_змінної*> = <*вираз*>;

Цей оператор обчислює значення *виразу* і надає здобуте значення *змінній*; при цьому слід враховувати відповідність типів виразу і змінної.

У мові С++, окрім простого присвоювання, є і складені операції присвоювання: += (присвоювання з додаванням), –= (присвоювання з відніманням), *= (присвоювання з множенням), /= (присвоювання з діленням) тощо (табл. 3.4). Наприклад, вираз `x += y` є еквівалентний до виразу `x = x + y`, але записується компактніше і виконується швидше.

Таблиця 3.4

Операції присвоювання

Вигляд	Операція	Типи операндів і результату	Приклади
=	присвоювання	будь-які	`x = y`
+=	присвоювання з додаванням	арифметичні, вказівники, структури, об'єднання	`x += y`
–=	присвоювання з відніманням	арифметичні, вказівники, структури, об'єднання	`x –= y`
*=	присвоювання з множенням	арифметичні	`x *= y`
/=	присвоювання з діленням	арифметичні	`x /= y`
%=	присвоювання остачі від ділення цілих чисел	цілі	`x %= y`
<=	присвоювання зі зсувом ліворуч	цілі	`x <= y`
>=	присвоювання зі зсувом праворуч	цілі	`x >= y`
&=	присвоювання з порозрядною операцією І	цілі	`x &= y`
^=	присвоювання з порозрядною операцією виключне АБО	цілі	`x ^= y`
|=	присвоювання з порозрядною операцією АБО	цілі	`x |= y`

Особливістю звичайного оператора присвоювання є те, що він може використовуватись у виразах, наприклад:

```
if((f = x – y) > 0) ... ;
```

і допускає багатократне застосування, наприклад:

```
a = b = c = x * y;
```

Виконується ця команда справа наліво, тобто спочатку обчислюється значення виразу `x * y`, після чого це значення присвоюється `c`, потім `b` і лише потім `a`.

3.5.4 Зведення типів

Перетворення типів виконуються, якщо операнди, які входять до виразу, мають різні типи. Зведення типів здійснюється автоматично за правилом: менш точний тип зводиться до більш точного. Наприклад, якщо в арифметичному виразі беруть участь коротке ціле (`short`) і ціле (`int`), результат зводиться до `int`, якщо ціле і дійсне – до дійсного.

Розглянемо загальні правила зведення типів:

1) операнди типу `float` зводяться до типу `double`;

2) якщо один операнд `long double`, то другий теж зводиться до цього типу;

3) якщо один операнд `double`, а другий – `float` чи ціле число, то другий операнд перетворюється до типу `double`;

4) операнди цілих типів `char` та `short` зводяться до типу `int`;

5) цілі операнди типу `unsigned char` та `unsigned short` зводяться до типу `unsigned int`;

6) якщо один операнд типу `unsigned long`, то другий цілий операнд перетворюється до типу `unsigned long`;

7) якщо один операнд типу `long`, то другий зводиться до типу `long`;

8) якщо один операнд типу `unsigned int`, то другий цілий операнд зводиться до цього ж типу.

Отже, можна зауважити, що при обчисленні виразів операнди зводяться до типу того операнда, котрий має більший розмір, наприклад:

```
unsigned char  ch;
double  ft, sd;
unsigned long  n;
int  i;
sd = ft * (i + ch / n);
```

При виконанні цього присвоювання правила зведення використовуватимуться у такий спосіб. Операнд `ch` зводиться до `unsigned int` (правило 5). Після чого він зводиться до типу `unsigned long` (правило 6). За цим самим правилом `i` зводиться до `unsigned long`, і результат операцій у круглих дужках матиме тип `unsigned long`. Тоді він зводиться до типу `double` (правило 3), і результат всього виразу матиме тип `double`.

Розглянемо кілька правил зведення типів.

1) Результат операції ділення буде цілим числом, якщо ділене і дільник є цілими, і дійсним числом, якщо один з операндів є дійсного типу. Наприклад, результатом 2/3 буде 0, а результатом 2.0/3 чи 2./3 – 0.666(6). Наведемо ще один приклад, коли обидва операнди є цілими змінними:

```
int m = 2, n = 3;
float a = m / n;        // в результаті a = 0
```

2) При присвоюванні остаточний результат зводиться до типу змінної, яка стоїть ліворуч “=”; при цьому тип може як підвищуватися, так і знижуватись. Якщо тип лівого операнда є менш точним, аніж тип результату виразу, розмі-

щеного праворуч оператора присвоювання, то можлива втрата точності чи то взагалі неправильний результат, наприклад:

```
float a = 2.8, b = 1.7;
int c = a * b;
```

Внаслідок виконання цих команд змінна `c` набуде значення 4, а не 4.76, тобто дійсну частину числа буде втрачено, оскільки `c` є цілою змінною.

3) Ще один приклад дає неправильний результат:

```
double a = 300, b = 200;
short c = a * b;
```

Після виконання цих команд змінна `c` набуде значення –5536 замість очікуваного 60 000. Це пов'язано з тим, що змінна типу `short` може зберігати значення не більше за 32767.

Зведення типів може бути неявним, при виконанні операцій та виклику функцій, чи явним, при виконанні операцій зведення типів.

Операцію явного зведення типів слід застосовувати, щоб уникнути помилок, подібних до вищенаведених прикладів. Ця операція має вигляд:

(<*тип*>) <*арифметичний_вираз*>

тобто перед ім'ям змінної у дужках зазначається тип, до якого її слід перетворити. Наприклад, вищерозглянутий приклад до правила 1 буде обчислювати правильний результат, якщо записати його у такий спосіб:

```
int m = 2, n = 3;
double a = (double) m/n;        // У результаті a=0.66(6)
int  i = 2;  long  l = 2;  double  d;  float  f;
d = (double) i * (double) l;
f = (float)d;
```

У цих прикладах величини `m`, `n`, `i`, `l`, `d` зводяться явно до зазначених у круглих дужках типів.

Потреба у зведенні типів виникає, наприклад, у разі, коли функція повертає вказівник на тип `void`, який слід присвоїти змінній конкретного типу для подальших операцій з нею:

```
float * w= (float *) malloc(25 * sizeof(float));
```

Явне зведення типу є джерелом можливих помилок, оскільки вся відповідальність за його результат покладається на програміста. Операцію явного зведення типів слід використовувати обережно, лише якщо іншого способу здобуття коректного результату немає. Наприклад, у виразі `z/(x+25)`, де `z`, `x` – цілі змінні, дробова частина втрачається. Запобігти цьому можна, якщо перетворити чисельник чи знаменник до дійсного типу. Для цього слід використати один з трьох способів: 1) приписати десяткову точку до числа 25. Вираз набуде вигляду `z/(x+25.)`; 2) домножити чисельник чи знаменник ліворуч на 1.0. Вираз набуде вигляду `1.0*z/(x+25)`; 3) явно зазначити тип результату `(double)z/(x+25)`.

Явне зведення типів дозволяє подати дані у зручному для програміста вигляді. Застосування явного зведення типів у наведеному нижче прикладі дозво-

ляє здобути зображення ASCII-символів за їхнім порядковим номером:

```
for(int i=32; i <= 125; i++) cout << (char)i;
```

3.5.5 Математичні функції

Математичні функції широко використовуються для запису різних математичних залежностей та виразів. Список математичних функцій C++ наведено у табл. 3.5.

Таблиця 3.5

Основні математичні функції

Функція	Опис	Заголовний файл
`int` **`abs`**`(int i)`	модуль (абсолютне значення) цілого числа x – $\|x\|$	stdlib.h
`double` **`fabs`**`(double x)`	модуль дійсного числа x – $\|x\|$	math.h
`double` **`sqrt`**`(double x)`	корінь квадратний – $\sqrt{x}$	math.h
`double` **`pow`**`(double x, double y)`	піднесення x до степеня y – x^y	math.h
`Extended` **`IntPower`** `(Extended x, int n)`	піднесення x до цілого степеня n – x^n	Math.hpp
`double` **`exp`**`(double x)`	експонента e^x	math.h
`double` **`log`**`(double x)`	натуральний логарифм – $\ln(x)$	math.h
`double` **`log10`**`(double x)`	десятковий логарифм – $\lg(x)$	math.h
`Extended` **`LogN`**`(Extended N, Extended x)`	логарифм x за основою N – $\log_N(x)$	Math.hpp
`double` **`cos`**`(double x)`	косинус – $\cos(x)$	math.h
`double` **`sin`**`(double x)`	синус – $\sin(x)$	math.h
`double` **`tan`**`(double x)`	тангенс – $\mathrm{tg}(x)$	math.h
`double` **`Cotan`**`(double x)`	котангенс – $\mathrm{ctg}(x)$	Math.hpp
`double` **`acos`**`(double x)`	арккосинус – $\arccos(x)$	math.h
`double` **`asin`**`(double x)`	арксинус – $\arcsin(x)$	math.h
`double` **`atan`**`(double x)`	арктангенс – arctg (x)	math.h
`double` **`atan2`**`(double y, double x)`	арктангенс – arctg (y/x)	math.h
`double` **`cosh`**`(double x)`	гіперболічний косинус – $(e^x + e^{-x})/2$	math.h
`double` **`sinh`**`(double x)`	гіперболічний синус – $(e^x - e^{-x})/2$	math.h
`double` **`ceil`**`(double x)`	округлення доверху: найменше ціле, не менше за x	math.h
`double` **`modf`**`(double x, double &z)`	виокремлює у дійсному числі x дробову частину і повертає у якості результату, а цілу частину числа доправляє на адресу вказівника *z	math.h
`double` **`floor`**`(double x)`	округлення донизу: найбільше ціле, не більше за x	math.h

Закінчення табл. 3.5

Функція	Опис	Заголовний файл
M_PI	константа π = 3.14159	math.h
M_PI_2	константа π/2	math.h
M_PI_4	константа π/4	math.h
M_1_PI	константа 1/π	math.h
M_2_PI	константа 2/π	math.h
M_1_SQRTPI	константа $1/\sqrt{\pi}$	math.h
M_2_SQRTPI	константа $2/\sqrt{\pi}$	math.h
M_E	константа 2.71828182845904523536	math.h

Якщо треба використовувати у програмі математичні функції, слід долучити бібліотеку, яка містить ці функції, тобто власноруч увести директиву:

```
#include <math.h>      // долучення бібліотеки math.h
#include <Math.hpp>    // долучення бібліотеки Math.hpp
```

Зауважимо, що заголовний файл <math.h> є стандартною математичною бібліотекою для мови С++, а <Math.hpp> є бібліотекою Borland C++ Builder.

У табл. 3.6 наведено приклади запису арифметичних виразів.

Таблиця 3.6

Приклади запису виразів мовою С++

Математичний запис виразу	Запис виразу мовою С++
$y = \cos x^2 + \sin^2 x$	`y = cos(x *x)+ pow(sin(x), 2);`
$y = \sin^4 x$	`y = pow(sin(x), 4);`
$y = \sqrt[5]{x}$ $(\sqrt[5]{x} = x^{1/5})$	`y = pow(x, 1./5);`
$y = \dfrac{\sin(x-\pi)+1}{e^x - 2.5x}$	`y = (sin(x - M_PI) + 1)/(exp(x)-2.5*x);`

3.6 Поширені функції перетворювання числових типів даних у C++ Builder

Уведення й виведення даних в С++ Builder відбувається у текстовому вигляді. Якщо записати, наприклад в Edit1, значення 3,25, то це є не саме число, а його зображення у текстовому вигляді типу AnsiString. З ним жодних обчислень робити не можна аж до перетворення типу AnsiString на числовий тип. У С++ Builder такі перетворення введених рядків на числа, а також зворотні перетворення чисел на рядки для виведення результатів обчислень, здійснюють такі функції:

```
X=StrToInt(S);     //Функція, що перетворює AnsiString-рядок S на ціле число X;
X=S.ToInt();       //Функція, що перетворює AnsiString-рядок S на ціле число X;
```

```
X=StrToFloat(S); //Функція, що перетворює AnsiString-рядок S на дійсне число X;
X=S.ToDouble();  //Функція, що перетворює AnsiString-рядок S на дійсне число X;
S=IntToStr(X);   //Функція, що перетворює ціле число X на AnsiString-рядок S;
S=FloatToStr(X); //Функція, що перетворює дійсне число X на AnsiString-рядок S;
S=FormatFloat("<формат>",X); /* Функція, що перетворює дійсне число X на
                    AnsiString-рядок S у заданому форматі, наприклад для
                    X = 3.4512 результатом команди S = FormatFloat("0.00",X);
                    буде S = 3.45*/
S=FloatToStrF(X,<формат>,n,m); /* Функція, що перетворює дійсне число X
                    на AnsiString-рядок S у заданому форматі з n цифр, з них
                    після коми m, наприклад для X = 653.732529 результатом
                    команди S = FloatToStrF(X, ffFixed, 5, 2); буде S = 653.73 */;
```

Для функції `FloatToStrF()` окрім фіксованого формату `ffFixed`, існують такі формати: `ffGeneral` – універсальний, `ffExponent` – експоненціальний формат з рухомою крапкою, `ffNumber` – з роздільниками груп розрядів, `ffCurrency` – фінансовий. Приміром, число 2345.6 ця функція при різних форматах виведе так: `ffFixed` – 2345.60; `ffGeneral` – 2345.6; `ffExponent` – 2.3456E+04; `ffNumber` – 2,345.6; `ffCurrency` – $2,345.60.

Розглянемо докладно кілька прикладів для введення і виведення з компонента `Edit1` числових змінних різного типу.

```
int k; double x;
k=StrToInt(Edit1->Text); /* Значення властивості Text компонента Edit1
                    перетворюється функцією StrToInt() з типу AnsiString до int,
                    і результат перетворення записується до цілої змінної k.
                    Отже, відбувається введення значення цілої змінної k із
                    компонента Edit1. */
Edit1->Text=IntToStr(k); /* Значення цілої змінної k перетворюється функцією
                    IntToStr() до типу AnsiString, і результат перетворення при-
                    воюється властивості Text компонента Edit1. Відбувається
                    виведення значення цілої змінної k до компонента Edit1 */
x=StrToFloat(Edit1->Text); /* Значення властивості Text компонента Edit1
                    перетворюється функцією StrToFloat() із типу AnsiString
                    до дійсного типу і результат перетворення записується
                    у змінну x. */
Edit1->Text=FloatToStr(x); /* Значення дійсної змінної x перетворюється
                    функцією FloatToStr() до типу AnsiString, і результат пере-
                    творення присвоюється властивості Text компонента Edit1.
                    Кількість значущих цифр цього числа обмежується 15-ма
                    знаками, а якщо їх недостатньо, число виводиться в експоне-
                    нціальній формі. */
Edit1->Text=FormatFloat("0.000",x); /* Значення дійсної змінної x перет-
                    ворюється функцією FormatFloat() до типу AnsiString
                    у заданому форматі, який визначає кількість цифр дійсної
                    частини числа – у даному прикладі їх 3. Результат перетво-
                    рення присвоюється властивості Text компонента Edit1. */
```

3.7 Функції С++ генерування випадкових чисел

У табл. 3.7 наведено синтаксис і опис функцій С++ для генерування випадкових чисел.

Таблиця 3.7

Функції роботи з випадковими числами в С++

Прототип	Опис	Заголовний файл
`int` **`random`** `(int num);`	Повертає випадкове ціле додатне число в діапазоні від 0 до (`num-1`). Наприклад, команда `int x= random(100);` надасть змінній випадкове число в межах від 0 до 99	`stdlib.h`
`int` **`rand`**`(void);`	Повертає випадкове ціле число у діапазоні від 0 до символьної константи `RAND_MAX`. Ця константа визначена у файлі `<stdlib.h>` значенням 32767. Початкове значення генерованого числа задається звертанням до функції `srand()`. Для того щоб генерувати різні послідовності випадкових чисел, початкова точка має обиратися випадково. Для цього можна використовувати функцію `randomize()`. Наприклад, команда `int x=rand() % 100;` надасть змінній випадкове число в межах від 0 до 99	`stdlib.h`
`void` **`randomize`**`(void);`	Ініціалізує генератор випадкових чисел, використовуючи поточний час, який повідомляє комп'ютер	`stdlib.h`, `time.h`

`void` **`srand`** `(unsigned seed);`	Встановлює початкове число `seed` для генерованої за допомогою функції `rand()` послідовності випадкових чисел. Якщо `seed=1`, генератор випадкових чисел ініціалізується значенням за замовчуванням. Наприклад, команди
	`int i; time_t t;`
	`srand((unsigned) time(&t));`
	`for(i=0; i<10; i++)`
	`printf("%d\n", rand() % 100);`
	генеруватимуть послідовність з 10-ти випадкових чисел
		`stdlib.h`

Приклади програм із застосуванням наведених функцій дивіться в наступних підрозд., а саме: 4.22, 4.43, 4.46, 4.55, 5.14, 12.5, 13.6, 13.8.

Питання та завдання для самоконтролю

1) Охарактеризуйте мову C++ і середовище C++ Builder.
2) Що називають лексемами у програмуванні?
3) Як записуються коментарі в C++?
4) Що відбувається при оголошенні змінної?
5) З яких символів може складатися ім'я (ідентифікатор) змінної?
6) Що визначає тип даних?
7) Які базові типи даних C++ Вам відомі?
8) Що означають специфікатори `signed` та `unsigned` і до яких типів даних їх можна застосовувати?
9) Яких значень можуть набувати змінна логічного типу `bool`?
10) Коли використовується тип даних `void`?
11) Назвіть призначення операції `sizeof`.
12) В який спосіб можна оголошувати власні типи?
13) Які особливості оголошування констант в C++?
14) Який тип надається константам за замовчуванням?
15) Що називають операндом, виразом, оператором?
16) Перелічіть арифметичні операції C++.
17) Яких значень набудуть змінні `j` та `i` після обчислення для `int j,i=2`:

а) `j = i++ + i++;` в) `j = i++ + ++i;`
б) `j = ++i + ++i;` г) `j = ++i + i--;`

18) Запишіть числові константи 0.2731e3, 2.5e–4 у фіксованому форматі.
19) Запишіть числа 0.0001, 2500 у формі з рухомою крапкою.
20) Обчисліть вирази для `int a=5, b=7, c=13, d=3`:

а) `a * b / (c - d);`
б) `(b + c) % (a + d)`

21) Якого значення набуде змінна `y` після обчислення:

а) `y = (1 + 2) / (3 + 2);` в) `y = 1 + 2. / 3 + 2;`
б) `y = 1 + 2 / 3 + 2;` г) `y = (1 + 2) / (3. + 2);`

22) Запишіть вирази $y = x^3 + \sin 2x$, $y = \dfrac{1+\sin x^2}{\sqrt{5a} + \ln x}$ засобами C++.
23) У чому полягає відмінність арифметичних операції `/` та `%`?
24) Якими правилами визначається послідовність виконання операцій у виразах?
25) Які дії виконує оператор присвоювання?
26) Коли використовують операцію явного зведення типів?
27) В який спосіб можна долучити бібліотеки математичних функцій?
28) Яка стандартна функція перетворює ціле число на рядок символів?
29) Яка стандартна функція перетворює рядок символів на дійсне число?
30) Запишіть команду для введення дійсної змінної `w` з компонента `Edit1`.

Розділ 4

Програмування базових алгоритмів

4.1 Види базових алгоритмів

Базові алгоритми організації обчислень поділяють на три основні види:

- ✓ лінійні (послідовності);
- ✓ розгалужені;
- ✓ циклічні.

Переважно вони є окремими частинами обчислювального процесу, тоді як загальний обчислювальний процес має складнішу (комбіновану) структуру. Зазвичай при написанні програми ці базові алгоритми поєднуються в такій структурі.

У *лінійних* алгоритмах усі операції виконуються послідовно у порядку їхнього запису. Типовим прикладом такого алгоритму є стандартна обчислювальна схема, яка складається з трьох етапів:

- ✓ введення початкових даних;
- ✓ обчислення за формулами;
- ✓ виведення результату.

У *розгалужених* алгоритмах для здобуття кінцевого результату передбачається вибір одного з кількох можливих напрямів обчислень (гілок) залежно від результату перевірки певної умови. Напрям обчислень обирається перевіркою, внаслідок якої можливі два випадки:

- ✓ "Так" – умову виконано;
- ✓ "Ні" – умову не виконано.

Циклом називається послідовність дій, яка багаторазово повторюється; алгоритм, який містить цикл, має назву *циклічного*. Цикли застосовують, коли для розв'язання задачі виникає потреба повторення низки кроків. Це призводить до багаторазового виконування одних і тих самих операторів у програмі за різних значень змінних. Керування повторенням циклу здійснюється за допомогою змінної, яка називається *параметром циклу*. Спочатку цьому параметру присвоюється певне початкове значення. Потім цикл виконується зі зміною параметра за кожного повторення від початкового до кінцевого значень на величину, що називається *кроком циклу*. Крок циклу може бути додатним чи від'ємним. Залежно від цього параметр циклу зростає чи зменшується. Цикл припиняється, якщо параметр циклу має значення, яке лежить поза межами діапазону між початковим і кінцевим значеннями.

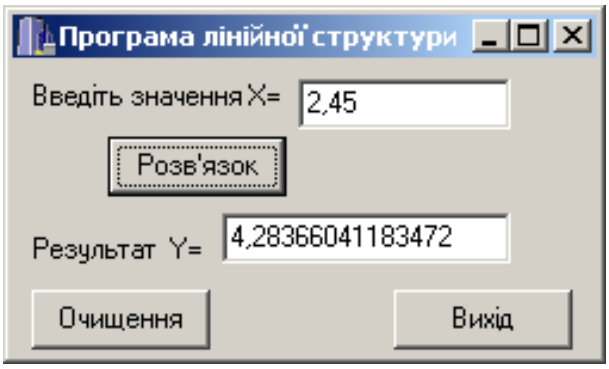
4.2 Програмування лінійних алгоритмів

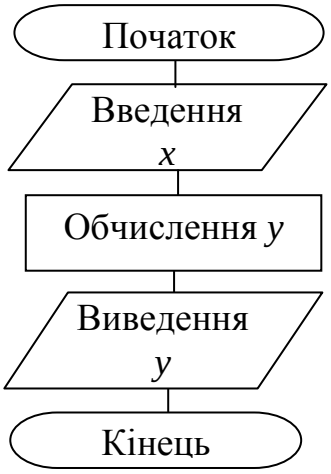
Лінійним називається алгоритм, в якому всі дії, від першої до останньої, виконуються послідовно. Іноді таку структуру називають просто *послідовністю*. Зазвичай у програмній реалізації таких обчислень використовують оператори введення початкових даних, оператори обчислення та оператори виведення результатів.

Приклади програм з лінійною структурою в C++ Builder

Приклад 4.1 Розробити схему алгоритму і створити програмний проект в C++ Builder для обчислення $y = \frac{0.2x^2 - x}{(\sqrt{3}+x)(1+2x)} + \frac{2(x-1)^3}{\sin^2 x + 1}$, де x – довільна змінна, яку слід увести.

Розв'язок. Робочий вигляд форми проекту і схема алгоритму розв'язування цього прикладу мають вигляд:





Послідовність виконування завдання:

1) Запустити C++ Builder.

2) На вікно форми слід встановити компоненти: два надписи `Label`, два компоненти `Edit` та три кнопки `Button`. Ці компоненти розташовано на палітрі компонентів вкладки Standard. Після розміщення компонентів на формі треба встановити властивості компонентів згідно з таблицею, поданою нижче. Для цього у вікні Object Inspector на закладці Properties слід обрати для кожного із зазначених компонентів властивість `Caption` і встановити її нове значення.

Компонент	Властивість	Значення
`Form1`	`Caption`	Програма лінійної структури
`Label1`	`Caption`	Введіть значення *X*=
`Label2`	`Caption`	Результат *Y*=
`Button1`	`Caption`	Розв'язок
`Button2`	`Caption`	Очищення
`Button3`	`Caption`	Вихід

3) Після створення форми слід задати для командних кнопок `Button` шаблони для відгуку на подію `OnClick` подвійним клацанням по відповідних кнопках і вписати поміж операторних дужок шаблонів програмний код для кожної кнопки.

Текст програмного коду:

```
//---------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
```

```
#include "Unit1.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
#include <math.h>   // Долучення математичної бібліотеки
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender) // "Розв'язок"
{float y, x=StrToFloat(Edit1->Text);      // Введення x з Edit1
 y=(0.2*x*x-x)/((sqrt(3)+x)*(1+2*x))+2*pow(x-1,3)/(pow(sin(x),2)+1);
 Edit2->Text=FloatToStr(y);               // Виведення y в Edit2
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender) // "Очищення"
{ Edit1->Clear(); Edit2->Clear();  // Очищує вікна Edit1 та Edit2
  Edit1->SetFocus();               // Встановлює курсор (фокус) в Edit1
}
//---------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender) // "Вихід"
{ Close();
}
```

4) Перед запуском на виконання програмний проект слід зберегти в *окремій* теці командою **File / SaveAll** (чи то через ярлик  , чи клавішами <Shift>+<Ctrl>+<S>). Унаслідок цих дій почергово з'являться два діалогових вікна, в яких слід у власно створеному новому каталозі зберегти і Unit і Project.

5) Для запуску проекту слід виконати команду меню **Run / Run** чи то натиснути на клавіатурі кнопку <F9>. Має з'явитися форма проекту, зображена на стор. 82 (це означає що, що програму написано без фізичних помилок). Якщо в перебігу створювання програми було припущено помилок, то рядок з помилкою висвітиться червоним кольором, а повідомлення компілятора, розташоване під вікном редактора коду, повідомить, якої саме помилки було припущено.

Приклади повідомлень щодо можливих помилок (детальніше про помилки див. розд. 15):

statement missing; – попередній оператор не завершено крапкою з комою (;);

undefined symbol "X" – є невідомий символ, тобто для *X* не визначено тип;

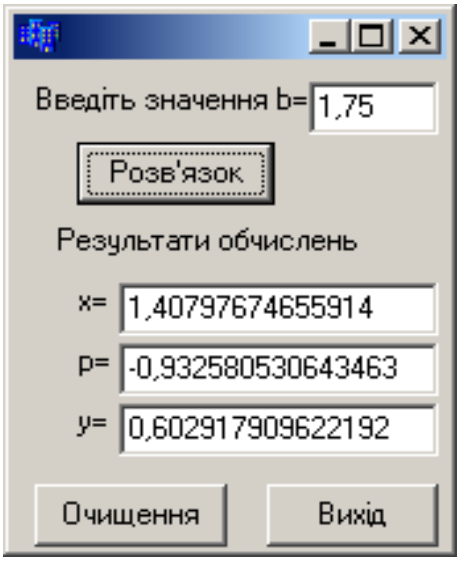
call to undefined function 'sin' – виклик невизначеної функції, тобто не долучено бібліотеку математичних функцій;

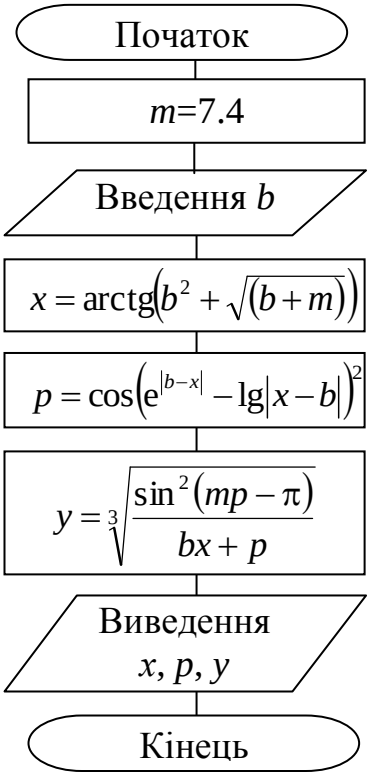
compound statement missing } – відсутня операторна дужка }.

Після виправлення усіх помилок слід зберегти зміни та запустити проект на виконання, виконати обчислення та записати результати обчислень до протоколу виконання роботи.

Приклад 4.2 Розробити схему алгоритму і програмний проект для обчислення $y=\sqrt[3]{\dfrac{\sin^2(mp-\pi)}{bx+p}}$; $x=\operatorname{arctg}\left(b^2+\sqrt{(b+m)}\right)$ і $p=\cos\left(e^{|b-x|}-\lg|x-b|\right)^2$. Значення m=7.4 задати як константу, а значення змінної b ввести за допомогою компонента Edit.

Розв'язок. Блок-схема розв'язування задачі і робочий вигляд форми мають вигляд:



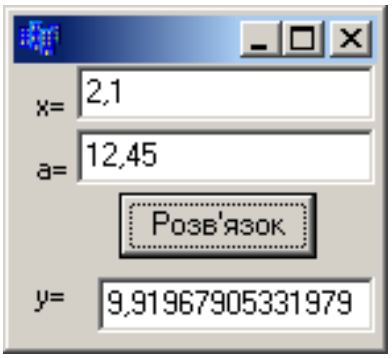


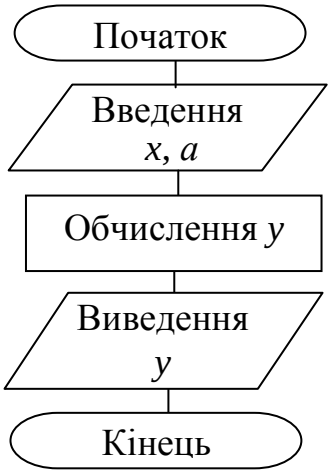
Текст програми:

```
#include <math.h> // Математична бібліотека
//---------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender) // "Розв'язок"
{ const float m=7.4;
  float b, x, p, y;
  b=StrToFloat(Edit1->Text);
  x=atan(b*b+sqrt(b+m));
  p=cos(pow(exp(fabs(b-x))- log10(fabs(x-b)),2));
  y=pow(pow(sin(m*p-M_PI),2)/(b*x+p),1./3);
  Edit2->Text=FloatToStr(x);
  Edit3->Text=FloatToStr(p);
  Edit4->Text=FloatToStr(y);
}
//---------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender) // "Очищення"
{ Edit1->Clear();   Edit2->Clear();
  Edit3->Clear();   Edit4->Clear();
}
//---------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender) // "Вихід"
{ Close();
}
```

Приклад 4.3 Розробити схему алгоритму і створити програмний проект в C++ Builder для обчислення $y=\dfrac{\left(\sin^3 x^{1.5a}+\operatorname{arctg}^3\sqrt{\left|x+\sqrt{ax}\right|}\right)^4}{\left(\ln\left(\left|x\right|+\mathrm{e}^{\sqrt{\lg^3\left(0.5x^2\right)^3}}\right)+x^2\right)}$, де x, a – довільні змінні, які ввести з форми.

Розв'язок. Блок-схема розв'язування задачі і робочий вигляд форми мають вигляд:





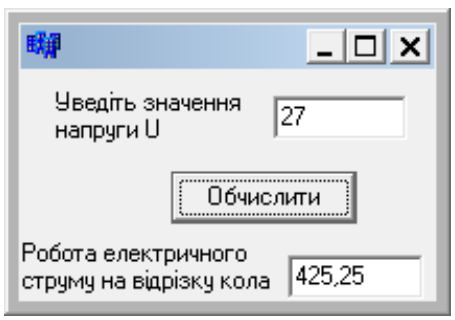
Текст програми:

```
#include <math.h>

void __fastcall TForm1::Button1Click(TObject *Sender)
{ double a, x, y;
  x=StrToFloat(Edit1->Text);
  a=StrToFloat(Edit2->Text);
  y=pow(pow(sin(pow(x, 1.5*a)), 3) +
    pow(atan(sqrt(fabs(x + sqrt(a*x)))), 3), 4) / (log(fabs(x) +
    exp(sqrt(pow(log10(pow(0.5* x*x, 3)), 3)))) + x*x);
  Edit3->Text=FloatToStr(y);
}
```

Приклад 4.4 Обчислити роботу електричного струму на ділянці кола за формулою $A=\dfrac{U^2}{R}\cdot t$, де опір R = 12 Ом, час t = 7 с, а значення напруги U ввести з клавіатури.

Текст програмного коду:

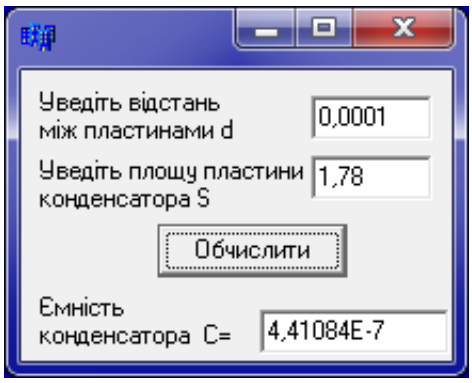
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float R=12, A, t=7, U;
  // Введення значення напруги U
  U=StrToFloat(Edit1->Text);
  // Обчислення роботи електричного струму
  A=pow(U,2)/R *t;
  // Виведення результату
  Edit2->Text=FloatToStr(A);
}
```



Приклад 4.5 Обчислити ємність конденсатора за формулою $C = \frac{\varepsilon_0 \cdot \varepsilon \cdot S}{d}$, де $\varepsilon_0 = 8.85 \cdot 10^{-12}$ Ф/м, $\varepsilon = 2.8$ Ф/м, S – площа кожної пластини конденсатора, d – відстань між пластинами, значення яких увести з клавіатури.

Текст програмного коду:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ const double E0=8.85e-12, E=2.8;
  double d, S, C;
  // Уведення d – відстані між пластинами
  d=StrToFloat(Edit1->Text);
  // Уведення S – площі пластини конденсатора
  S=StrToFloat(Edit2->Text);
  // Обчислення ємності конденсатора
  C=(E0*E*S)/d;
  // Виведення результату
  Edit3->Text=FloatToStr(C);
}
```



Приклад 4.6 Обчислити ослаблення T-образного резисторного атенюа́тора за формулою

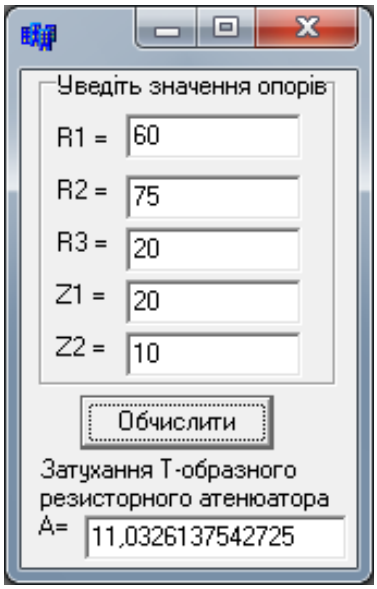
$$A = 20 \cdot \lg \frac{(1 + a \cdot R1/R2) \cdot b \cdot R3 + R1}{C},$$

для якої значення опорів $R1$, $R2$, $R3$, $Z1$, $Z2$ увести з клавіатури, а частотні коефіцієнти обчислити за формулами:

$$a = 1 + \frac{Z1}{R1}; \quad b = 1 + \frac{Z2}{R2}; \quad C = Z1 + Z2.$$

Текст програмного коду:

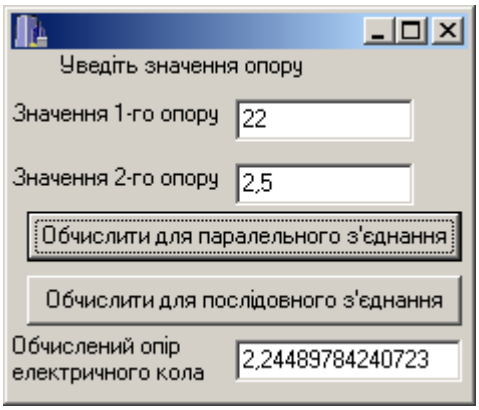
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float R1,R2,R3,Z1,Z2, a,b,C,A;
  R1=StrToFloat(Edit1->Text);
  R2=StrToFloat(Edit2->Text);
  R3=StrToFloat(Edit3->Text);
  Z1=StrToFloat(Edit4->Text);
  Z2=StrToFloat(Edit5->Text);
  // Обчислення частотних коефіцієнтів a, b, C
  a=1+(Z1/R1);
  b=1+(Z2/R2);
  C=Z1+Z2;
  // Обчислення затухання
  A= 20*log10(((1+a*R1/R2)*b*R3+R1)/C);
  // Виведення результату
  Edit6->Text = FloatToStr(A);
}
```

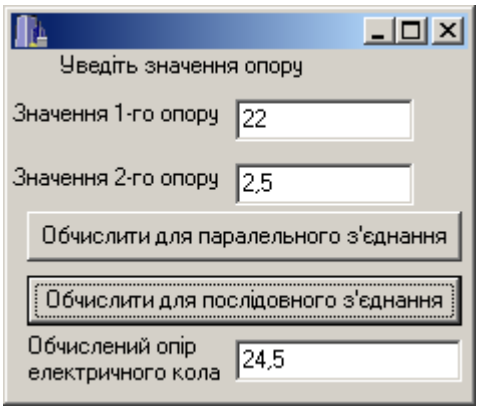


Приклад 4.7 Створити проект для обчислення опору електричного кола, яке складається з двох поєднаних опорів, у два варіанти: 1) за паралельного з'єднання опорів та 2) за послідовного з'єднання опорів в електричному колі.

Розв'язок. Нагадаємо, що формула для обчислення опору електричного кола за паралельного з'єднання опорів має вигляд $R = \frac{r1 \cdot r2}{r1 + r2}$, а за послідовного з'єднання опорів – $R = r1 + r2$.

Обчислення для двох випадків сформуємо у програмі у двох окремих командних кнопках:

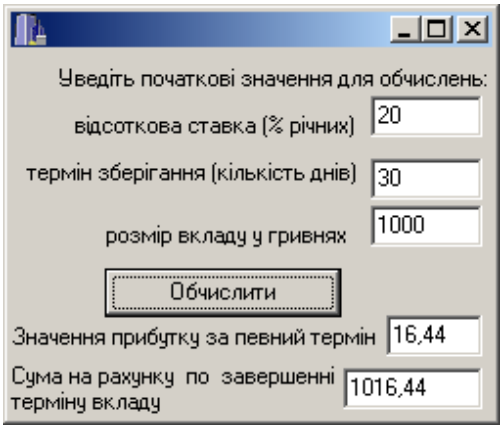




Текст програми:

```
// Обчислити для паралельного з'єднання
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float r1, r2, R;
  r1=StrToFloat(Edit1->Text);
  r2=StrToFloat(Edit2->Text);
  R=r1*r2/(r1+r2);
  Edit3->Text=FloatToStr(R);
}
// Обчислити для послідовного з'єднання
void __fastcall TForm1::Button2Click(TObject *Sender)
{ float R, r1=StrToFloat(Edit1->Text), r2=StrToFloat(Edit2->Text);
  R=r1+r2;
  Edit3->Text=FloatToStr(R);
}
```

Приклад 4.8 Створити програмний проект для обчислення величини прибутку по банківським вкладам залежно від процентної ставки (% річних) і терміна зберігання (кількість днів), значення яких задаються користувачем.



Текст програмного коду:

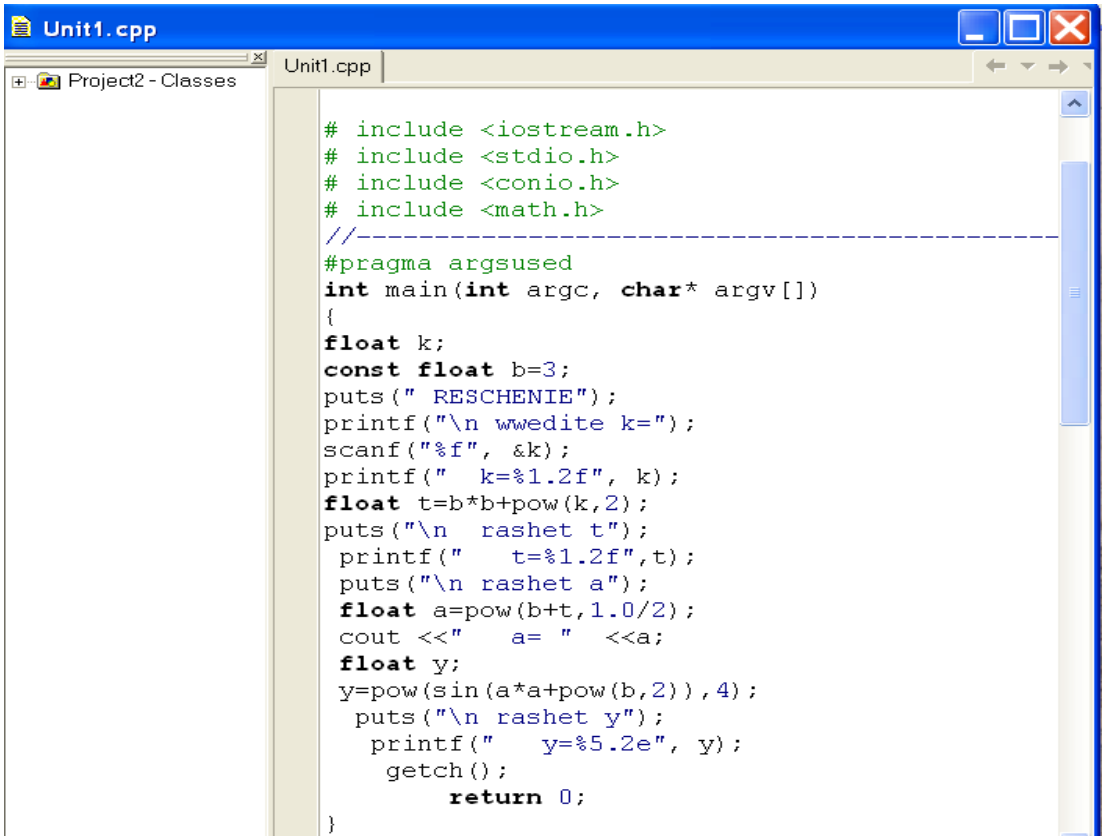
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float stavka, t, sum, pr;
  stavka=StrToFloat(Edit1->Text);          // Процентна ставка
  t=StrToFloat(Edit2->Text);               // Термін (кількість днів) вкладу
  sum=StrToFloat(Edit3->Text);             // Сума вкладу
  pr=sum*stavka/365/100*t;                 // Прибуток по вкладу
  sum=sum+pr;
  Edit4->Text=FormatFloat("0.00",pr);
  Edit5->Text=FormatFloat("0.00",sum);
}
```

Приклади лінійних програм у консольному режимі C++ Builder

Приклад 4.9 Обчислити значення функцій: $y=\sin^4\left(a^2+b^2\right)$, $a=\sqrt{b+t}$, $t=b^2+k^2$, де $b = 2$, $k = 1.2$, причому значення b задати як константу, а значення k – ввести з клавіатури.

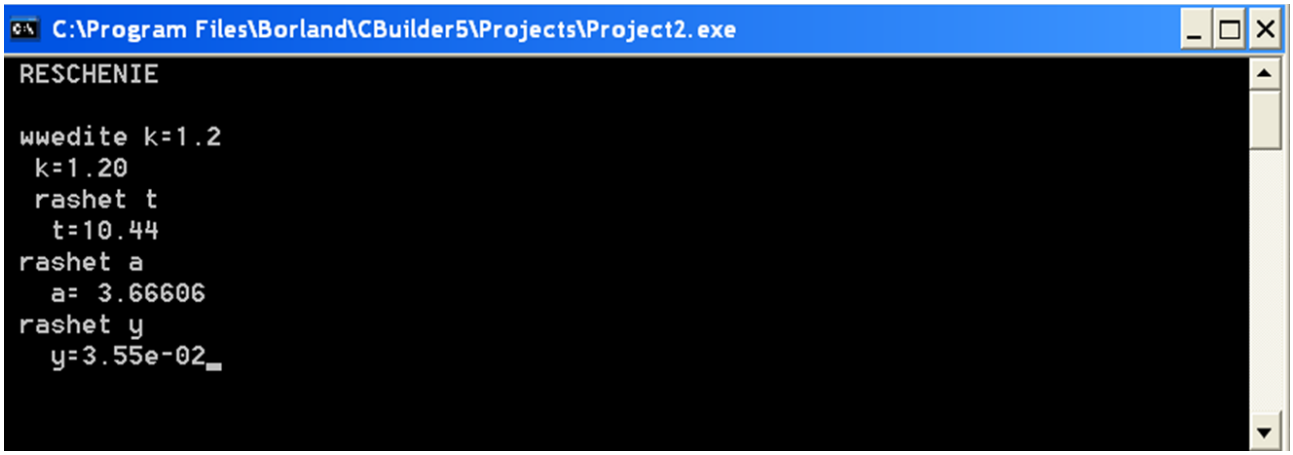
Розв'язок. Для виведення повідомлення використовуватимемо функцію **puts()**; для введення даних з клавіатури – функцію **scanf()**, а для виведення результатів – функції **printf()** та **cout**.

Вікно консольного додатка з текстом програми обчислення:



Рядки, які розпочинаються з подвійної скісної риски (//), містять коментар і не беруться до уваги при компілюванні файла.

Результати роботи програми у консольному додатку:



Зауважимо, що не рекомендується в одній програмі змішувати функції введення-виведення, які надійшли з C (`scanf`, `printf`), з потоковим введенням-виведенням (`cin`, `cout`), яке поширене у C++.

Приклад 4.10 Створити консольний програмний проект для обчислення сили струму електричного кола залежно від значень опору та напруги, які введе користувач.

Текст програмного коду:

```
#include <vcl.h>
#pragma hdrstop
#include <iostream.h>
#include <conio.h>
//---------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{ float U, R, I;
  cout << " Voltage U= " ;
  cin >> U;
  cout << " Resistence R= " ;
  cin >> R;
  I = U/R;
  cout << "Result Current I= " << I << " Amper"<< endl;
  getch();
  return 0;
}
```

Результати роботи консольного додатка:

```
Voltage U= 24
Resistence R= 48
Result Current I= 0.5 Amper
```

4.3 Програмування розгалужених алгоритмів

4.3.1 Розгалужені алгоритми

Алгоритм називається *розгалуженим* якщо певні його команди (інструкції) можуть не виконуватись жодного разу. Це відрізнює такі алгоритми від лінійних. Вочевидь, деякі команди не виконуються за певних вхідних умов, але можуть виконуватися за інших умов. Якщо за аналогією проаналізувати повсякденні життєві ситуації, то можна віднайти численну кількість умов, які розпочинаються словом "якщо". Наприклад, "*якщо на вулиці йде дощ, я залишуся вдома, якщо – ні, то поїду на пікнік*" або "*якщо на вулиці дощ, треба взяти парасолю*".

Отже, *розгалуженням* називається вибір програмою тієї чи іншої низки команд залежно від того, чи виконується певна умова. При цьому спрацьовує лише одна з гілок алгоритму.

Для програмної реалізації таких обчислень слід використовувати оператори передавання керування, котрі дозволяють змінювати порядок виконування операторів програми. У мові C++ для цього передбачено інструкції: безумовного переходу – **`goto`**, умовного переходу – **`if`** та вибору варіанта – **`switch`**. Для записування умови переходу слід використовувати логічні (булеві) вирази.

4.3.2 Оператор безумовного переходу goto

Оператор `goto` (перейти до) дозволяє передавати керування у будь-яку точку коду (програми), котру позначено спеціальною міткою.

Синтаксис оператора `goto`:

`goto` <*мітка*>;

Мітку записують перед оператором, на який слід передати керування, і відокремлюють від цього оператора символом двокрапки (`:`). Мітки в мові C++ не оголошують. У якості мітки може застосовуватись сполучення будь-яких латинських літер та цифр, але розпочинатися мітка повинна з літери, наприклад: `start`, `M1`, `a`, `second`.

Застосовувати цей оператор слід вельми обережно і помірковано, особливо при переході всередину блока чи циклу, оскільки це може призвести до непередбачуваних помилок. Тому в C++ оператор `goto` застосовується рідко і вважається застарілим. А застосування структурного і об'єктно-орієнтованого підходів до програмування дозволяє повністю відмовитись від застосування операторів `goto`. Однак на практиці часто трапляються випадки, коли цей оператор значно спрощує код програми (див. приклад програми 4.25 у п. 4.3.8).

4.3.3 Операції відношення та логічні операції

Операції відношень (<, >, <=, >=, = = (дорівнює), != (не дорівнює)) порівнюють два вирази і видають значення 1 (**`true`** – істина – "так") або 0 (**`false`** – хибність – "ні"). Типом результату є `int` (чи `bool`).

Наприклад:

```
x < y     /* вираз 1 */          x >= y     /* вираз 4 */
y > x     /* вираз 2 */          x = = y    /* вираз 5 */
x <= y    /* вираз 3 */          x != y     /* вираз 6 */
```

Якщо `x`, `y` є однакові, то вирази 3, 4 та 5 мають значення 1, а вирази 1, 2 і 6 – значення 0.

Умову для перевірки певної цілої змінної `k` на ненульове значення `if(k!=0)` можна записати простіше – як `if(k)`, оскільки ненульове значення розцінюється як `true`. Тоді умову `if(k==0)` можна записати як `if(!k)`.

Варто зауважити, що у С++ результатом логічного виразу може бути і цілочисельне арифметичне значення. При цьому значення 0 розцінюється як `false`, а кожне ненульове значення – як `true`. Розглянемо приклади:

```
int  tr = (105<=109);
int  fal = (109>105);
```

Унаслідок виконання цих операторів змінна `tr` набуде значення 1, а змінна `fal` – 0.

Логічні змінні можуть набувати значень: `true` (істина) чи `false` (хибність). Вельми часто ці значення позначають цифрами 1 та 0 (1 – істина, 0 – хибність). Логічний тип ще називають *булевим*, від прізвища англійського математика Джорджа Буля, засновника математичної логіки. При оголошенні змінних булевого типу їх позначають як `bool`, наприклад: `bool m`;

Логічна операція – дія, яка виконується над логічними змінними, її результат є 1 (`true`) або 0 (`false`). Логічні операції обчислюють кожен операнд з огляду на його еквівалентність нулю.

Базові логічні операції:

|| – *логічне додавання* (операція "АБО", *диз'юнкція*), результатом виконання якого є значення 1 (`true`), якщо хоча б один з операндів має ненульове значення. Якщо перший операнд має ненульове значення, то другий операнд вже не обчислюється;

&& – *логічне множення* (операція "І", *кон'юнкція*), результатом виконання якого є 1, якщо обидва операнди мають ненульові значення, у решті випадків результат – 0. Якщо значення першого операнда дорівнює 0, то другий операнд не обчислюється;

! – *логічне заперечення* (операція "НЕ", *інверсія*) виконується над однією логічною змінною, результатом чого є значення `true` (1), якщо початковим значенням було `false` (0), і `false` (0), – якщо початковим значенням було `true` (1), наприклад:

```
bool w, q = true;
w = !q;          // Змінна w набуде значення false
int t, z = 0;
t = !z;          // Змінна t набуде значення 1
```

З логічних змінних за допомогою логічних операцій та дужок (для зазначення порядку дій) будуються логічні вирази. Результати застосування цих

операцій до булевих значень наведено в табл. 4.1. Операнди логічних операцій можуть бути цілого, дійсного типу чи типу вказівника, при цьому в кожній операції можуть брати участь операнди різних типів. Операнди логічних виразів обчислюються зліва направо. Якщо значення першого операнда вистачає, щоб визначити результат операції, то другий операнд не обчислюється. Логічні операції не спричинюють стандартних арифметичних перетворень. Вони оцінюють кожен операнд з огляду на його еквівалентність нулю. Результатом логічної операції є 0 чи 1.

Таблиця 4.1

Булеві операції

A	B	A || B	A && B	! A
0	0	0	0	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	0

Логічні побітові операції виконуються над цілими числами і опрацьовують число побітово. До цих операцій належать: побітове логічне заперечення (~), побітове логічне множення "І" (&), побітове логічне додавання "АБО" (|), побітове логічне "виключне АБО" (^) (табл. 4.2). Операнди логічних операцій мають бути будь-якого цілого типу.

Таблиця 4.2

Логічні побітові операції

Позначення	Операція	Приклади	
~	інверсія, НЕ	`~X`	`~1011 = 0100`
&	І	`X & Y`	`1011 & 1010 = 1010`
|	АБО	`X | Y`	`1011 | 1010 = 1011`
^	виключне АБО	`X ^ Y`	`1011 ^ 1010 = 0001`
<<	зсув ліворуч	`X << 2`	`1011 << 2 = 1100`
>>	зсув праворуч	`Y >> 2`	`1011 >> 2 = 0010`

Зауважимо, що в наведених у табл. 4.2 прикладах для простоти та наочності ми обмежились лише чотирма розрядами двійкових чисел.

Операція побітового логічного заперечення "НЕ" (~) інвертує кожен біт операнда: 0 – на 1, 1 – на 0, тобто продукує двійкове доповнення свого операнда ($\sim 1010_2 = 0101_2$), наприклад:

```
short  x = 987;   unsigned short  y = 0xAAAA;
y = ~y;           // У результаті y = 0x 5555 (шістнадцяткове).
if (!(x < y));  // Результат виразу  0 (false).
```

Операція побітового логічного множення "І" (&) порівнює кожен біт першого операнда з відповідним бітом другого операнда і, якщо обидва порівнюваних біти є одиницями, то відповідний біт результату встановлюється в 1, інакше – в 0. Наприклад, $1010_2 \ \& \ 1001_2 = 1000_2$.

Операція побітового логічного додавання "АБО" (|) порівнює кожен біт першого операнда з відповідним бітом другого операнда, і, якщо хоча б один

(чи обидва) з них дорівнює 1, то відповідний біт результату встановлюється в 1, інакше – біт результату дорівнює 0. Наприклад, $1010_2 \mid 1001_2 = 1011_2$.

Операція побітове логічне "виключне АБО" (^) порівнює кожен біт першого операнда з відповідним бітом другого операнда і, якщо обидва опрацьовувані біти мають однакове значення, результат дорівнює 0, у решті випадків – 1. Наприклад, $1010_2 \wedge 1001_2 = 0011_2$.

Наведемо приклади виконування побітових логічних операцій.

```
short i=0x45FF, //  i= 0100 0101 1111 1111
r,  j=0x00FF;   //  j= 0000 0000 1111 1111
r = i ^ j;      //  r=0x4500 = 0100 0101 0000 0000
r = i | j;      //  r=0x45FF = 0100 0101 0000 0000
r = i & j       //  r=0x00FF = 0000 0000 1111 1111
short n,  i = 0xAB00,  j = 0xABCD;
n = i & j;      // У результаті  n = AB00 (шістнадцяткове)
n = i | j;      // У результаті  n = ABCD (шістнадцяткове)
n = i ^ j;      // У результаті  n = CD (шістнадцяткове)
```

Операції побітового зсуву >> і << виконують зсув бітів лівого операнда на кількість розрядів, зазначену правим операндом, відповідно праворуч чи ліворуч. Значення бітів, яких бракує, доповнюються нулями. При виконанні операції зсуву можуть втрачатися старші чи молодші розряди:

>> – побітовий зсув праворуч на задану кількість бітів (65 >> 2 = 16, оскільки число $65 = 41_{16} = 0100\ 0001_2$ після зсуву праворуч становить $0001\ 0000_2 = 10_{16} = 16$). Операція є еквівалентна до цілочисельного поділу на 2 у відповідному степені (65 / 4 = 16);

<< – побітовий зсув ліворуч (13 << 2 = 20, оскільки число $13 = D_{16} = 1101_2$ після зсуву ліворуч становить $11\ 0100_2 = 34_{16} = 52$). Операція є еквівалентна до цілочисельного множення на 2 у відповідному степені (13 * 4 = 52). Слід пам'ятати, що при цьому є можливе втрачання старших розрядів.

Наведемо кілька прикладів роботи операцій зсуву.

```
int  c = 12, m, n;
m = c >> 2;           //  m=6
n = c << 2;           //  n=24
c = n >> 3            //  c=24/2^3=24/8=3
int  i = 0x1234, j, k ;
k = i << 4;           //  k=0x0234
j = i << 8;           //  j=0x3400
i = j >> 8;           //  i=0x0034
unsigned  int x = 0x00AA, y = 0x5500, z;
z = (x << 8) + (y >> 8);
```

В останньому прикладі `x` зсувається ліворуч на 8 позицій, а `y` – праворуч на 8 позицій. Результати зсувів додаються, утворюючи величину 0xAA55, котра присвоюється `z`.

Застосовування операцій >> та << почергово до однієї й тієї самої змінної може спричинити змінення її значення через втрату розрядів.

Перетворювання, виконувані операціями зсуву, не забезпечують опрацювання ситуацій переповнення і втрати знаків. Знак першого операнда після ви-

конання операції зберігається. Результат операції зсуву є невизначений, якщо другий операнд буде від'ємний.

Побітові операції є зручні для організації зберігання у стислому вигляді інформації про стан on/off (ввімкнено/вимкнено). В одному байті можна зберігати 8 таких "прапорців". Якщо змінна `ch` є сховищем таких "прапорців", то перевірити, чи є "прапорець" третього біту у стані on, можна у такий спосіб:

```
if (ch & 4) . . .
```

Ця перевірка базується на двійковому поданні числа 4 = 0000 0100.

Зауважимо, що вираз `a ^ a` завжди повертає значення 0, а вираз `a ^ b ^ a` – значення `b`. Ці закономірності часто застосовуються у растровій графіці.

4.3.4 Пріоритет логічних операцій

Обчислення в логічних виразах виконуються зліва направо у відповідності з таким *пріоритетом* операцій:

1) ~ , !;
2) >>, <<;
3) <, <=, >, >=;
4) = =, !=;
5) & (побітове "І");
6) ^ (побітове "виключне АБО");
7) | (побітове "АБО");
8) && ("І");
9) || ("АБО").

Вирази у круглих дужках виконуються першочергово.

4.3.5 Пріоритети операцій і порядок обчислень

З урахуванням вищенаведеного пріоритету логічних операцій та пріоритету арифметичних операцій, наведеного в підрозд. 3.5, наведемо загальний пріоритет для всіх операцій мови С в табл. 4.3. Операції с вищими пріоритетами обчислюються першими.

Таблиця 4.3

Пріоритет і порядок виконування операцій

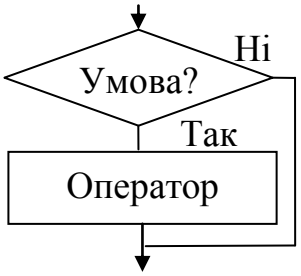
Пріоритет	Знак операції	Тип операції	Порядок виконування
1	::, (), [], ., –>	вираз	зліва направо
2	–, ~, !, *, &, ++, --, sizeof, (<*тип*>)	унарні	справа наліво
3	*, /, %	мультиплікативні	зліва направо
4	+, –	адитивні	
5	<<, >>	зсув	
6	<, >, <=, >=	відношень	
7	==, !=	відношень (порівнянь)	

Продовження табл. 4.3

Пріоритет	Знак операції	Тип операції	Порядок виконування
8	&	побітове “І”	зліва направо
9	^	побітове “виключне АБО”	
10	|	побітове “АБО”	
11	&&	логічне “І”	
12	||	логічне “АБО”	
13	? :	умовна	
14	=, *=, /=, %=, +=, –=, &=, |=, >>=, <<=, ^=	просте і складені присвоювання	справа наліво
15	,	послідовне обчислення	зліва направо

4.3.6 Умовний оператор if

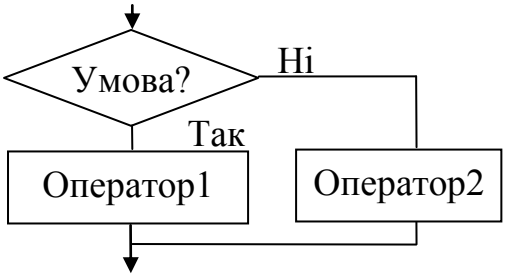
Оператор `if` має дві форми: скорочену та повну.



Скорочена форма має вигляд:

`if` (<*умова*>) <*оператор*>;

Якщо значення логічного виразу в *умові* є ненульове, тобто *умова* є істинна, то виконується *оператор* чи група операторів в операторних дужках {}, інакше відбудеться перехід на наступний оператор.



Повна форма цього оператора:

`if` (<*умова*>) <*оператор1*>;
`else` <*оператор2*>;

Якщо значення логічного виразу в *умові* є ненульове, тобто *умова* є істинна, то виконується *оператор1* чи група операторів в операторних дужках {}, інакше – виконується *оператор2*, після чого відбудеться перехід на наступний оператор. Зауважимо, що обчислюється лише один з операторів, а не обидва.

Наприклад, обчислення значення виразу

$$y = \begin{cases} 1 + b^x & \text{за } x = b; \\ \dfrac{x+b}{b-x} & \text{за } x \neq b \end{cases}$$

можна реалізувати чи то двома операторами `if` скороченої форми, чи одним оператором `if` повної форми:

1)
```
if(x == b) y = 1 + pow(b, x);
if(x != b) y = (x + b)/(b - x);
```
2)
```
if(x == 0) y = 1 + pow(b, x); else y = (x + b)/(b - x);
```

Зауважимо, що умову (x≠0) можна записати чи то як (x != 0), чи як (x). І, навпаки, для того щоб перевірити, чи дорівнює нулю певне число x, умова може мати вид (x==0) чи (!x).

Окрім того, слід розрізняти вирази в умовах типу

```
if(x ==5) . . .;
```

та

```
if(x = 5) . . .;
```

Обидва ці оператори є працездатними, оскільки в C++ будь-який вираз, який має певне числове значення, може використовуватися в умовних операторах. Але, якщо у першому випадку значення логічного виразу може бути як істинним, так і хибним, залежно від значення змінної x, то у другому випадку значення логічного виразу завжди є істинне, оскільки в результаті присвоювання значення x = 5 – ненульове.

Наведемо кілька прикладів використання умовного оператора if.

1) Обчислити частку від ділення двох дійсних чисел, якщо це є можливо.

```
if(y!=0)      // Якщо знаменник не 0, можна обчислити.
  r = x / y;
```

Це приклад скороченої форми умовного оператора (без гілки “інакше”).

2) Обчислити квадратний корінь числа, якщо це є можливо.

```
if(x>=0)      // Якщо під коренем додатне число, можна обчислити,
  y = sqrt(x);
else          // інакше – виводимо повідомлення, що обчислити неможливо.
  ShowMessage("Обчислити неможливо!");
```

Це приклад повної форми умовного оператора (з гілкою “інакше”).

3) Якщо змінні x та y є однаковими, обчислити їхній добуток, інакше – обчислити модуль їх різниці.

```
if(x == y) r = x * y;
else r = fabs(x - y);
```

4) Якщо модуль змінних x та y є менше за 10, збільшити їх удвічі; інакше – зменшити їх удвічі.

```
if(fabs(x)<10 && fabs(y)<10)
 { x=x*2;
   y=y*2;
 }
else
 { x=x/2;
   y=y/2;
 }
```

У цьому прикладі, оскільки за істинності умови має виконуватись набір з двох команд, ці команди слід записати в операторних дужках {}.

Команди в операторних дужках {} називаються *блоком команд*.

Умовні оператори можуть бути необмежено вкладеними один в одного. У деяких випадках складно розібратися, в якій послідовності виконуються такі вкладені оператори. Вельми часто зустрічається використовування конструкції

`if – else – if.` Приміром, щоб визначити знак змінної `x` (вона може бути додатною, від'ємною чи нулем), можна використовувати три простих незалежних команди `if`:

```
if(x>0) ShowMessage("positive");
if(x<0) ShowMessage("negative");
if(x==0) ShowMessage("zero");
```

чи оператор `if – else` вкладеної конструкції:

```
if (x>0) ShowMessage("positive");
else
   if (x<0) ShowMessage("negative");
   else ShowMessage("zero");
```

Тут, якщо спрацював перший `else`, це означає, що `x` є від'ємним числом чи нульовим. Тому слід перевірити обидві гілки за допомогою ще однієї команди `if`.

Обчислення функції

$$y=\begin{cases}\sin(x+a) & \text{за} \quad x\le -2;\\ \cos(x-\pi)^2 & \text{за} \quad -2<x\le 3;\\ \operatorname{arctg}^2\dfrac{x}{a} & \text{за} \quad x>3,\ a\ne 0;\\ 1 & \text{за} \quad x>3,\ a=0,\end{cases}$$

аналогічно до попереднього прикладу, можна реалізувати чи то за допомогою чотирьох операторів `if` скороченої форми, чи оператором `if – else` вкладеної конструкції:

1)
```
if (x <= –2) y = sin(x + a);
if (x > –2 && x <= 3) y = cos(pow(x – M_PI, 2));
if (x > 3 && a != 0) y = pow(atan(x / a), 2);
if (x > 3 && a = = 0) y = 1;
```

2)
```
if (x <= –2) y = sin(x + a);
  else
    if (x <= 3) y = cos(pow(x – M_PI, 2));
    else
       if (a != 0) y = pow(atan(x / a), 2);
       else y = 1;
```

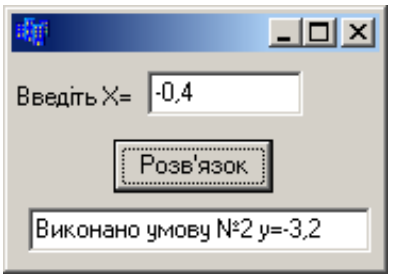
У конструкціях `if – else – if` умови операторів `if` перевіряються зверху донизу. Тільки-но якась з умов набуває ненульового значення, одразу виконуватиметься оператор, який слідує за цією умовою, а останню частину конструкції буде проігноровано.

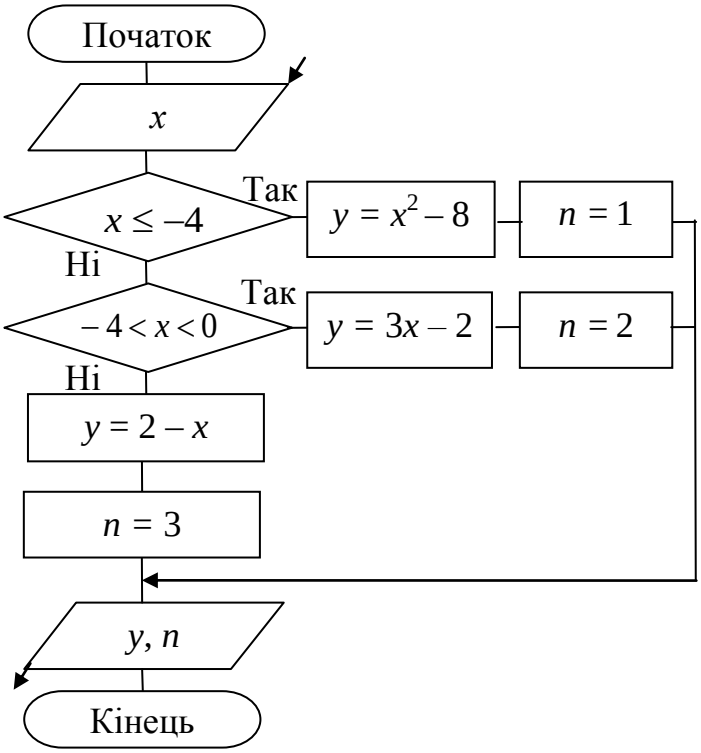
Слід записувати подібні вкладені оператори максимально наочно, використовуючи відступи з пропусків, уникати заплутаних складових і великого рівня вкладеності.

Приклади створювання проектів програм з розгалуженою структурою, організованих за допомогою оператора if

Приклад 4.11 Ввести x та обчислити $y = \begin{cases} x^2 - 8 & \text{за} \quad x \le -4; \\ 3x - 2 & \text{за} \quad -4 < x < 0; \\ 2 - x & \text{за} \quad x \ge 0. \end{cases}$

Розв'язок. Робочий вигляд форми і блок-схема:

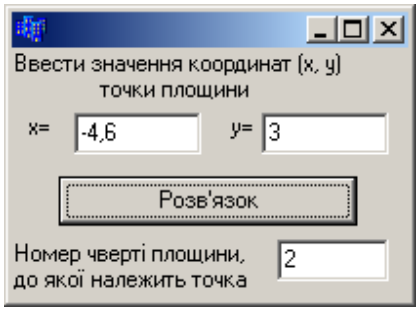


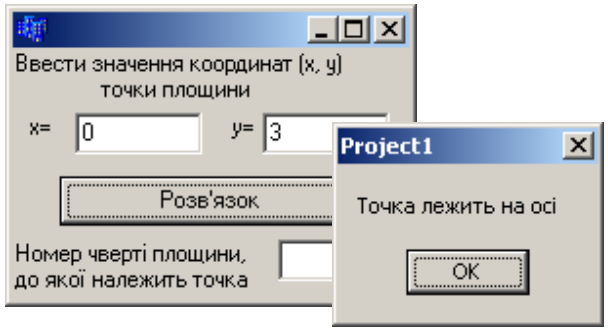


Текст програмного коду:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int n;  double y, x = StrToFloat(Edit1->Text);
  if(x <= -4) { y = x*x - 8; n = 1;}
  else
    if(-4 < x && x < 0) { y = 3*x - 2; n = 2;}
    else      { y = 2 - x; n = 3;}
  Edit2->Text = "Виконано умову №" + IntToStr(n) +
                                          " y="+FloatToStr(y);
}
```

Приклад 4.12 Ввести значення координат (x, y) точки площини і визначити, до якої чверті площини належить ця точка.





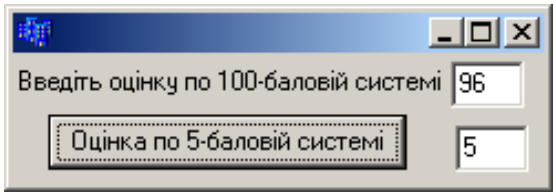
Текст програми:

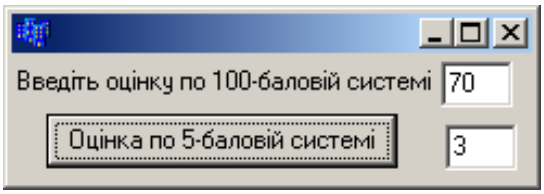
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{float x = StrToFloat(Edit1->Text);
 float y = StrToFloat(Edit2->Text);
 int nomer;
 if((x==0)||(y==0))
  {ShowMessage("Точка лежить на осі");
   Edit3->Clear();
   return;
  }
  else
    if(x>0 && y>0)  nomer = 1;
    else
      if(x<0 && y>0) nomer = 2;
      else
        if(x<0 && y<0) nomer=3;
        else
          nomer = 4;
 Edit3->Text = IntToStr(nomer);
}
```

Функція `ShowMessage();` – призначена для виведення на екран діалогового вікна з текстом відповідного повідомлення та однією командною кнопкою ОК, при натисканні на яку вікно повідомлення зникає.

Приклад 4.13 Перевести оцінку по 100-бальній системі до 5-бальної за алгоритмом: оцінка менша за 60 відповідає незадовільній оцінці “2”; в межах від 60 до 79 – оцінці “3”; від 80 до 94 – “4”; розпочинаючи з 95 і до 100 – “5”.

Зауважимо, що в різних вузах можливі незначні відмінності від наведеного алгоритму.





Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{  int m100, m5;
   m100=StrToInt(Edit1->Text);
   if(m100>=95) m5=5;
   if(m100>=80 && m100<95) m5=4;
   if(m100>=60 && m100<80) m5=3;
   if(m100<60) m5=2;
   Edit2->Text=IntToStr(m5);
}
```

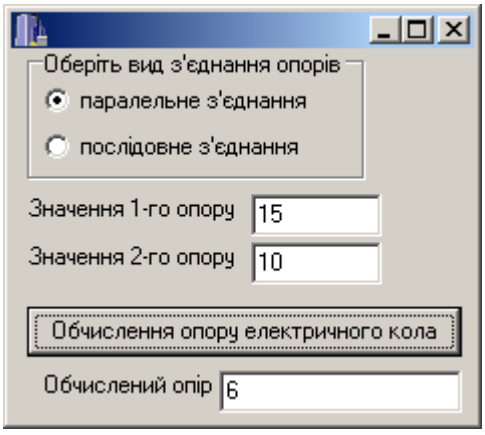
Аналогічна програма із вкладеною конструкцією `if – else – if`:

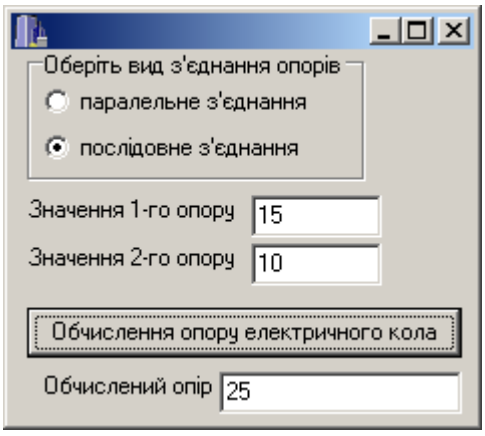
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{   int m100, m5;
    m100=StrToInt(Edit1->Text);
    if(m100>=95) m5=5;
    else
      if(m100>=80) m5=4;
      else
        if(m100>=60) m5=3;
        else m5=2;
    Edit2->Text=IntToStr(m5);
}
```

У цьому варіанті програми немає потреби перевіряти праву межу інтервалів.

Приклад 4.14 Створити програмний проект для обчислення опору електричного кола, яке складається з двох з'єднаних опорів, залежно від виду їхнього з'єднання: паралельне чи послідовне (формули див. у прикладі 4.7).

Розв'язок. Тобто вдосконалимо розв'язок прикладу 4.4. Обирання виду з'єднаних опорів сформуємо за допомогою компонента `RadioGroup`, який дозволяє обирати лише одне зі значень списку, "вмикаючи" відповідну радіокнопку напроти елемента списку. Значення усіх елементів списку (видів з'єднаних опорів) слід записати до властивості `Items`. Кількість елементів списку визначає кількість радіокнопок. Номер обраного елемента (нумерація починаючи з нуля) записується до властивості `ItemIndex`. Деталі роботи з компотнентом `RadioGroup` наведено також у табл. 2.2 та прикладі 4.24.

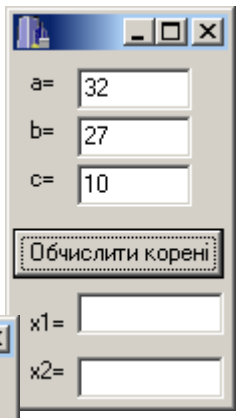


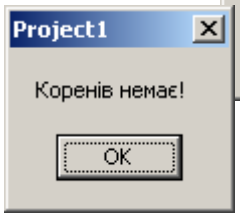


Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float r1, r2, R;
  r1=StrToFloat(Edit1->Text);  r2=StrToFloat(Edit2->Text);
  if(RadioGroup1->ItemIndex==0)
    R=r1*r2/(r1+r2);
  else  R=r1+r2;
  Edit3->Text=FloatToStr(R);
}
```

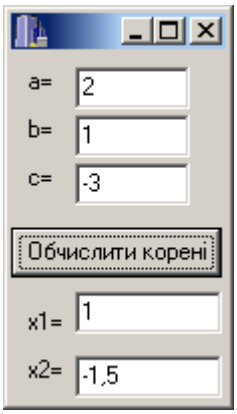
Приклад 4.15 Створити програмний проект для обчислення коренів квадратного рівняння $ax^2+bx+c=0$. Передбачити перевірку наявності можливих коренів, тобто перевірку, чи є дискримінант додатним.





Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
   {double a, b, c, D, x1, x2;
    a=StrToFloat(Edit1->Text);
    b=StrToFloat(Edit2->Text);
    c=StrToFloat(Edit3->Text);
    D=b*b-4*a*c;
    if(D<0)
      { Edit4->Clear(); Edit5->Clear();
        ShowMessage("Коренів немає!");
      }
    else
      { x1=(-b+sqrt(D))/(2*a);
        x2=(-b-sqrt(D))/(2*a);
        Edit4->Text=FloatToStr(x1);
        Edit5->Text=FloatToStr(x2);
      }
   }
```



Приклад 4.16 Створити програмний проект для визначення координат та координатної чверті (квадранта) точки, яку задаватиме користувач.

Розв'язок. Посеред форми розташуємо два компоненти `Bevel` (закладка `Additional`) як координатні осі, для яких слід змінити значення властивості `Shape` на значення `bsBottomLine` та `bsRightLine`, щоби вони набули вигляду горизонтальної та вертикальної ліній. За допомогою компонента `StatusBar1` (закладка `Win32`) внизу форми у першій панелі цього компонента виводитимуться динамічно змінювані координати переміщення курсора. Точку користувач зможе поставити, просто клацнувши мишею. При цьому на другій панелі компонента `StatusBar1` виводитимуться її координати та квадрант. Зображення точки задамо компонентом `Shape1`, встановивши для властивості `Shape` значення `stCircle`, а її розмір у 10 пікселів – властивостями `Width` та `Height`.

Текст програми:

```
int x, y, Xt, Yt;       // Координати курсура і точки
//----------- Подія форми на пересування миші -----------------
void __fastcall TForm1::FormMouseMove(TObject *Sender,
                          TShiftState Shift, int X, int Y)
{ // Координати пов'язуємо з центром форми
  x=X-Form1->Width/2;   y=Form1->Height/2-Y;
  // Виведення динамічно змінюваних координат переміщення курсора
  StatusBar1->Panels->Items[0]->Text=" X="+IntToStr(x)+
                                     "  Y="+IntToStr(y);
  Xt=X; Yt=Y;
}
//----------- Подія форми на клацання миші --------------------
void __fastcall TForm1::FormClick(TObject *Sender)
{ Shape1->Left=Xt - Shape1->Width/2;// Горизонтальна координата точки
  Shape1->Top=Yt - Shape1->Height/2;// Вертикальна координата точки
  Shape1->Visible = 1;              // Увімкнення видимості точки
  String s;
  if(x>0 && y>0) s="1 чверть";      // Визначення квадранта точки
  if(x<0 && y>0) s="2 чверть";
  if(x<0 && y<0) s="3 чверть";
  if(x>0 && y<0) s="4 чверть";
  // Виведення координат точки у другій панелі компонента StatusBar1
  StatusBar1->Panels->Items[1]->Text="Координати точки:
                X="+IntToStr(x)+"  Y="+IntToStr(y)+"  "+s;
}
```

Приклад 4.17 Обчислити силу струму I на ділянці електричного ланцюга довжиною L за формулою

$$I=\begin{cases}\sin^2(LT)-\cos^2\left(\dfrac{L}{T}\right) & \text{за } 0\le L\le\dfrac{1}{3};\\ \sin^2\left(\dfrac{T}{L}\right)+\cos^2\left(\dfrac{1}{TL}\right) & \text{за } \dfrac{1}{3}<L\le\dfrac{2}{3};\\ e^{LT}\left(\sin^2\left(\dfrac{1}{TL}\right)-\cos^2(LT)\right) & \text{за } \dfrac{2}{3}<L\le 1.\end{cases}$$

Значення періоду T та довжини L ввести з клавіатури.

Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float I, T=StrToFloat(Edit1->Text), L=StrToFloat(Edit2->Text);
 // Обчислення сили струму за умовою
 if ((L>=0)&&(L<=1./3)) I=sin(L*T)-cos(L/T);
 else
  if (L<=2./3) I=sin(T/L)+cos(1./(T*L));
  else
   if (L<=1) I=exp(L*T)*(pow(sin(1./(T*L)),2)-pow(cos(L*T),2));
 Edit3->Text = FloatToStr(I); // Виведення сили струму I
}
```

Приклади консольних програм з умовним оператором

Приклад 4.18 Увести значення року та перевірити, чи є він високосним.

Текст програми:

```
#include <vcl.h>
#pragma hdrstop
#include <iostream.h>
#include <conio.h>
#pragma argsused
//---------------------------------------------------------
 int main(int argc, char* argv[])
{ int year, r;
  cout << " Уведіть рік= " ;
  cin >> year;
  r = year % 4; // Остача від ділення year на 4
  if( r ) cout << " Рік не високосний "  ;
  else  cout << " Рік високосний "  ;
  getch();   return 0;
}
```

Результати виконання програми:

```
Уведіть рік= 2008
 Рік високосний

Уведіть рік= 2010
 Рік не високосний
```

Приклад 4.19 Увести ненульове ціле число і визначити його знак (вивести “Додатне” чи “Від’ємне”).

Текст програми:

```
#include <iostream.h>
#include <conio.h>
int main(int argc, char* argv[])
{  int x;
   cout << "Увести ненульове число:" ;
   cin >> x;
   if(x>0) cout << "Додатне" << endl;
   else  cout << "Від’ємне" << endl;
   getch();   return 0;
}
```

Результати виконання програми:

```
Увести число: 5
Додатне

Увести число: -15
Від’ємне
```

Приклад 4.20 Написати програму для перевірки знання року заснування Одеси. У разі помилкової відповіді користувача програма має вивести правильну відповідь.

Текст програми:

```
#include <iostream.h>
#include <conio.h>
int main(int argc, char* argv[])
{ int year;
  cout << "Уведіть рік заснування Одеси= " ;
  cin >> year ;
  if(year==1794) cout << "Відповідь є правильна "<<endl;
   else cout<<"Ви помилились. Одесу було засновано 1794 року";
  getch();
  return 0;
}
```

Результати виконання програми:

```
Уведіть рік заснування Одеси=1790
Ви помилились. Одесу було засновано 1794 року

Уведіть рік заснування Одеси=1794
Відповідь є правильна
```

Приклад 4.21 Ввести число і обчислити, якщо це можливо, значення виразу $y = \frac{3}{\sqrt{x-7}-2}$.

Розв'язок. Обчислити значення цього виразу можливо, якщо, по-перше, під коренем розташовано невід'ємне число і, по-друге, знаменник не дорівнює 0, тобто якщо виконуються дві умови: $x-7 \geq 0$ та $\sqrt{x-7}-2 \neq 0$.

Текст програми:

```
#include <vcl.h>
#pragma hdrstop
#include <iostream.h>
#include <conio.h>
#pragma argsused
int main(int argc, char* argv[])
{ float x, y;
  cout << "Уведіть число:" ;
  cin >> x;
  if(x - 7 >= 0  &&  sqrt(x - 7) - 2 != 0)
    { y = 3/(sqrt(x - 7) - 2);
      cout << " y= " << y << endl;
    }
   else  cout<<"Обчислити неможливо"<<endl;
   getch();
   return 0;
}
```

Результати виконання програми:

```
Уведіть число: 15
y=3.62132

Уведіть число: 5
Обчислити неможливо
```

Приклад 4.22 Написати програму для перевірки знання таблиці множення, тобто множення двох випадкових одноцифрових чисел.

Розв'язок. Після кожної відповіді відбувається перевірка її правильності та виводиться відповідне повідомлення.

Текст основної програми:

```
#include <iostream.h>
#include <conio.h>
int main(int argc, char* argv[])
{ int x,y,p,ans;// Співмножники, добуток та відповідь
  time_t t;      // Поточний час для ініціалізації  генератора випадкових чисел
  // Ініціалізація  генератора випадкових чисел (див. докладніше підрозд. 3.7)
  srand((unsigned)time(&t));         // або  randomize();
  x=rand()%10+1;       y=rand()%10+1;
  p=x*y;
  cout << "Скільки буде  " << x <<" x "<< y << endl;
  cin >> ans ;
  if(p==ans)cout<<"Відповідь є правильна"<<endl;
  else cout << "Ви помилились. "<< x <<" x "<< y << " = " << p;
  getch();
  return 0;
}
```

Результати виконання програми:

```
Скільки буде  6 x 9
56
Ви помилились.  6 x 9 = 54

Скільки буде  6 x 3
18
Відповідь є правильна
```

4.3.7 Тернарна умовна операція ?:

У мові C++ є одна тернарна операція, яка має три операнди, це – умовна операція, котра має такий формат:

<умова> ? *< операнд1>* : *< операнд2>*;

Якщо *умова* має ненульове значення, результатом буде значення *операнда1,* інакше – значення *операнда2*. Зауважимо, що обчислюється лише один з операндів, а не обидва. Наприклад:

```
j = (i < 0) ? (–i) : (i);
```

У результаті j набуде абсолютного значення i, оскільки, якщо i є менше за 0, то j присвоїться –i, а якщо i є більше чи дорівнює нулю, то j присвоїться i.

Для визначення більшого з двох чисел – x та y – слід записати оператор:

```
max = (x > y) ? x : y;
```

Тип результату залежить від типів *операнда1* і *операнда2*:

✓ якщо *операнд1* чи *операнд2* має цілий чи дійсний тип (зауважимо, що їхні типи можуть відрізнятися), то виконуються звичайні арифметичні перетворення. Типом результату є тип операнда після перетворення;

✓ якщо *операнд1* чи *операнд2* має один і той самий тип структури, об'єднання чи вказівника, то тип результату буде тим самим типом структури, об'єднання чи вказівника;

✓ якщо обидва операнди мають тип void, то результат матиме тип void;

✓ якщо один операнд є вказівником на об'єкт будь-якого типу, а другий операнд є вказівником на void, то вказівник на об'єкт перетворюється на вказівник на void, котрий і буде типом результату;

✓ якщо один операнд є вказівником, а другий – константним виразом зі значенням 0, то типом результату буде тип вказівника.

Умовна операція ?: за своєю дією є схожою до оператора if-else, однак слід зважати, що вона використовується лише у виразах і передає значення одного з операндів. Наприклад, оператор

```
if( z ) res = x;  else  res = y;
```

можна записати за допомогою умовної операції так

```
z ? (res = x) : (res = y) ;
```

або ясніше для сприйняття

```
res = z ? x : y ;
```

Окрім того, зауважимо, що таку конструкцію можна записати і за допомогою логічних операцій && та ||

```
z && (res = x);
z || (res = y);
```

але таку заміну більшість людей сприймають важко і неоднозначно.

Розглянемо на прикладі, як застосовувати умовну операцію ?:, і побачимо, що її результат є ясніший для сприйняття, аніж рівноцінний оператор if-else.

Приклад 4.23 Увести два числа і обрати з них більше.

Текст програмного коду:

```
#include <iostream.h>
#include <conio.h>
int main()
{ int x, y, z;                    // Оголошуємо три змінні
  cout << "Enter two numbers:\n";
  cin >> x >> y;                  // Вводимо з клавіатури два числа
  if(x>y) z=x; // Якщо перше число є більше за друге, присвоюємо його змінній z,
  else    z=y; // інакше – присвоюємо змінній z друге число
```

```
  cout << "\n \nResalt 1: " << z;   // Виводимо z
  z =  (x > y) ? x : y;          // Тернарна операція
  cout << "\nResalt 2: " << z << "\n";
  getch();
  return 0;
}
```

Результати роботи консольного додатка:

```
Enter two numbers.
15  8

Resalt 1: 15
Resalt 2: 15
```

Наведена у програмі тернарна операція читається як: “Якщо `x` більше за `y`, повернути значення `x`. Інакше – повернути значення `y`”. Значення, яке повертається, присвоюється змінній `z` і виводиться на екран. Вочевидь, що тут умовна операція є скороченим еквівалентом команди `if-else`.

Однак бувають ситуації, коли операцію `?:` неможна замінити конструкцією `if-then-else`, наприклад, якщо організувати попередню програму так:

```
#include <iostream.h>
#include <conio.h>
int main()
{ int x, y;                       // Оголошуємо дві змінні
  cout << "Enter two numbers:\n";
  cin >> x >> y;                  // Вводимо з клавіатури два числа
  int z =  (x > y) ? x : y;       // Тернарна операція
  cout << "\n max = " << z << "\n";
  getch();
  return 0;
}
```

Тут змінна `z` ініціалізується на момент оголошення результатом роботи тернарної операції. Подібного ефекту не вдалося б досягти простим привласненням в тому чи іншому випадку.

4.3.8 Оператор вибору варіантів switch

Формат оператора вибору варіантів `switch`:

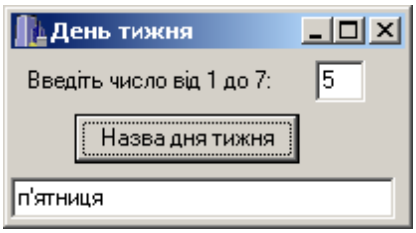
```
switch (<вираз>)
{ case <значення-мітка_1> : {<послідовність операторів>; break;}
  . . . . . .
  case <значення-мітка_n> : {<послідовність операторів>; break;}
  [ default:  <послідовність операторів>; ]
}
```

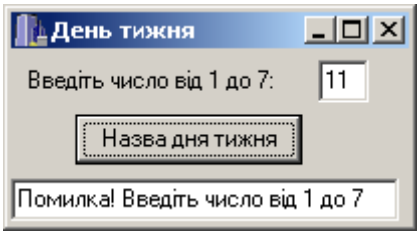
Спочатку обчислюється *вираз* у дужках. *Вираз* повинен мати цілий чи то символьний тип. Значення *виразу* порівнюється зі значеннями *міток* після клю-

чових слів case. Якщо значення *виразу* збіглося зі значенням якоїсь *мітки*, то виконується відповідна *послідовність операторів*, позначена цією *міткою* і записана після двокрапки, допоки не зустрінеться оператор break. Якщо значення *виразу* не збіглося з жодною *міткою*, то виконуються оператори, які слідують за ключовим словом default. Мітка default є необов'язковою конструкцією оператора switch, на що вказують квадратні дужки [] у форматі. Оператор break здійснює вихід із switch. Якщо оператор break є відсутній наприкінці операторів відповідного case, то буде почергово виконано всі оператори до наступного break чи то до кінця switch для всіх гілок case незалежно від значення їхніх міток.

Розглянемо роботу оператора switch на прикладі функції, яка за числом виводить назву дня тижня.

```
//------------ Кнопка "Назва дня тижня"-------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString s;
  int n=StrToInt(Edit1->Text);
  switch (n)
   { case 1: {s="понеділок"; break; }
     case 2: {s="вівторок";  break; }
     case 3: {s="середа";    break; }
     case 4: {s="четвер";    break; }
     case 5: {s="п'ятниця";  break; }
     case 6: {s="субота";    break; }
     case 7: {s="неділя";    break; }
     default:
       s="Помилка! Введіть число від 1 до 7";
   }
  Edit2->Text = s;
}
```





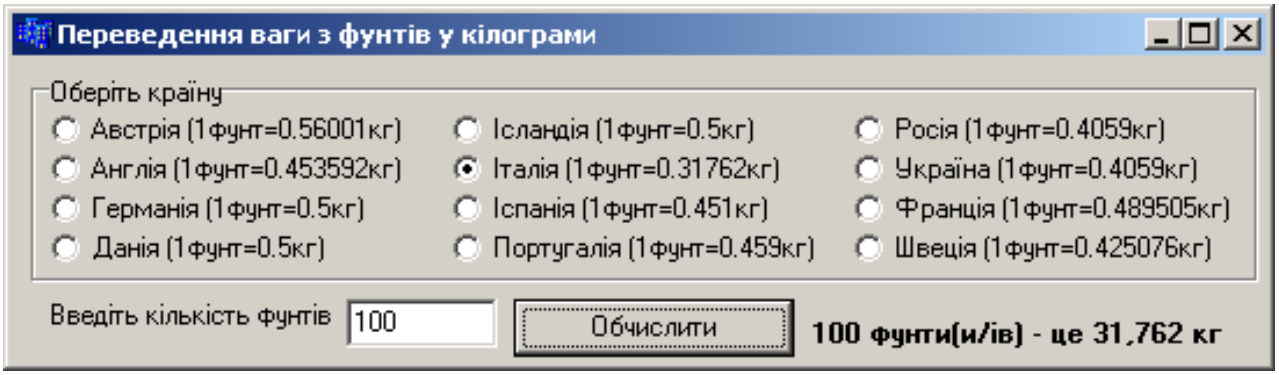
Приклади створювання проектів програм з розгалуженою структурою, організованих за допомогою оператора switch

Приклад 4.24 Написати програму для переведення ваги з фунтів у кілограми, зважаючи на те, що у різних країнах фунт має різні значення.

Текст програми:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{ RadioGroup1->Columns=3;  // кількість колонок у RadioGroup1
  // Формування елементів RadioGroup1 при створенні форми
  RadioGroup1->Items->Add("Австрія (1фунт=0.56001кг)");
  RadioGroup1->Items->Add("Англія (1фунт=0.453592кг)");
  RadioGroup1->Items->Add("Германія (1фунт=0.5кг)");
  RadioGroup1->Items->Add("Данія (1фунт=0.5кг)");
  RadioGroup1->Items->Add("Ісландія (1фунт=0.5кг)");
  RadioGroup1->Items->Add("Італія (1фунт=0.31762кг)");
```

```
  RadioGroup1->Items->Add("Іспанія (1фунт=0.451кг)");
  RadioGroup1->Items->Add("Португалія (1фунт=0.459кг)");
  RadioGroup1->Items->Add("Росія (1фунт=0.4059кг)");
  RadioGroup1->Items->Add("Україна (1фунт=0.4059кг)");
  RadioGroup1->Items->Add("Франція (1фунт=0.489505кг)");
  RadioGroup1->Items->Add("Швеція (1фунт=0.425076кг)");
  RadioGroup1->ItemIndex=0; // Детальніше про компонент див. стор. 106
}
//------------ Кнопка "Обчислити" --------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float  funt, kg, k; // Вага у фунтах, кілограмах і коефіцієнт переведення
  switch (RadioGroup1->ItemIndex)
  { case 0: k=0.56001; break;     // Австрія
    case 1: k=0.453592;break;     // Англія
    case 2:                       // Германія
    case 3:                       // Данія
    case 4: k=0.5;     break;     // Ісландія
    case 5: k=0.31762; break;     // Італія
    case 6: k=0.451;   break;     // Іспанія
    case 7: k=0.459;   break;     // Португалія
    case 8:                       // Росія
    case 9: k=0.4059;  break;     // Україна
    case 10: k=0.489505;  break; // Франція
    case 11: k=0.425076;  break; // Швеція
  }
  funt=StrToFloat(Edit1->Text);
  kg=k*funt;
  Label1->Caption = Edit1->Text + " фунти(и/ів) - це " +
                    FloatToStrF(kg, ffFixed, 6, 3) + " кг";
}
```



Приклад 4.25 Обчислити значення функції $L=\begin{cases} x^{2+a}+1 & \text{за } k=1; \\ (x+1)/a & \text{за } k=2; \\ \mathrm{e}^{x-a}-a^{x} & \text{за } k=3; \\ \lg|x-a| & \text{за } k=4 \end{cases}$

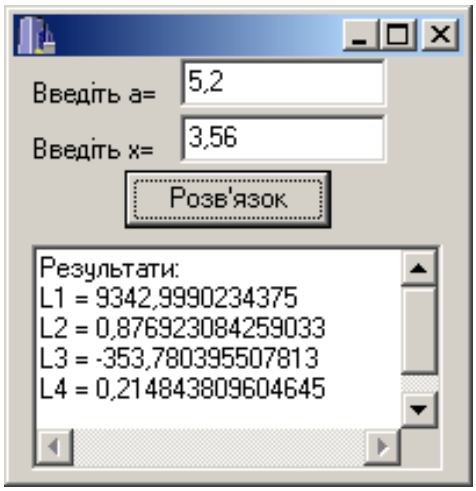
для всіх значень параметра k.

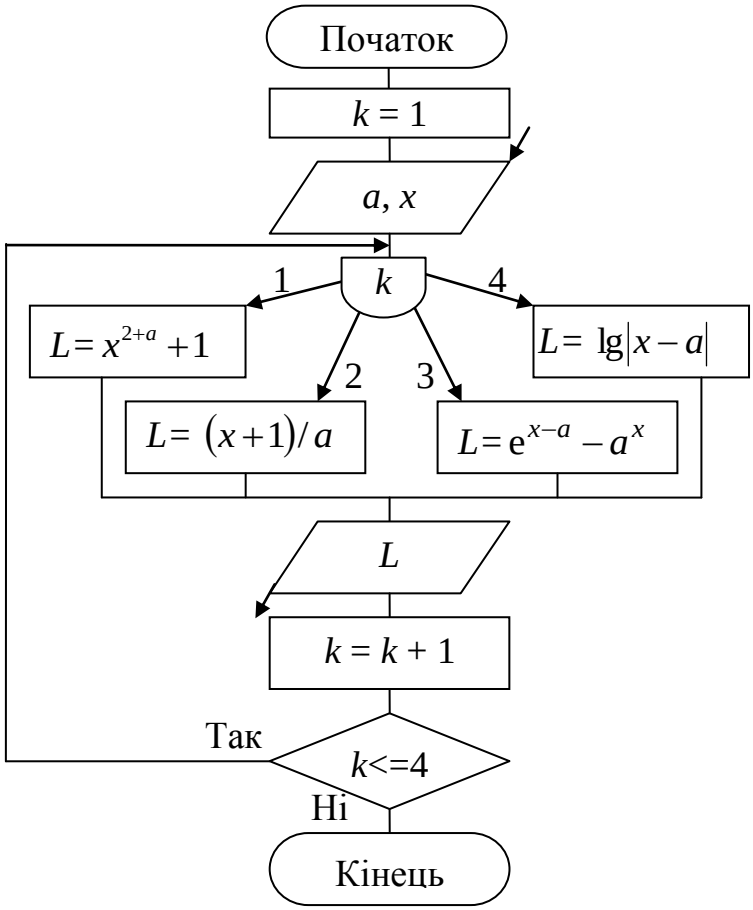
Виведення всіх чотирьох значень функції L організуємо до компонента `Memo` – багаторядкове текстове вікно для введення чи виведення значень даних програми. У вікні компонентів C++ Builder він має позначення

Основні властивості компонента `Memo`:

- ✓ `Name` – ім'я компонента у програмі;
- ✓ `Lines` – вікно для введення чи редагування початкових даних програми;
- ✓ `Lines->Count` – кількість заповнених рядків;
- ✓ `ScrollBars` – лінійки прокручування вікна.

Розв'язок. Робочий вигляд форми та схема алгоритму:





Текст програми:

```
#include <math.h>
void __fastcall TForm1::Button1Click(TObject *Sender)
{ Memo1->Clear();    Memo1->Lines->Add("Результати:");
  float x, a, L;     int k = 1;
  a = StrToFloat(Edit1->Text);
  x = StrToFloat(Edit2->Text);
 M1:  switch (k)
      { case 1:  L = pow(x, 2 + a) + 1;  break;
        case 2:  L = (x + 1) / a;        break;
        case 3:  L = exp(x - a) - pow(a, x); break;
        case 4:  L = log10(fabs(x - a)); break;
      }
  Memo1->Lines->Add("L" + IntToStr(k) + " = " + FloatToStr(L));
  k++;
  if(k<=4) goto M1;
}
```

Використання оператора

```
if(k<=4) goto M1;
```

у цій програмі організовує чотириразове повторення виконання групи операторів, починаючи від мітки `M1`, що фактично утворює псевдоцикл.

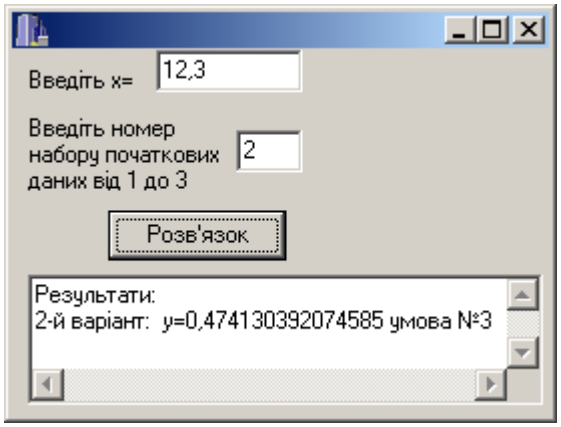
Приклад 4.26 Увести значення x та обчислити значення

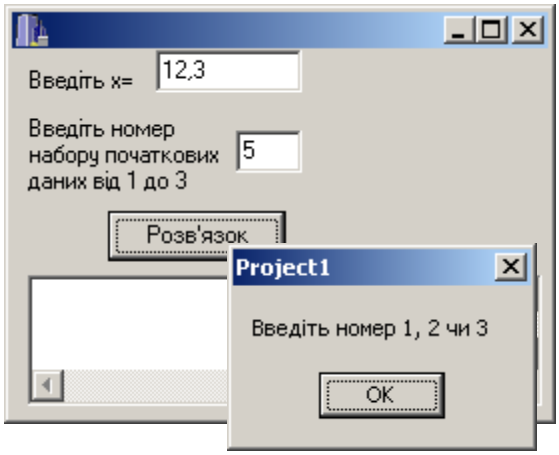
$$y=\begin{cases}\sin\left(e^{|a+c|}\right)+x^{2+b} & \text{за} \quad x<a;\\ \sqrt[3]{a+|5.3b|}+c & \text{за} \quad a\le x<b;\\ \cos^2(a+b)+\sin x^2 & \text{за} \quad x>b\end{cases}$$

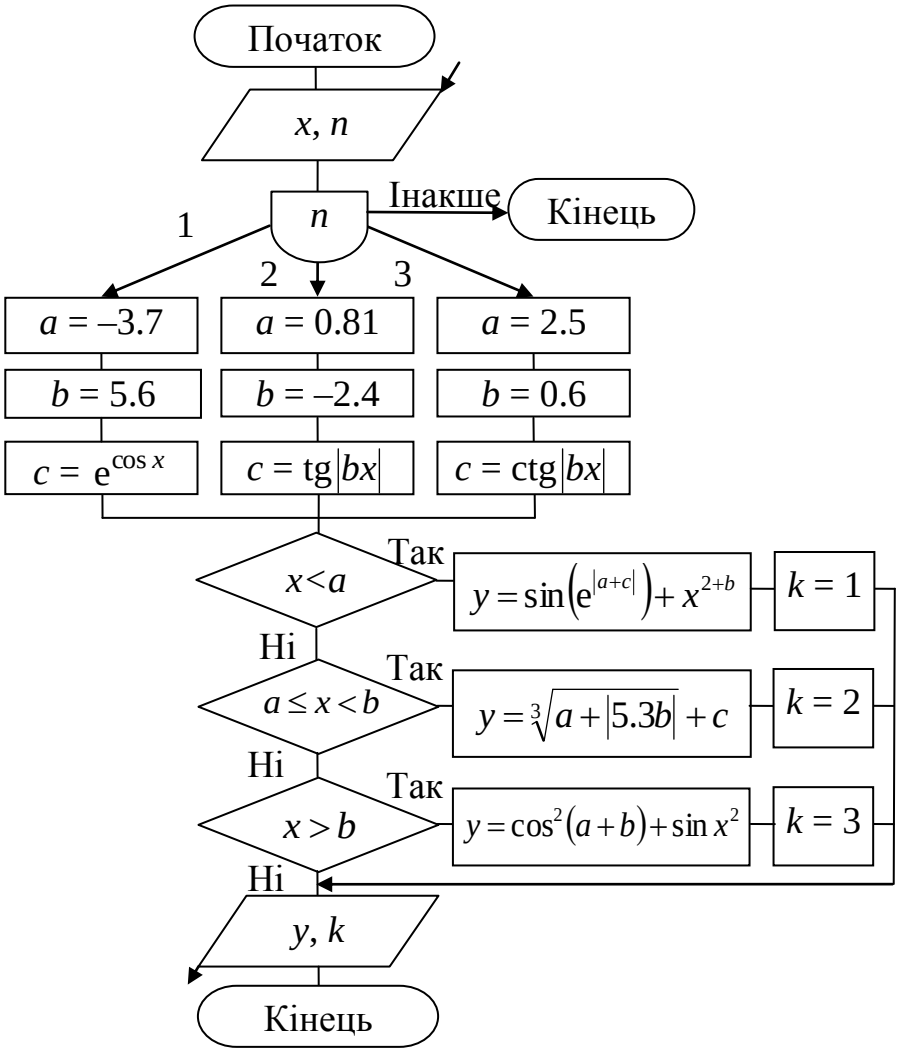
для одного з трьох варіантів параметрів:

1) $a = -3{,}7;\ b = 5{,}6;\ c = e^{\cos x}$;
2) $a = 0{,}81;\ b = -2{,}4;\ c = \operatorname{tg}|bx|$;
3) $a = 2{,}5;\ b = 0{,}6;\ c = \operatorname{ctg}|bx|$.

Розв'язок. Робочий вигляд форми і схема алгоритму:







Текст програми:

```
#include <math.h>
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float x, y, a, b, c;
  x = StrToFloat(Edit1->Text);
  int k, n = StrToInt(Edit2->Text);
  Memo1->Clear();   Memo1->Lines->Add("Результати:");
  switch (n)
  { case 1: { a=-3.7; b=5.6;  c=exp(cos(x)); break;}
    case 2: { a=0.81; b=-2.4; c=tan(fabs(b*x)); break;}
    case 3: { a=2.5;  b=0.6;  c=1/tan(fabs(b*x)); break;}
    default:{ Memo1->Clear();
              ShowMessage("Введіть номер 1, 2 чи 3"); return; }
  }
  if(x<a)   { y=sin(exp(fabs(a+c)))+pow(x,2+b); k=1;}
  if(x>=a && x<b) { y=pow(a+fabs(5.3*b),1./3)+c; k=2;}
  if(x>=b) { y=pow(cos(a+b),2)+sin(x*x); k=3;}
  Memo1->Lines->Add(IntToStr(n)+"-й варіант:  y=" +
                          FloatToStr(y)+" умова №"+IntToStr(k));
}
```

Приклад 4.27 Визначити за введеною датою день тижня, на який ця дата припадає. Для обчислення користуватися формулою

$$\left(d+\left[\frac{13m-1}{5}\right]+y+\left[\frac{y}{4}\right]+\left[\frac{c}{4}\right]-2c+777\right)\bmod 7,$$

де d – день місяця;

m – номер місяця, починаючи рахунок від березня, як це робили у Стародавньому Римі (березень – 1, квітень – 2 , …, лютий – 12);

y – рік у столітті;

c – кількість століть.

Квадратні дужки у формулі означають, що слід взяти лише цілу частину від значення у дужках. Обчислене за формулою значення визначає день тижня: 1 – понеділок, 2 – вівторок, …, 6 – субота, 0 – неділя.

Текст програми для консольного додатка:

```
#include <vcl.h>
#pragma hdrstop
#include <iostream.h>
#include <conio.h>
//---------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{int day,month,year; // день, місяць, рік
 int c, y;           // сторіччя і рік у сторіччі
 int m;              // місяць за римським календарем
 int d;              // день тижня
 puts("Введіть дату: день місяць рік.");
 puts("Наприклад, 24 04 2009 ");
 puts(" ->");
 scanf("%i %i %i",&day,&month,&year);
 if(month==1||month==2)year--;//січень і лютий належать до минулого року
 m = month-2;                 //рік розпочинається з березня
 if(m<=0)m+=12;               //для січня і лютого
 c = year/100;
 y = year - c*100;
 d = (day+(13*m-1)/5+y+y/4+c/4-2*c+777)%7;
 switch(d)
 { case 1: puts("Понеділок 1"); break;
   case 2: puts("Вівторок 2"); break;
   case 3: puts("Середа 3"); break;
   case 4: puts("Четвер 4"); break;
   case 5: puts("П'ятниця 5"); break;
   case 6: puts("Субота 6"); break;
   case 0: puts("Неділя 7");
 }
 printf("\n Для завершення натисніть <Enter> \n");
 getch();  return 0;
}
```

4.4 Програмування циклічних алгоритмів

4.4.1 Циклічні алгоритми

Обчислювальний процес називається *циклічним*, якщо він неодноразово повторюється, допоки виконується певна задана умова. Блок повторюваних операторів називають *тілом* циклу. Існують три різновиди операторів циклу:

- ✓ оператор циклу **for**;
- ✓ оператор циклу з передумовою **while**;
- ✓ оператор циклу з післяумовою **do-while**.

Всі оператори циклу неодмінно містять такі складові частини:

- ✓ присвоювання початкових значень (ініціалізація);
- ✓ умова продовження циклу;
- ✓ тіло циклу;
- ✓ зміна параметра циклу.

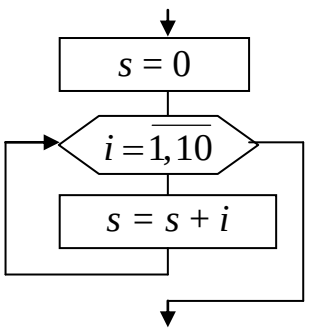
4.4.2 Оператор циклу з параметром for

Цей оператор зазвичай використовується, коли є заздалегідь відома кількість повторювань, або коли умова продовження виконання циклу записується коротким виразом. За приклади використання даного оператора можуть слугувати обчислення сум заданої кількості доданків, пошук мінімального (максимального) елемента послідовності чисел, зчитування рядків `Memo`, сортування елементів масиву за збільшенням (за зменшенням) тощо.

Синтаксис оператора:

for (<*ініціалізація*>; <*умова*>; <*модифікації*>) <*тіло циклу*>;

Конструктивно цей оператор складається з трьох основних блоків, розміщених у круглих дужках і відокремлених один від одного крапкою з комою (;), та команд (тіла циклу), які мають багаторазово виконуватись у цьому циклі. На початку виконання оператора циклу одноразово у блоці *ініціалізації* задаються початкові значення змінних (параметрів), які керують циклом. Потім перевіряється *умова* і, якщо вона виконується (`true` чи має ненульове значення), то виконується *команда* (чи група команд в операторних дужках {}) тіла циклу. *Модифікації* змінюють параметри циклу і, в разі істинності умови, виконування циклу триває. Якщо *умова* не виконується (`false` чи дорівнює нулю), відбувається вихід із циклу і керування передається на оператор, який слідує за оператором `for`. Суттєвим є те, що перевірка умови виконується на початку циклу. Це означає, що тіло циклу може не виконуватись жодного разу, якщо *умова* спочатку є хибна. Кожне повторення (крок) циклу називається *ітерацією*.



Простий приклад для обчислення суми $S = \sum_{i=1}^{10} i$ проілюструє використання оператора `for`:

```
int  s = 0;
for(int i = 1; i <= 10; i++)  s += i;
```

Цей оператор можна прочитати так: “виконати команду `s += i` 10 разів (для значень `i` від 1 до 10 включно, де `i` на кожній ітерації збільшується на 1)”. У наведеному прикладі є два присвоювання початкових значень: `s=0` і `i=1`, умова продовження циклу: (`i<=10`) і змінення параметра: `i++`. Тілом циклу є команда `s += i`.

Порядок виконання цього циклу комп’ютером є такий:

1) присвоюються початкові значення (`s=0`, `i=1`);

2) перевіряється умова (`i<=10`);

3) якщо умова є істинна (`true`), виконується команда (чи команди) тіла циклу: до суми, обчисленої на попередній ітерації, додається нове число;

4) параметр циклу збільшується на 1.

Далі повертаємось до пункту 2. Якщо умову у пункті 2 не буде виконано (`false`), станеться вихід із циклу.

В операторі можливі конструкції, коли є відсутній той чи інший блок: *ініціалізація* може бути відсутня, якщо початкове значення задати попередньо; *умова* – якщо припускається, що умова є завжди істинна, тобто слід неодмінно виконувати тіло циклу, допоки не зустрінеться оператор `break`; а *модифікації* – якщо приріст параметра здійснювати в тілі циклу чи взагалі це є непотрібне. Тоді сам вираз блока пропускається, але крапка з комою (;) неодмінно має залишитись. Можливою є наявність *порожнього* оператора (оператор є відсутній) у тілі циклу. Приміром, суму з попереднього прикладу можна обчислити в інший спосіб:

```
for(int s = 0,  i = 1;  i <= 10;  s += i++);
```

У цьому прикладі є відсутній *оператор*, а блок *ініціалізації* вмістить два оператори, розділених операцією “кома”, які задають початкові значення змінних `s` та `i`.

Розглянемо приклади циклів `for`.

1) Виведення до компонента `Memo` чисел від 1 до 10 з їхніми квадратами:

```
for(int i=1; i<11; i++)
  Memo1->Lines->Add(IntToStr(i)+ "  " + IntToStr(i*i));
```

Цикл виконується за `i` = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Після цього `i` набуде значення 11, умова (11<11) не виконується (`false`) і відбувається вихід з циклу.

2) Виведення додатних непарних цілих від 1 до 100 з їхніми квадратами. Цикл записується аналогічно до попереднього, але параметр збільшуватиметься на 2, тобто цикл виконуватиметься для значень `i` = 1, 3, 5, 7, …, 97, 99.

```
for(int i=1; i<100; i+=2)
  Memo1->Lines->Add(IntToStr(i)+ "  " + IntToStr(i*i));
```

3) Обчислення факторіала F=n! (нагадаємо, що факторіал обчислюється за формулою n! = 1·2·3...(n–2)(n–1)n, наприклад: 4! = 1·2·3·4 = 24). Наведемо три аналогічні за дією форми запису оператора `for`:

1) `int F=1, n=5; for(int i=1; i<=n; i++) F *= i;`

2) `int F, i, n=5; for(F=1, i=1; i<=n; F *= i++);`

3) `int F=1, i=1, n=5; for( ; i<=n; ) F *= i++;`

Для завчасного початку чергової ітерації циклу можна використати оператор переходу до наступної ітерації **continue** (див. п. 4.4.5), наприклад:

```
for(i = 0; i < 20; i++)
 { if(array[i] == 0)  continue;
   array[i] = 1/array[i]; }
```

Для завчасного виходу з циклу застосовують оператори **break** (вихід з конструкції) (див. п. 4.4.5), **goto** (безумовний перехід) (див. п. 4.3.2) чи **return** (вихід з поточної функції) (див. підрозд. 8.1).

Деякі варіанти застосування оператора `for` підвищують його гнучкість за рахунок можливості використання декількох змінних-параметрів циклу.

Наприклад:

```
int main ( )
{ int top, bot;
  char string[100], temp;
  for ( top=0, bot=98 ; top<bot ; top++, bot--)
  { temp = string[top];
    string[top] = string[bot];
    string[bot] = temp;
  }
  return 0;
}
```

У цьому прикладі для реалізації записування рядка символів у зворотному порядку параметрами циклу є дві змінні `top` та `bot`, значення яких рухаються назустріч один одному. Зауважимо, що у блоці ініціалізації оператора `for` задаються через кому початкові значення відразу обох змінних (параметрів). Так само через кому записано модифікацію цих змінних після умовного виразу.

Наведемо ще один варіант оператор циклу `for`, в якому задається початкове значення параметра `i`, його модифікація, проте в умовному виразі ця змінна фігурує лише як індекс елемента масиву:

```
for (i=0; t[i]=>0; i++) ;
```

У цьому операторі параметр циклу `i` набуде значення номера першого у масиві від'ємного елемента масиву `t`. Отже, такий цикл може використовуватись, приміром, для пошуку елементів за певним значенням.

Іншим цікавим варіантом застосування оператора `for` є нескінчений цикл. Для організації такого циклу можна записати пустий умовний вираз, а для виходу з циклу скористатись умовним оператором `if` разом з оператором `break`.

Наприклад:

```
for ( ; ; )
{ . . .
  if ( <деяка умова> ) break;
  . . .
}
```

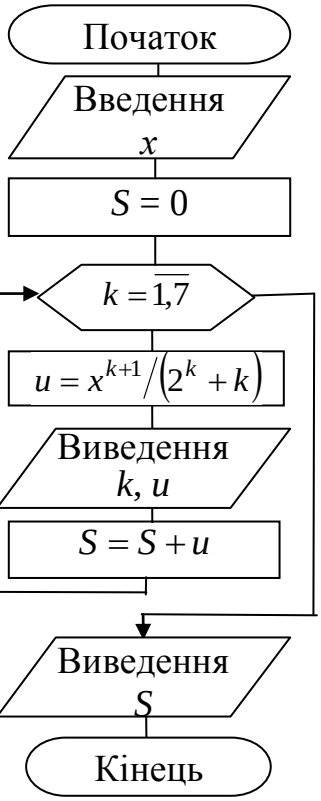
Розглянемо кілька прикладів розв'язування задач, в яких є доцільне використання оператора циклу `for`. Для усіх програм розроблено алгоритмічні схе-

ми (блок-схеми), які визначають порядок виконування дій, і наведено форми з результатами обчислень.

Приклади проектів програм з оператором циклу for в C++ Builder

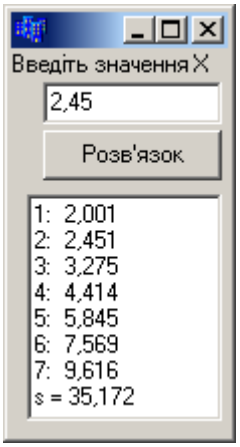
Приклад 4.28 Обчислити суму $s = \sum_{k=1}^{7} \frac{x^{k+1}}{2^k + k}$, значення x ввести з екрана.

Розв'язок. Щоби впевнитися, що сума обчислюватиметься з усіх семи доданків, спочатку виведемо кожен з них, після чого виведемо вже саму суму.



Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float u, s = 0, x;
  x = StrToFloat(Edit1->Text);
  Memo1->Clear();
  for(int k = 1; k <= 7; k++)
  { u = pow(x, k+1)/(pow(2, k)+k);
    Memo1->Lines->Add(IntToStr(k)+
    ":  "+ FormatFloat("0.000", u));
    s = s + u;
  }
  Memo1->Lines->Add("s = " +
          FormatFloat("0.000", s));
}
```



Приклад 4.29 Обчислити добуток $z = \prod_{k=-2}^{4} \frac{(k+1)(k-5)}{k(k+6)}$. При виконуванні обчислень не слід враховувати ті множники, в яких чисельник чи знаменник дорівнюють нулю.

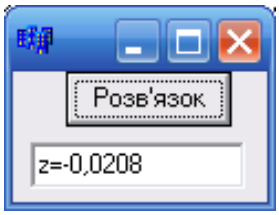
Розв'язок. При обчисленні добутку змінна `k` почергово набуває таких значень: –2, –1, …, 3, 4. За значення `k = -1` утворюється нуль у чисельнику, а отже, весь добуток перетворюється на нуль, а за `k = 0` – утворюється нуль у знаменнику. Тому, щоб вилучити ці множники, слід застосувати умовний оператор `if`, який можна організувати у два способи:

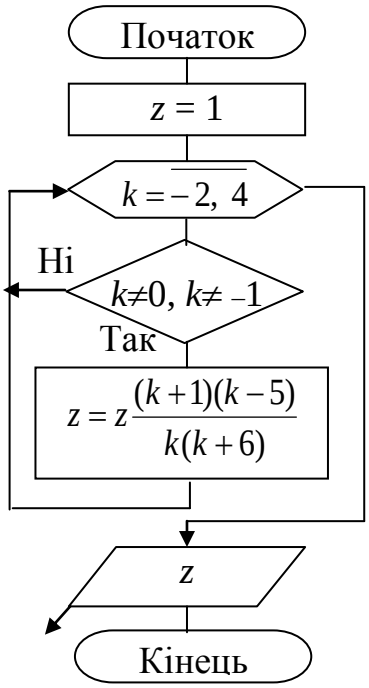
```
if(k != -1 && k != 0)
  z *= (float)(k+1)*(k-5)/(k*(k+6));
```

або

```
if(k==0 || k==-1) continue;
else z *= (float)(k+1)*(k-5)/(k*(k+6));
```

Умову `if(k!=0)` можна записати простіше, `if(k)`, оскільки ненульове значення розцінюється як `true`.





Текст програми:

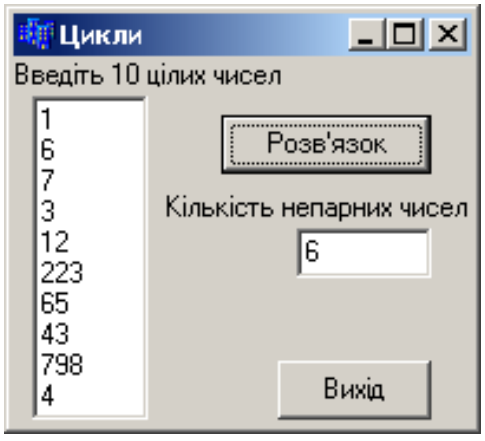
```
void __fastcall TForm1::Button1Click
                                (TObject *Sender)
{int k;
 float p, z = 1;
 for(k = -2; k <= 4; k++)
   if(k != -1 && k != 0)
     z *= (float)(k+1)*(k-5)/(k*(k+6));
 Edit1->Text = "z="+FormatFloat("0.0000",z);
}
```

Приклад 4.30 Увести десять цілих чисел і визначити кількість непарних.

Розв'язок. Уводитимемо числа до компонента `Memo`.

Текст програми:

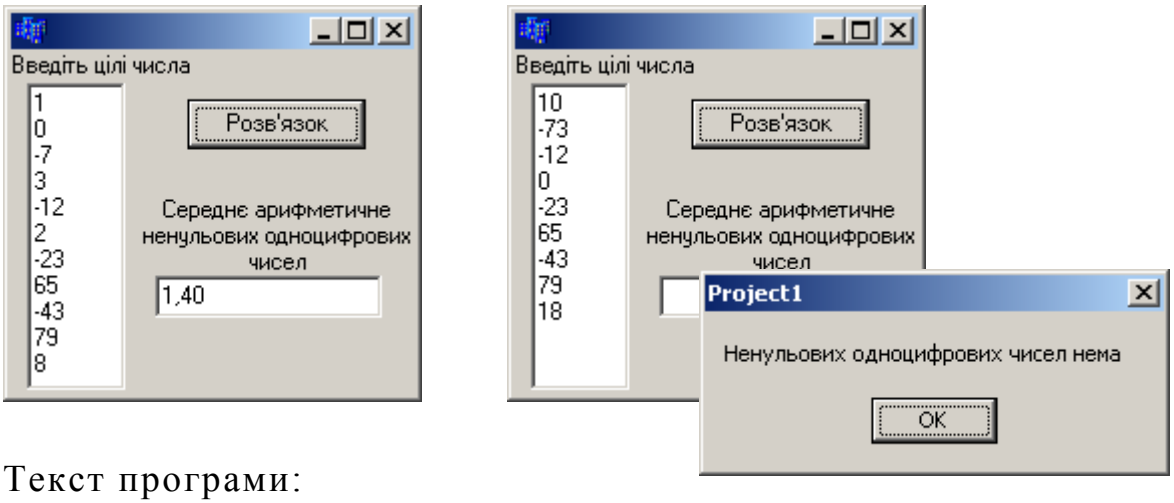
```
// Кнопка "Розв'язок".
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, x, k=0;
  int N=Memo1->Lines->Count;     // Кількість чисел у Memo1
  if(N<10).
   { ShowMessage("Не всі 10 чисел уведено"); return;    }
  for(i=0; i<10; i++)
  { x=StrToInt(Memo1->Lines->Strings[i]); // Зчитування чисел
    if(x%2==1) k++;               // і обчислення кількості непарних.
  }
  //Виведення кількості непарних чисел.
  Edit1->Text=IntToStr(k);
}
//-----------------------------
// Кнопка "Вихід".
void __fastcall TForm1:: Button2Click
                        (TObject *Sender)
{ Close();
}
```



Приклад 4.31 Увести цілі числа у Memo і обчислити середнє арифметичне ненульових одноцифрових чисел.

Розв'язок. Спочатку у циклі обчислимо суму та кількість ненульових одноцифрових чисел, а вже за циклом розділимо цю суму на кількість.

Для зчитування чисел з Memo використовуватимемо змінну x, для суми – змінну sum, для кількості – змінну k і для середнього арифметичного – aver.



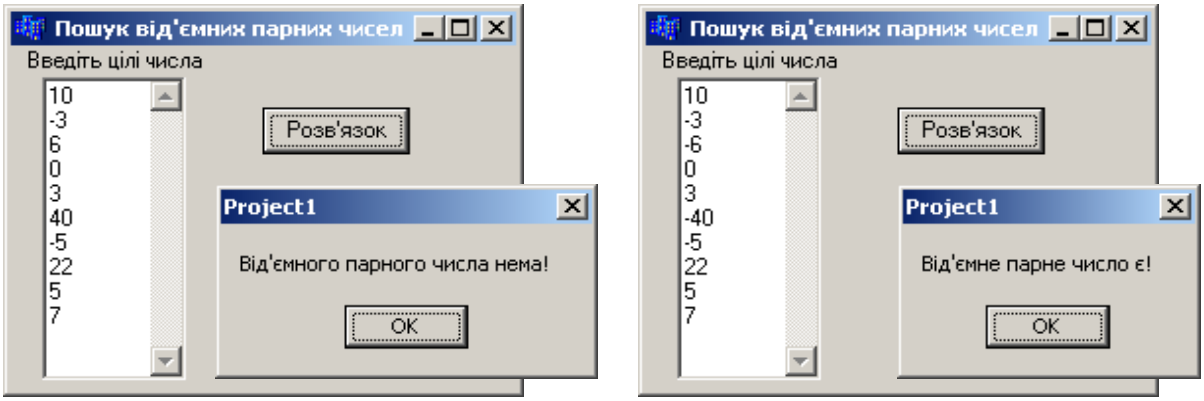
Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float aver, sum=0;   int x, k=0, i, n;
  n=Memo1->Lines->Count;          // n – кількість рядків у Memo1
  if(n==0)                 // Якщо чисел у Memo1 нема, припинити обчислення
   {ShowMessage ("Введіть числа в Memo1"); return;}
  for(i=0; i<n; i++)  // Повторити n разів (послідовно для кожного числа)
  {x=StrToInt(Memo1->Lines->Strings[i]); //Зчитати число з Memo1
   if(x>-10 && x<10 && x!=0)    // Якщо x – ненульове одноцифрове число,
   {sum+=x; k++; }     // додаємо його до суми sum і збільшуємо кількість на 1
  }
  if(k!=0)                 // Якщо такі числа є,
   { aver=sum/k;       // обчислити середнє арифметичне,
     Edit1->Text=FormatFloat("0.00",aver);
   }
  else                     // інакше – вивести повідомлення
   { Edit1->Clear();
     ShowMessage("Ненульових одноцифрових чисел нема");
}  }
```

Приклад 4.32 Увести цілі в Memo і визначити, чи є серед них від'ємні парні числа. Результат вивести у вигляді повідомлення.

Розв'язок. Для того щоб визначити, чи є в Memo від'ємні парні числа, можна обчислити їхню кількість, як це було зроблено у попередньому прикладі, і якщо ця кількість є більше за 0, можна стверджувати, що є як мінімум одне таке число, інакше – таких чисел немає. Однак, якщо відшукали перше від'ємне парне число, немає потреби у перевірці решти чисел з Memo1, можна одразу вивести повідомлення про результат і зупинити виконання програми за допомогою

команди `return`. Якщо жодного від'ємного парного числа не віднайдено, то по завершенні циклу виконуватиметься наступна команда, яка виведе повідомлення про відсутність шуканих елементів.



Текст програми:

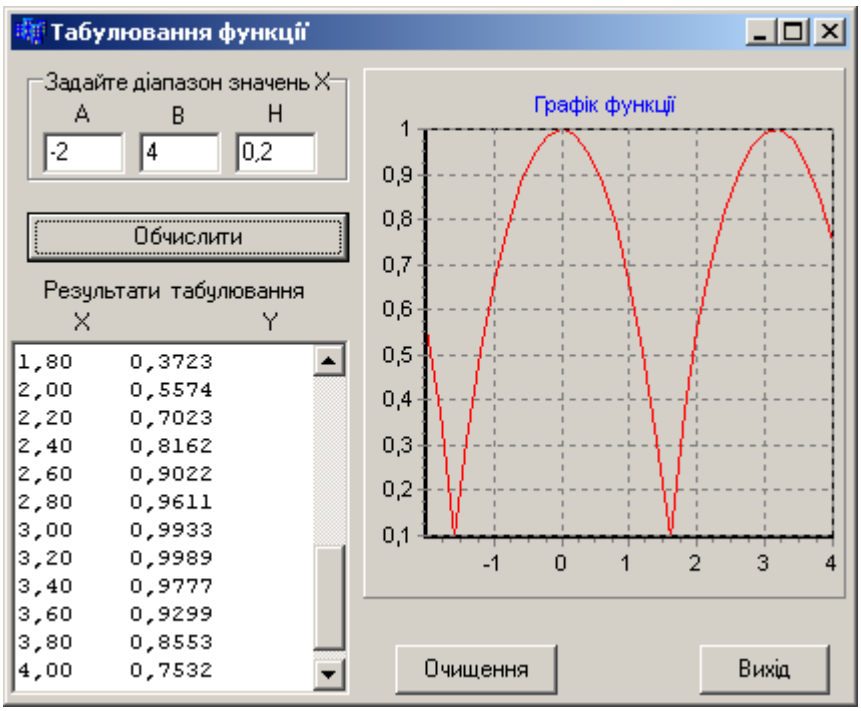
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int x, i, n=Memo1->Lines->Count;
 if(n==0) {ShowMessage ("Нема чисел у Memo1"); return;}
 for(i=0; i<n; i++)
 { x=StrToInt(Memo1->Lines->Strings[i]);
   if(x<0 && x%2==0)              // Якщо x – від'ємне парне число,
   { ShowMessage("Від'ємне парне число є!"); // вивести повідомлення
     return;                      // і зупинити виконування Button1Click.
 } }
 ShowMessage("Від'ємного парного числа нема!");
}
```

Приклад 4.33 Скласти програму табулювання функції $y = f(x)$ при змінюванні x від A до B з кроком h, коли $f(x) = \sqrt[3]{\cos^2|x|}$.

Розв'язок. Досить часто при дослідженні функціональних залежностей виникає потреба побудови графіка $y = f(x)$ чи обчислення значень $y = f(x)$ на заданому проміжку $x \in [A, B]$ з кроком h.

Для побудови графіка функції розташуємо на формі компонент `Chart`, розміщений на закладці `Additional`. Для налагодження компонента `Chart` слід двічі клацнути на зображенні цього компонента. У діалоговому вікні, яке з'явиться, слід клацнути по кнопці `Add`. З додаткового діалогового вікна оберемо тип графіка `Line`. Для збільшення розміру графіка на формі можна вилучити зображення "легенди", для чого слід перейти на закладку `Legend`, в позиції `Visible` вилучити "галочку". З аналогічною метою можна перейти на закладку `Titles` і видалити надпис `TChart`, а можна просто змінити надпис графіка, наприклад на "Графік функції".

У наведеному нижче прикладі програми табулювання заданої функції початковими даними, які слід ввести із компонентів `Edit1`, `Edit2` та `Edit3`, є початкове `A` і кінцеве `B` значення `x`, а також крок змінювання `x - h`.



У циклі, починаючи з `x=A` і допоки буде істинна умова `x<=B+0.1*h`, обчислюється значення функції `y` і виводиться її числове значення до компонента `Memo1` і на графік `Series1`, після чого `x` збільшується на значення кроку `x+=h`. Незначна величина `0.1*h`, яка не перевищує величини кроку, в умові використовується з тієї причини, що при обчисленні існує так зване накопичення похибки обчислень, наприклад, коли 4 ≅ 3,9999(9). У такому разі останнє значення `x` та `y` не виводились би на форму. Виведення точок на графік здійснюється за допомогою методу `AddXY(x,y,"",clRed)`. Чотири параметри в дужках для цього методу задають параметри точки, що виводиться: перше значення `x` – координата по горизонтальній осі, друге значення `y` – координата по вертикальній осі, третій параметр задає параметри відображення числових значень на осях (двоє порожніх лапок `""` означають, що підписи формуватимуться автоматично), четвертий параметр задає колір точки. Колір (color) записують після символів `cl` з великої літери; так, `clRed` задає червоний колір, `clBlue` – синій, `clGreen` – зелений тощо.

Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float A, B, h, x, y;    Memo1->Clear();  Series1->Clear();
  A = StrToFloat(Edit1->Text);
  B = StrToFloat(Edit2->Text);
  h = StrToFloat(Edit3->Text);
  for (x = A; x <= B + 0.1 * h; x += h)
   { y = pow(pow(cos(fabs(x)), 2), (float)1./ 3);
     Memo1->Lines->Add(FormatFloat("0.00",x)+"  "+
                                        FormatFloat("0.0000",y));
     Series1->AddXY(x, y, "", clRed);
   }
}
```

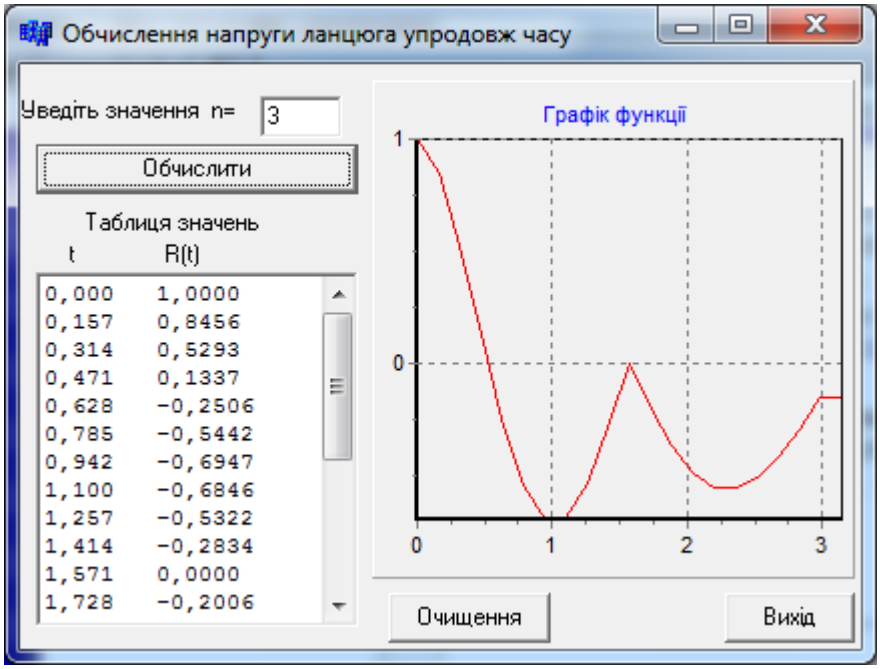
Приклад 4.34 Написати програму обчислення напруги ланцюга $R(t)$, яка змінюється упродовж певного часу t за законом:

$$R(t)=\begin{cases} e^{(-t/n)}\cos(nt) & \text{за} \quad 0\le t<\dfrac{\pi}{2}; \\ e^{(-t/n+1)}\sin((n-1)\cdot t) & \text{за} \quad \dfrac{\pi}{2}\le t\le \pi. \end{cases}$$

якщо t змінюється на проміжку $[0, \pi]$ з кроком $h = \pi/20$. Параметр n ввести з клавіатури.

Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ Memo1->Clear();  Series1->Clear();
  int n=StrToInt(Edit1->Text);
  float t, R, h=M_PI/20;
  for(t=0; t<=M_PI+0.01*h; t+=h)
  { if (t>=0 && t<M_PI/2) R=exp(-t/n)*cos(n*t);
    else
      if (t<=M_PI)   R=exp(-t/(n+1))*sin((n-1)*t);
    Memo1->Lines->Add(FormatFloat("0.000", t)+"   " +
                                        FormatFloat("0.0000", R));
    Series1->AddXY( t, R, "", clRed);
  }
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Edit1->Clear();  Memo1->Clear(); Series1->Clear(); }
//---------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{ Close(); }
```



Приклад 4.35 Написати програму наближеного обчислення інтегралу функції $f(x)=x^3-x^2-4.4x+4$ трьома методами: прямокутників, трапецій та Сімпсона на заданому інтервалі $x\in[a,b]$ із заданою точністю ε.

Розв'язок. У багатьох наукових та інженерних задачах треба віднайти значення інтегралу функції $f(x)$, де x змінюється на проміжку $[a, b]$. Якщо для неперервної на проміжку підінтегральної функції можна віднайти первісну функцію $F(x)$, то визначений інтеграл легко обчислити за формулою Ньютона–Лейбніца:

$$\int_a^b f(x)dx = F(b)-F(a)$$

Але у багатьох випадках не вдається знайти первісну функцію або вона є настільки складною, що обчислювати інтеграл за її допомогою важче, аніж у інші способи. У таких випадках застосовують методи чисельного інтегрування, які базуються на тому, що значення інтегралу дорівнює площі криволінійної трапеції, розміщеної поміж лінією графіка функції та віссю абсцис (рис. 4.1). Згідно з цими методами, проміжок інтегрування $[a, b]$ слід поділити на кілька відрізків (приміром, n відрізків), та замінити площу S зазначеної криволінійної трапеції на суму площин маленьких криволінійних трапецій s_i, на яких задану функцію $f(x)$ замінено наближеною функцією $f_i(x)$, тобто

$$S=\int_a^b f(x)dx=\sum_{i=0}^{n-1} s_i\,.$$

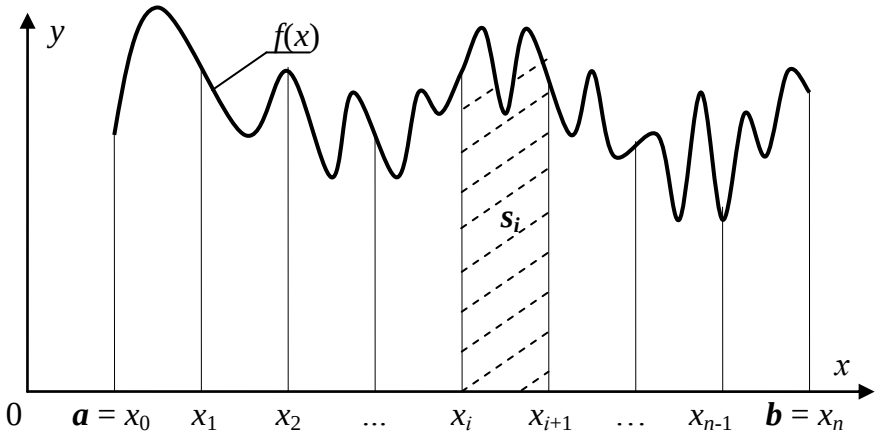


Рис. 4.1. Криволінійні трапеції для інтегрування

Залежно від того, якими є наближені функції $f_i(x)$, існують різні методи (формули) обчислення визначених інтегралів. В усіх цих методах інтеграл наближено обчислюють як певну сукупність значень підінтегральної функції $f(x_i)$, де $x_i = a + ih$ – межі відрізків інтегрування, $h = x_{i+1} - x_i$, – крок інтегрування, $i = 0, 1, \ldots, n$. Найбільш відомими в інженерній практиці є методи чисельного інтегрування: прямокутників, трапецій та Сімпсона.

Для чисельного обчислення інтегралів *методом прямокутників* задану функцію $f(x)$ на кожному відрізку $h = x_{i+1} - x_i$ (де $i = 0, 1, \ldots, n$) замінюють на прямі лінії, паралельні до осі абсцис (рис. 4.2). При цьому криволінійна трапеція замінюється на n прямокутників.

$$S = \sum_{i=0}^{n-1} s_i = \sum_{i=0}^{n-1} (x_{i+1} - x_i) f(x_i) = h \sum_{i=0}^{n-1} f(x_i).$$

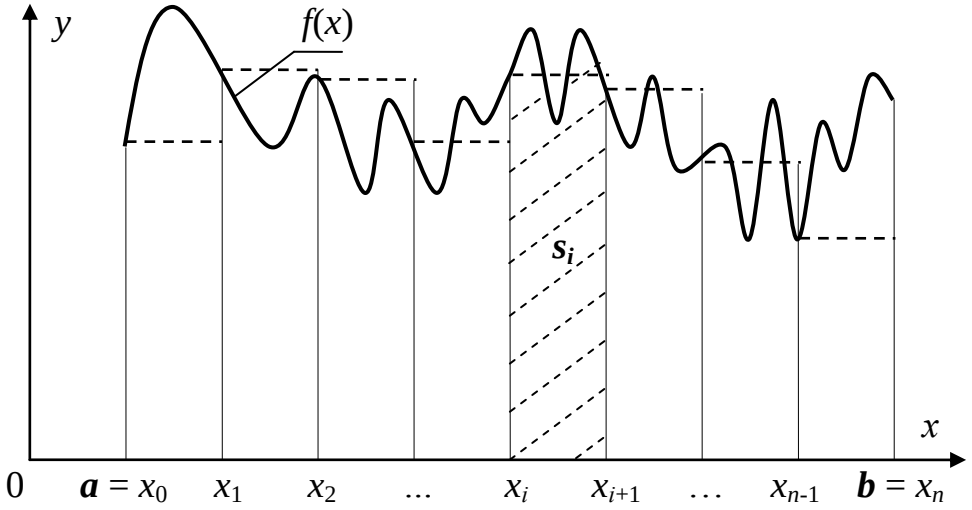


Рис. 4.2. Інтерпретація інтегрування методом прямокутників

В основу *формули трапецій* покладено заміну кривої підінтегральної функції на ламану, тобто з'єднуються прямими лініями значення функцій на кінцях відрізків. Отже, площу криволінійної трапеції наближено замінюємо на суму площин *n* трапецій (рис. 4.3).

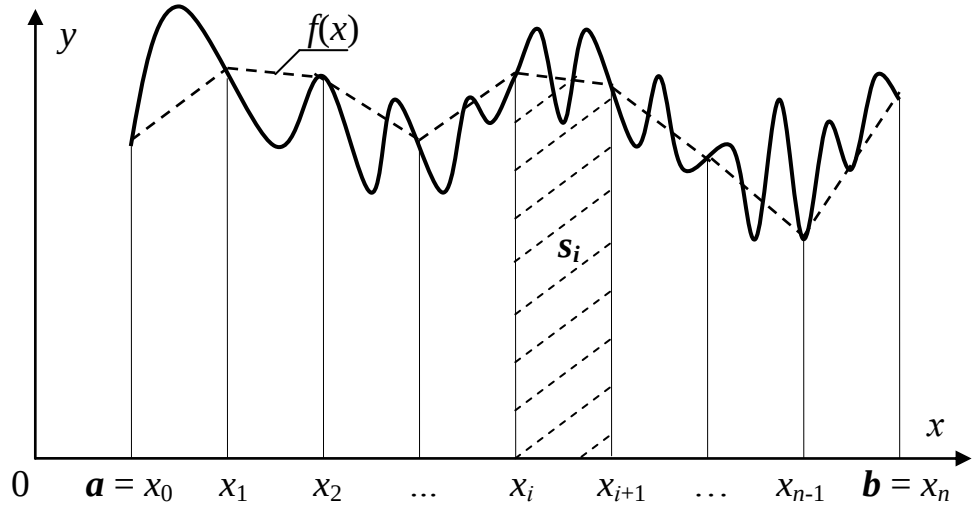


Рис. 4.3. Інтерпретація інтегрування методом трапецій

Загальна площа трапецій *S* і відповідно наближене значення інтегралу за методом трапецій обчислюється за формулою

$$S = \sum_{i=0}^{n-1} s_i = \sum_{i=0}^{n-1} \frac{h}{2} \left(f(x_i) + f(x_{i+1}) \right) = \frac{h}{2} \left(f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(x_i) \right).$$

Для наближеного обчислення інтеграла за *методом Сімпсона* крива підінтегральної функції замінюється на відрізки квадратичних парабол, проведених через кінці кожних трьох сусідніх ординат значень функції, при цьому весь

проміжок інтегрування розбивають на парну кількість n відрізків $[x_i - h, x_i + h]$. Отже, площу криволінійної трапеції наближено замінюємо на суму $n/2$ площин під параболами (рис. 4.4).

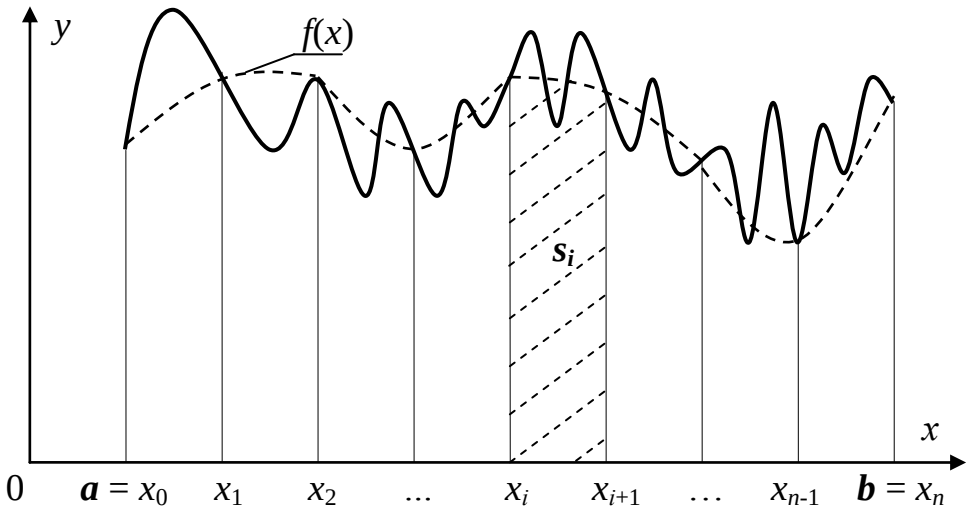


Рис. 4.4. Інтерпретація інтегрування методом Сімпсона

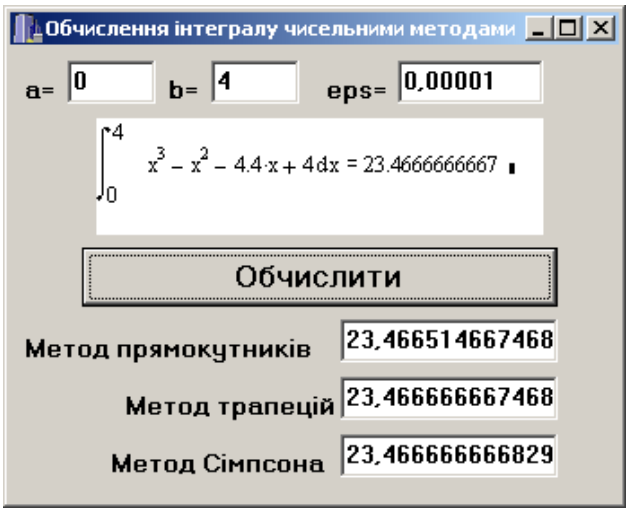
Остаточна формула обчислення інтегралу S методом Сімпсона:

$$S = \frac{h}{3}\left(f(a) + f(b) + 4\sum_{i=1}^{n-1} f(x_i) + 2\sum_{j=1}^{n-2} f(x_j) \right),$$

де i – непарні числа, j – парні числа.

Порівнюючи похибки наведених трьох методів, зазначимо, що метод Сімпсона дає точний результат для поліномів до третього ступеня включно, а методи прямокутників та трапецій є точними лише за інтегрування поліномів ступеня не вище за перший.

Зображення інтегралу на формі організовано за допомогою компонента `Image`  . Зображення у форматі bmp завантажено до властивості `Picture` цього компонента.



Текст програми:

```
// Підінтегральна функція
double f(double x)
{ return x*x*x-x*x-4.4*x+4;
}
// Метод прямокутників
double integral_Pr(double a, double b,double eps)
{ double h=eps,s,x;
  for(s=0, x=a; x<b ; x+=h) s+=f(x);
  return h*s;
}
```

```
// Метод трапецій
double integral_Trap(double a,double b,double eps)
{ double h=eps,s,x;
  for(s=0, x=a+h; x<b ; x+=h) s+=f(x);
  return h/2*(f(a)+f(b)+2*s);
}
// Метод Сімпсона
double integral_Simp(double a,double b,double eps)
{ double h=eps,s1,s2,x; int i, n;
  n=(int)((b-a)/h)+1;
  if(n%2==0) {n++;h=(b-a)/(n-1);}
  for(s1=0, i=1,x=a+h; i<n-1 ; i+=2,x+=2*h) s1+=f(x);
  for(s2=0, i=2,x=a+2*h; i<n-2 ;i+=2, x+=2*h) s2+=f(x)  ;
  return h/3*(f(a)+f(b)+4*s1+2*s2);
}
//---------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{double a,b,i_bar,i_trap,i_simp,eps;
 a=StrToFloat(Edit1->Text);
 b=StrToFloat(Edit2->Text);
 eps=StrToFloat(Edit3->Text);
 i_bar= integral_Pr(a,b,eps);        // Інтеграл методом прямокутників
 Edit4->Text=FloatToStr(i_bar);
 i_trap= integral_Trap(a,b,eps);     // Інтеграл методом трапецій
 Edit5->Text=FloatToStr(i_trap);
 i_simp= integral_Simp(a,b,eps);     // Інтеграл методом Сімпсона
 Edit6->Text=FloatToStr(i_simp);
}
```

Приклади консольних програм з оператором циклу for

Приклад 4.36 Обчислити суму всіх двоцифрових чисел, кратних до числа 3.

Розв'язок. Двоцифрові числа – це цілі числа від 10 до 99 включно. Змінюючи параметр циклу від 12 (перше двоцифрове число, кратне до 3) до 99 (останнє двоцифрове число, кратне до 3) із кроком 3, можна використовувати його як число для додавання до суми.

Текст програми:

```
#include <vcl.h>
#pragma hdrstop
#include <iostream.h>
#include <conio.h>
#pragma argsused
//---------------------------------------------------------------
int main(int argc, char* argv[])
{int i, s=0;                  // Спочатку сума дорівнює 0
 for(i=12;i<=99;i+=3) s+=i;// Обчислити суму чисел від 12 до 99 з кроком 3
```

```
   cout << "Сума двоцифрових чисел, кратних до числа 3, дорівнює"
        << s << endl;        // Виведення значення суми
   getch();  return 0;
}
```

Результат виконання програми:

```
Сума двоцифрових чисел, кратних до числа 3, дорівнює 1665
```

Приклад 4.37 У програмі організовано введення 12 цілих чисел та обчислюється добуток парних з цих чисел.

Розв'язок. У циклі, який повторюється 12 разів, виконуються такі дії:

- ✓ виводиться запрошення для введення наступного числа;
- ✓ вводиться нове число;
- ✓ перевіряється введене число на парність і, якщо це так, то число перемножується на добуток.

Текст програми:

```
#include <vcl.h>
#pragma hdrstop
#include <iostream.h>
#include <conio.h>
#pragma argsused
//---------------------------------------------------------------
int main(int argc, char* argv[])
{ int i, x, p = 1;              // Спочатку добуток дорівнює 1.
  for(i = 1; i <= 12; i++)      // У циклі 12 разів повторити дії:
   { cout << "Ввести " << i << " число: ";//вивести запрошення,
     cin >> x;                  // вести число, і,
     if(x%2==0 && x!=0) p *= x;// якщо число парне, перемножити його.
   }
   cout << "Добуток парних чисел: " << p << endl;
   getch();  return 0;
}
```

Результати виконання програми:

```
Ввести 1 число: 3
Ввести 2 число: 1
Ввести 3 число: -7
Ввести 4 число: 6
Ввести 5 число: 3
Ввести 6 число: -2
Ввести 7 число: 4
Ввести 8 число: 7
Ввести 9 число: 11
Ввести 10 число: -4
Ввести 11 число: -5
Ввести 12 число: 3
Добуток парних чисел: 192
```

У цієї програми є один суттєвий недолік. Якщо парних чисел немає, то результатом обчислення добутку буде виведено 1. Замість цього бажано вивести повідомлення, що парних чисел введено не було. Для цього слід вести підрахунок кількості введених парних чисел. Якщо після циклу ця кількість становитиме 0, то слід вивести повідомлення. Для обчислення кількості візьмемо ще одну цілу змінну, значення якої спочатку є 0 (парних чисел ще нема). Якщо введене число є парне, кількість збільшується на 1.

Текст програми:

```
int main()
{ int i, x, p = 1;
  for(i = 1; i <= 12; i++)
  { cout << "Ввести " << i << " число: ";
     cin >> x;              // Введення числа.
     if(x%2==0 && x!=0)     // Для парних чисел
      { p *= x;             // обчислюватиметься їхній добуток
        k++;                // і кількість таких чисел.
  }   }
  if(k > 0)    // Якщо кількість парних чисел є більше за 0, виведеться на екран
     cout << "Добуток парних чисел: " << p << endl; // добуток,
   else        // інакше – виведеться повідомлення, що парних чисел немає.
     cout << "Парних чисел немає." << endl;
  getch();
  return 0;
}
```

Результати виконання програми:

```
Ввести 1 число: 3
Ввести 2 число: 1
Ввести 3 число: -7
Ввести 4 число: 5
Ввести 5 число: 3
Ввести 6 число: -7
Ввести 7 число: 3
Ввести 8 число: 7
Ввести 9 число: 11
Ввести 10 число: -17
Ввести 11 число: -5
Ввести 12 число: 3
Парних чисел немає.
```

Приклад 4.38 Увести шість дійсних чисел та віднайти найбільше з них.

Розв'язок. Алгоритм пошуку максимального числа:

1) ввести перше число x;

2) вважаємо, що воно є максимальним:

```
max = x;
```

3) у циклі почергово вводимо решту чисел. Кожне з уведених чисел порівнюємо зі значенням max і, якщо число x буде більшим за max, запишемо його значення до max:

```
if (x > max) max = x;
```

Текст програми:

```
int main()
{ int i;
  float x, max;
  cout << "Ввести 1-ше число: ";
  cin >> x;
  max = x;
  for (i = 2; i <= 6; i++)
   { cout << "Ввести " << i << " число ";
     cin >> x;
     if (x > max) max = x;
   }
  cout << "Найбільше число: " << max << endl;
  getch();
  return 0;
}
```

Результати виконання програми:

```
Ввести 1 число: 5
Ввести 2 число: 2
Ввести 3 число: -7
Ввести 4 число: 3
Ввести 5 число: 15
Ввести 6 число: -1
Найбільше число: 15
```

Приклад 4.39 Вивести таблицю степенів двійки від нульової до десятої.

Розв'язок. Для обчислення і виведення усіх значень степенів двійки достатньо організувати лише один цикл.

Текст програми:

```
int main()
{ int n;    // Показник степеня
  int st;   // Значення 2 у степені n
  printf("\n Таблиця степенів двійки \n");
  for(n=0, st=1; n<=10; n++)
   { printf("%3i %5i\n", n, st);
     st*=2;
   }
  printf("\n Для завершення натисніть <Enter>\n");
  getch();
  return 0;
}
```

Результат виконання програми:

```
Таблиця степенів двійки
  0     1
  1     2
  2     4
  3     8
  4    16
  5    32
  6    64
  7   128
  8   256
  9   512
 10  1024
Для завершення натисніть <Enter>
```

Приклад 4.40 Написати програму, яка обчислює часткову суму ряду: $1-1/3+1/5-1/7+1/9-\ldots$ для заданої кількості членів ряду, яка вводиться користувачем (при підсумовуванні достатньо великої кількості членів цього ряду, значення часткової суми наближується до $\pi/4$).

Розв'язок. Для обчислення запишемо формулу для обчислення доданка ряду $\pm\frac{1}{2i-1}$, де $i = 1, 2, 3, \ldots$

Текст програми:

```
int main()
{// Обчислення числа π з використанням властивості ряду 1 – 1/3 + 1/5 – 1/7 + ...
 float x; int n;       // Член ряду та кількість доданків у сумі
 float sum=0;          // Часткова сума, значення якої при підсумовуванні
 // вельми значної (понад 1000) кількості елементів ряду, наближається до π/4
 printf("Обчислення суми ряду 1 -1/3 + 1/5 - 1/7 + ...\n");
 printf("Введіть кількість членів ряда для суми ->" );
 scanf("%i", &n);
 for(int i = 1; i <= n; i++)
 { x = 1./(2*i - 1) ;
   if((i % 2)==0) x = -x;
   sum += x;
 }
 printf("Сума ряду: %2.6f\n", sum);
 printf("Обчислене значення числа ПІ = %2.6f\n", sum * 4);
 printf("\nДля завершення натисніть <Enter>");
 getch(); return 0;
}
```

Результат виконання програми:

```
Обчислення суми ряду 1 -1/3 + 1/5 - 1/7 + ...
Введіть кількість членів ряду для суми -> 10000
Сума ряду: 0.785375
Обчислене значення числа ПІ = 3.141499
Для завершення натисніть <Enter>
```

Приклад 4.41 Послідовно ввести вісім дійсних чисел і обчислити середнє арифметичне додатних з цих чисел.

Розв'язок. Для обчислення середнього арифметичного додатних чисел слід обчислити в циклі їхню суму та кількість. Після циклу, коли сума та кількість вже остаточно обчислено, значення суми слід поділити на кількість, якщо кількість є більше за нуль (серед уведених чисел є додатні).

Текст програми:

```
#include <vcl.h>
#pragma hdrstop
#include <stdio.h>
#include <conio.h>
#pragma argsused
//---------------------------------------------------------
int main(int argc, char* argv[])
{ int i, k = 0;   float x, s = 0, sred;
  for(i = 1; i <= 8; i++)
   { cout << "Ввести " << i << " число ";
     cin >> x;
     if(x > 0) {s += x; k++;} // Накопичення суми і кількості додатних
   }
  if(k > 0)
   { sred = s / k;
     cout << "Середнє арифметичне додатних: " << sred << endl;
   }
  else cout << "Додатних чисел нема" << endl;
  getch();
  return 0;
}
```

Результати виконання програми:

```
Ввести 1 число: 5
Ввести 2 число: 2
Ввести 3 число: -7
Ввести 4 число: 3
Ввести 5 число: 5
Ввести 6 число: -1
Ввести 7 число: 3
Ввести 8 число: 7
Середнє арифметичне додатних: 4.16667
```

Приклад 4.42 Написати програму, яка переведе введене користувачем десяткове число до двійкової системи числення.

Розв'язок. Алгоритм такого переведення розглядався в п. 1.4.4 (див. стор. 19).

Текст програми:

```
#include <vcl.h>
#pragma hdrstop
#include <stdio.h>
```

```
#include <conio.h>
#pragma argsused
//---------------------------------------------------------
int main(int argc, char* argv[])
{int dec; // Десяткове число
 int v, i; // Вага і номер формованого розряду
 printf("\nПереведення десяткового числа до двійкового \n");
 printf("Введіть ціле число від 0 до 255 і натисніть <Enter>");
 printf("\n -> ");
 scanf("%i", &dec);
 printf("Десятковому числу %i відповідає двійкове ", dec) ;
 v = 128; // Вага старшого (восьмого) розряду
 for(i = 1; i <= 8; i++)
 { if(dec >= v)
    { printf("1");
      dec -= v;
    }
   else printf("0");
   v = v / 2; // Вага наступного розряду удвічі менше
 } printf("\n\nДля завершення натисніть <Enter>");
 getch ();  return 0;
}
```

Результати виконання програми:

```
Переведення десяткового числа до двійкового
Введіть ціле число від 0 до 255 і натисніть <Enter>
-> 200
Десятковому числу 200 відповідає двійкове 11001000

Для завершення натисніть <Enter>"
```

Приклад 4.43 Написати програму-тест для перевірки знань таблиці множення. Програма має вивести 10 прикладів і виставити оцінку: за 10 правильних відповідей – "відмінно", за 9 та 8 – "добре", за 7 та 6 – "задовільно", за 5 і менш – "погано".

Текст програми:

```
#include <vcl.h>
#pragma hdrstop
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> // Для доступу до srand та rand (див. підрозд. 3.7)
#include <time.h>
#pragma argsused
//---------------------------------------------------------
int main(int argc, char* argv[])
{// Програма перевіряє знання таблиці множення.
 int numb1,numb2;// Оголошення змінних для зберігання значень співмножників,
 int res;        // добутку,
 int otv;        // відповіді,
```

```
 int kol = 0; // кількості правильних відповідей,
 int i;       // лічильнику циклів
 time_t t;    // поточного часу – для ініціалізації генератора випадкових чисел.
 printf("*** Перевірка знань таблиці множення ***\n");
 printf("Введіть відповідь і натисніть <Enter>\n");
 srand((unsigned) time(&t)); // Ініціалізація генератора випадкових чисел
 for(i = 1; i <= 10; i++)    // 10 тестів
 { numb1 = rand()%7 + 2 ;    // Два випадкових числа
   numb2 = rand()%7 + 2 ;    // у межах від 2 до 9
   res = numb1 * numb2;
   printf("%i x %i=", numb1, numb2);
   scanf("%i",&otv);
   if(otv == res)kol++;
    else printf("Ви помилились! %i x %i = %i \n
                Спробуйте ще раз\n", numb1, numb2, res);
 }
 printf("\nПравильних відповідей: %i\n", kol);
 printf("Ваша оцінка: ");
 switch (kol)
 { case 10: puts("Відмінно"); break;
   case 9: puts("Добре"); break;
   case 8: puts("Добре"); break;
   case 7: puts("Задовільно"); break;
   default: puts("Погано"); break;
 }
 printf("\nДля завершення натисніть <Enter>");
 getch();
 return 0;
}
```

Результати виконання програми:

```
*** Перевірка знань таблиці множення ***
Введіть відповідь і натисніть <Enter>
8 x 4 = 32
3 x 2 = 6
5 x 7 = 35
9 x 6 = 56
Ви помилились! 9 x 6 = 54
Спробуйте ще раз
6 x 7 = 42
9 x 3 = 27
8 x 8 = 64
4 x 3 = 12
2 x 6 = 12
7 x 8 = 56

Правильних відповідей: 9
Ваша оцінка: Добре

Для завершення натисніть <Enter>");
```

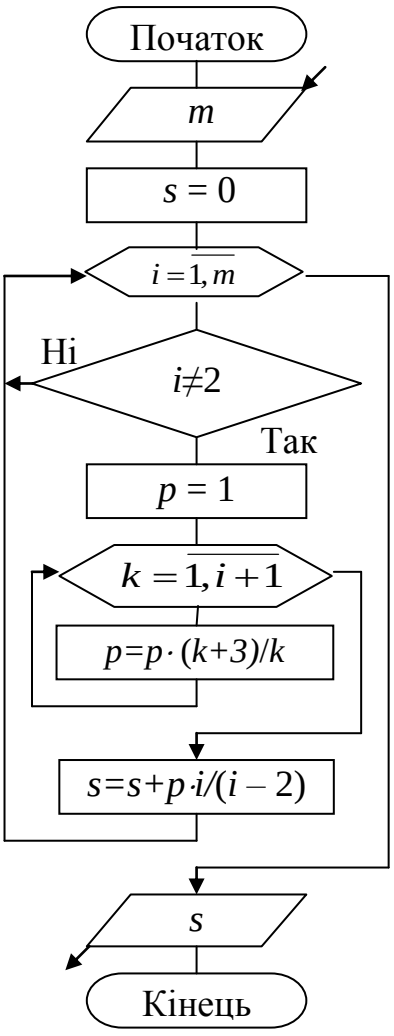
4.4.3 Вкладені цикли

Цикли може бути вкладено один в одного. При використанні вкладених циклів треба складати програму в такий спосіб, щоб внутрішній цикл повністю вкладався в тіло зовнішнього циклу, тобто цикли не повинні перетинатися. Своєю чергою, внутрішній цикл може містити власні вкладені цикли. Імена параметрів зовнішнього та внутрішнього циклів мають бути різними. Припускаються такі конструкції:

```
for(k=1; k<=10; k++)
 {  . . .
   for(i=1; i<=10; i++)
    { . . .
        for(m=1; m<=10; m++)
         {    ...
         }
    }
 }
```

Приклади проектів програм із вкладеними циклами

Приклад 4.44 Обчислити $S=\sum_{i=1}^{m}\frac{i}{i-2}\prod_{k=1}^{i+1}\frac{k+3}{k}$, значення `m` ввести з екрана. З обчислень вилучити доданки та множники, які дорівнюють нулю в чисельнику чи то знаменнику.



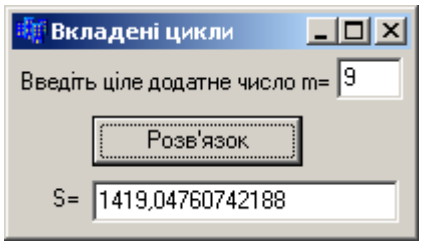
Розв'язок. В цьому прикладі програми наведені цикли є вкладеними один в одного, оскільки параметр внутрішнього циклу `k` залежить від параметра зовнішнього циклу `i` (`k` змінюється від 1 до `i`). Добуток $\prod_{k=1}^{i+1}\frac{k+3}{k}$ є співмножником доданка і обчислюється у внутрішньому циклі у змінній `P`. Оскільки внутрішній цикл складається лише з одного оператора, то операторні дужки `{ }` не є обов'язковими.

Перед зовнішнім циклом для обчислення суми слід обнулити змінну `S`, в якій будуть накопичуватись доданки, а перед зовнішнім циклом для обчислення добутку змінній `p` слід присвоїти значення 1.

Оскільки при обчислюванні добутку беруть участь лише цілі числа, то, щоб при діленні не втратити дробову частину, слід перетворити чисельник до дійсного типу; для цього можна дописати крапку до числа 3: `(k+3.)/k`.

Текст програми для кнопки “Розв’язок”:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, k, m = StrToInt(Edit1->Text);
  float S = 0, p;
  for(i = 1; i <= m; i++)
    if( i != 2)
    { p = 1;
      for(k = 1; k <= i+1; k++)
        p*=(k+3.)/k;
      S += i/(i-2.) * p;
    }
  Edit2->Text = FloatToStr(S);
}
```

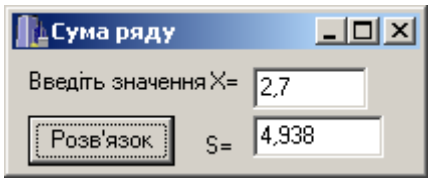


Приклад 4.45 Обчислити суму ряду $S=\sum_{i=1}^{7}\frac{2x^{2i-1}}{3(2i-1)!}$, де $i = 1, 2, \ldots, 7$.

Розв’язок. Для обчислення суми `S` треба підсумувати сім доданків, для обчислення кожного з яких слід сформувати вкладений цикл для підрахунку факторіалів $(2i-1)!$. У даній програмі кожний доданок обчислюється в окремій змінній `a`.

Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, k, fact;
  float s=0, a, x=StrToFloat(Edit1->Text);
  for(i=1; i<=7; i++)
   { fact=1;
     for(k=1; k<=2*i-1; k++)
       fact *= k;
     a = 2*pow(x,2*i-1)/(3*fact);
     s += a;
   }
  Edit2->Text = FloatToStrF(s, ffGeneral, 4, 3);
 }
```



Приклади консольних програм із вкладеними циклами

Приклад 4.46 Написати програму, котра генерує три послідовності з десяти випадкових чисел в діапазоні від 1 до 10, виводить на екран і обчислює середнє арифметичне кожної послідовності.

Текст програми:

```
// Обчислення середнього арифметичного послідовностей випадкових чисел
#include <vcl.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
```

```
#pragma argsused
//---------------------------------------------------------
int main(int argc, char* argv[])
{int r;          // Випадкове число
 int sum;        // Сума чисел послідовності
 float sr;       // Середнє арифметичне
 int i,j;        // Лічильники циклів
 time_t t;       // Поточний час – для ініціалізації генератора випадкових чисел
 // Ініціалізація генератора випадкових чисел (див. докладніше підрозд. 3.7)
 srand((unsigned) time(&t));
 for(i = 1; i <= 3; i++)      // Організуються три послідовності
 { printf("\n Випадкові числа: ");
   sum =0;
   for(j = 1; j <= 10; j++)   // Для послідовності з 10-ти чисел
   { r = rand() % 10 +1 ;     // генерується випадкове число
     printf ("%i ", r) ;      // і виводиться його значення
     sum += r;
   }
   sr = (float)sum / 10;
   printf("\n Середнє арифметичне: %3.2f\n", sr);
 }
 printf("\n Для завершення натисніть <Enter>");
 getch();  return 0;
}
```

Результат виконання програми:

```
Випадкові числа: 6 4 5 2 4 5 6 5 1 8
Середнє арифметичне: 4.60

Випадкові числа: 8 9 8 3 6 6 8 7 6 6
Середнє арифметичне: 6.70

Випадкові числа: 10 3 8 4 7 4 7 5 2 2
Середнє арифметичне: 5.20

Для завершення натисніть <Enter>
```

Приклад 4.47 Вивести на екран квадрат Піфагора – таблицю множення.

Текст основної програми:

```
#include <vcl.h>
#pragma hdrstop
#include <conio.h>
#include <iostream.h>
#pragma argsused
//---------------------------------------------------------
int main(int argc, char* argv[])
{ int i,j;      // Номер рядка і стовпчика таблиці
  printf("\n Квадрат Піфагора - таблиця множення \n\n");
  for(j=1; j<=10; j++)  printf("%4i", j);
```

```
  printf("\n \n");
  for(i=1; i<=10; i++)
    { printf("%4i  ", i);
      for(j=1; j<=10; j++)  printf("%4i", i*j);
      printf("\n");
    }
  printf("\n Для завершення натисніть <Enter>\n");
  getch();  return 0;
}
```

Результат виконання програми:

```
Квадрат Піфагора – таблиця множення

        1   2   3   4   5   6   7   8   9  10

   1    1   2   3   4   5   6   7   8   9  10
   2    2   4   6   8  10  12  14  16  18  20
   3    3   6   9  12  15  18  21  24  27  30
   4    4   8  12  16  20  24  28  32  36  40
   5    5  10  15  20  25  30  35  40  45  50
   6    6  12  18  24  30  36  42  48  54  60
   7    7  14  21  28  35  42  49  56  63  70
   8    8  16  24  32  40  48  56  64  72  80
   9    9  18  27  36  45  54  63  72  81  90
  10   10  20  30  40  50  60  70  80  90 100

Для завершення натисніть <Enter>
```

Приклад 4.48 Увести цілі числа `m` і `n` та вивести на екран:

```
**********
**********
**********
```

де `m` – кількість рядків “зірочок” і `n` – кількість “зірочок” у рядку.

Розв’язок. Для розв’язання поставленого завдання слід організувати два вкладені цикли: зовнішній – для організації виведення окремих `m` рядків, тобто переходу на новий рядок, і внутрішній – для безпосереднього виведення `n` “зірочок” у рядок.

Текст основної програми:

```
#include <conio.h>
#include <iostream.h>
//-------------------------------------------------------------
int main(int argc, char* argv[])
{ int i, j, m, n;
  cout << "Ввести кількість рядків m: ";
  cin >> m;
  cout << "Ввести кількість рядків n: ";
  cin >> n;
  clrscr();                              // Екран буде очищено
```

```
  for(i=1;i<=m;i++)
  { for(j=1;j<=n;j++) cout << "*";   // Виведення рядка з n "зірочок"
    cout << endl;                    // Перехід на новий рядок
  }
  getch();   return 0;
}
```

Результат виконання програми:

```
Ввести кількість рядків m: 2
Ввести кількість рядків n: 8
********
********
```

Приклад 4.49 Увести кількість рядків і вивести на екран трикутник "зірочок":

```
*
**
***
****
. . .
```

Розв'язок. Як і у попередньому прикладі, для виведення одного рядка слід сформувати окремий цикл, а для виведення кількох таких рядків необхідно вкласти перший цикл у цикл, який буде формувати перехід до наступного рядка. У першому рядку має бути лише одна "зірочка", у другому – дві, у третьому – три і т. д. Тобто рядок з номером `i` має складатися з `i` "зірочок", тому внутрішній цикл виконуватиметься `i` разів.

Текст програми:

```
#include <conio.h>
#include <iostream.h>
//-------------------------------------------------------------
int main(int argc, char* argv[])
{ int i, j, m;
  cout << "Ввести кількість рядків m: ";
  cin >> m;
  clrscr();
  for(i=1; i<=m; i++)
  { for(j=1; j<=i; j++) cout << "*";
    cout << endl;
  }
  getch();   return 0;
}
```

Результат виконання програми:

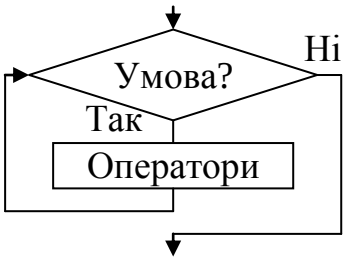
```
Ввести кількість рядків m: 5
*
**
***
****
*****
```

4.4.4 Оператори циклу з передумовою while та післяумовою do-while

Оператори з передумовою та післяумовою використовуються для організації циклів і є альтернативними операторові `for`. Зазвичай цикл з передумовою використовується, якщо кількість повторювань заздалегідь є невідома або немає явно вираженого кроку змінювання параметра циклу. А для багаторазових повторювань тіла циклу відомим є вираз умови, за істинності якої цикл продовжує виконання. Цю умову слід перевіряти кожного разу перед черговим повторенням. Приміром, при зчитуванні даних з файла умовою є наявність цих даних у файлі, тобто повторювати читання даних слід аж допоки вказівник не добудеться кінця файла (докладніше див. розд. 12).

Синтаксис циклу з передумовою:

```
while (<умова>)
  { <тіло циклу> };
```

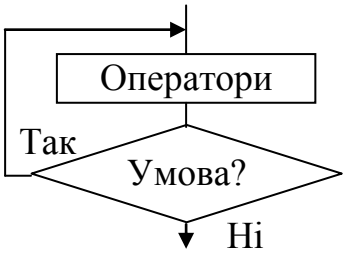


Послідовність операторів (тіло циклу) виконується, допоки *умова* є істинна (`true`, має ненульове значення), а вихід з циклу здійснюється, коли *умова* стане хибною (`false`, матиме нульове значення). Якщо *умова* є хибною при входженні до циклу, то *послідовність операторів* не виконуватиметься жодного разу, а керування передаватиметься до наступного оператора програми.

Цикл з післяумовою використовується, якщо є потреба перевіряти умову кожного разу після чергового повторення. Відмінність циклу з передумовою від циклу з післяумовою полягає в першій ітерації: цикл з післяумовою завжди виконується принаймні одноразово незалежно від умови.

Синтаксис циклу з післяумовою:

```
do {
    <тіло циклу>
    } while (<умова>);
```



Послідовність операторів (*тіло циклу*) виконується один чи кілька разів, допоки *умова* стане хибною (`false` чи дорівнюватиме нулю). Якщо *умова* є істинна (ненульова), то оператори тіла циклу виконуються повторно. Оператор циклу `do-while` використовується в тих випадках, коли є потреба виконати тіло циклу хоча б одноразово, оскільки перевірка умови здійснюється після виконання операторів.

Якщо тіло циклу складається з одного оператора, то операторні дужки `{}` не є обов'язкові.

Оператори `while` та `do-while` можуть завчасно завершитись при виконуванні операторів `break` (див. п. 4.4.5), `goto` (див. п. 4.3.2), `return` (вихід з поточної функції, див. підрозд. 8.1) усередині тіла циклу.

Варто зауважити, що в тілі циклу слід передбачати змінювання параметрів, які беруть участь в умові, інакше умову виходу з циклу ніколи не буде виконано й відбуватиметься зациклювання.

Розглянемо відмінність роботи різних операторів циклу на прикладі обчислювання суми всіх непарних чисел у діапазоні від 10 до 100:

1) з використанням оператора `for`

```
int i, s=0;
for(i=11; i<100; i += 2) s += i;
```

2) з використанням оператора `while`

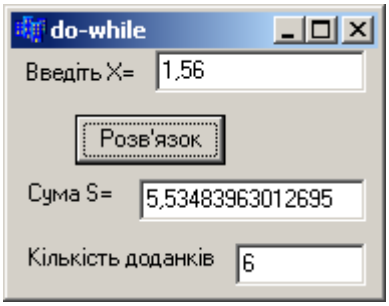
```
int s=0, i=11;
while(i<100) { s += i; i += 2; }
```

3) з використанням оператора `do-while`

```
int s=0, i=11;
do { s += i; i += 2;} while(i<100);
```

Приклади програм із застосуванням циклів while та do-while

Приклад 4.50 Обчислити суму ряду $S=\sum_{k=1}^{\infty}\frac{x^{2k+1}}{(2k-1)!}$, підсумовуючи члени ряду, значення яких за модулем перевищують задану точність $\varepsilon=10^{-4}$. Визначити кількість доданків. Значення x $(-2 < x < 2)$ вводити з клавіатури.



Розв'язок. Для цього завдання в програмі недоцільно використовувати оператор циклу з параметром `for`, оскільки кількість повторювань попередньо є невідома. Доцільним буде використання оператора циклу з післяумовою **do-while**, оскільки на момент першої перевірки умови вже треба знати значення першого доданка.

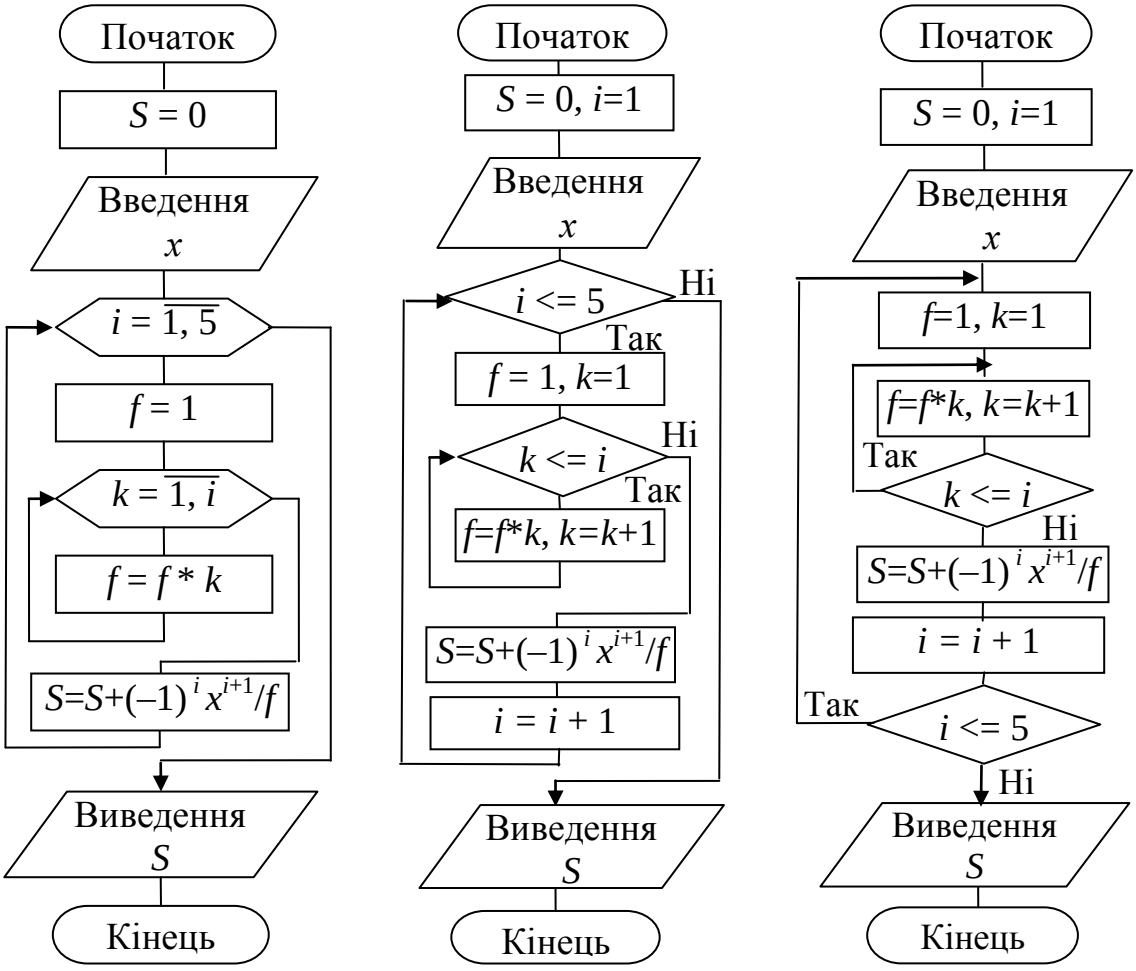
Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float x, a, s=0;              // Оголошення змінних і встановлення
  int f, i, k=0;                // їхніх початкових значень
  x=StrToFloat(Edit1->Text);    // Введення значення x
  do                            // Цикл з післяумовою
   { k++;                       // Збільшення змінної k на 1
     for(i=1,f=1; i<=2*k-1; i++) f *= i; // Обчислення факторіала
     a=pow(x, 2*k+1)/ f;        // Обчислення k-го доданка
     s+=a;                      // Підсумовування доданків
   }
  while(fabs(a)>=1e-4);
  Edit2->Text=FloatToStr(s) ;   // Виведення обчисленої суми
  Edit3->Text=IntToStr(k);      // та кількості доданків
}
```

Приклад 4.51 Обчислити суму знакозмінного ряду $S=\sum_{i=1}^{5}\frac{(-1)^i x^{i+1}}{i!}$ трьома варіантами, використовуючи різні оператори циклу.

Розв'язок. Тут $(-1)^{i+1}$ за непарних значень i (1, 3, 5, …) дорівнює 1, а за парних значень i (2, 4, 6, …) – (–1), тобто розглядається знакозмінний ряд, де всі парні доданки будуть від'ємними, а всі непарні – зі знаком "+".

Схеми алгоритмів усіх трьох програм наведено нижче.



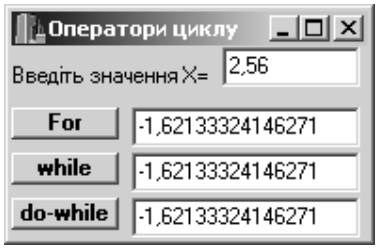
а **б** **в**

Схеми алгоритмів програм з різними операторами циклу:
а – **for**; б – **while**; в – **do-while**

Програма з використанням різних операторів циклу:

```
void __fastcall TForm1::Button1Click(TObject *Sender)    // for
{ int i, f, k;   float S=0, x=StrToFloat(Edit1->Text);
  for(i=1; i<=5; i++)
   { for(k=1, f=1; k<=i; k++) f *= k;
     S += pow(-1,i)*pow(x,i+1)/f;
   }
  Edit2->Text = FloatToStr(S);
}
//---------------------------------------------------------
```

```
void __fastcall TForm1::Button2Click(TObject *Sender) // while
{ int i=1, f, k;
  float S=0, x=StrToFloat(Edit1->Text);
  while(i<=5)
   { f=1; k=1;
     while(k<=i) { f *= k; k++; }
     S += pow(-1, i) * pow(x, i+1) / f;
     i++;
   }
  Edit3->Text = FloatToStr(S);
}
//---------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender) // do-while
{ int i=1, f, k;
  float S=0, x=StrToFloat(Edit1->Text);
  do
   { f=1; k=1;
     do
      { f *= k;
        k++;
      }
     while(k <= i);
     S += pow(-1, i) * pow(x, i+1) / f;
     i++;
   }
  while(i <= 5);
  Edit4->Text = FloatToStr(S);
}
```



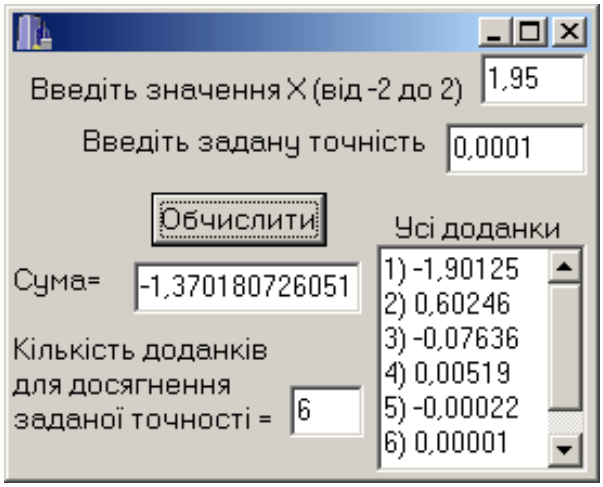
Приклад 4.52 Обчислити суму ряду $y = \sum_{k=1}^{\infty} \frac{(-1)^k x^{2k}}{2k(2k-1)!}$, підсумовуючи члени ряду, значення яких за модулем є більшими за задану точність ε. Визначити кількість доданків. Значення x $(-2 < x < 2)$ та $\varepsilon = 10^{-4}$ вводити з клавіатури.

Розв'язок. Наведемо два способи розв'язання цього завдання. Для наочності й контролю правильності окремо виведемо усі доданки.

Перший спосіб розв'язання

```
#include <math.h>
void __fastcall TForm1::Button1Click(TObject *Sender)
{Memo1->Clear();
 float x, y = 0, u, eps;
 x = StrToFloat(Edit1->Text);
 eps = StrToFloat(Edit2->Text);
 int i, f, k = 1;
 do
```

```
    {for(i = 1, f = 1; i <= 2 * k - 1; i++) f *= i;
     u = pow(-1, k)*pow(x, 2 * k) / (2 * k * f);
     Memo1->Lines->Add(IntToStr(k)+")"+FormatFloat("0.00000",u));
     y += u;
     k++;
    }
    while (fabs(u) >= eps) ;
    Edit3->Text = FloatToStr(y);
    Edit4->Text = IntToStr(k - 1);
}
```



Вигляд цієї форми не буде відрізнятися для кожного з наведених способів розвязання цієї задачі.

Другий спосіб розв'язання

Для того щоб зробити алгоритм програми більш оптимальним, його можна вдосконалити, але для цього слід обчислити рекурентний множник. Це дозволить в даній програмі позбавитись від вкладеного циклу для обчислення факторіала. Зупинимось більш докладно на виведенні рекурентної формули.

Подамо суму ряду $y = \sum_{k=1}^{\infty} \frac{(-1)^k x^{2k}}{2k(2k-1)!}$ у вигляді $y = \sum_{k=1}^{\infty} u_k$, де $u_k = \frac{(-1)^k x^{2k}}{2k(2k-1)!}$. Рекурентний множник R – це співвідношення двох поряд розташованих членів ряду:

$$u_2 = R \cdot u_1,\ u_3 = R \cdot u_2,\ \ldots,\ u_k = R \cdot u_{k-1},\ u_{k+1} = R \cdot u_k,$$

звідки

$$\boxed{R = \frac{u_k}{u_{k-1}}} \text{ або } \boxed{R = \frac{u_{k+1}}{u_k}}.$$

Тобто можна обчислити рекурентний множник зі співвідношення з попереднім членом ряду або зі співвідношення з наступним членом ряду. Наведемо обидва різновиди обчислення рекурентного множника.

1 різновид обчислення рекурентного множника

Для визначення R до формули $R=\frac{u_k}{u_{k-1}}$ слід підставити $u_k=\frac{(-1)^k x^{2k}}{2k(2k-1)!}$ та u_{k-1}. Для обчислення u_{k-1} підставимо до виразу для u_k $(k-1)$ замість k:

$$u_{k-1}=\frac{(-1)^{k-1}x^{2(k-1)}}{2(k-1)(2(k-1)-1)!}=\frac{(-1)^{k-1}x^{2(k-1)}}{2(k-1)(2k-3)!};$$

$$R=\frac{u_k}{u_{k-1}}=\frac{\frac{(-1)^k x^{2k}}{2k(2k-1)!}}{\frac{2(k-1)(2k-3)!}{(-1)^{k-1}x^{2(k-1)}}}=\frac{(-1)^k\cdot x^{2k}\cdot(2k-2)\cdot(2k-3)!}{(-1)^{k-1}\cdot x^{2k-2}\cdot 2k\cdot(2k-1)!}=$$

$$=\frac{(-1)^{k-k+1}\cdot x^{2k-(2k-2)}\cancel{(2k-2)}}{2k}\cdot\frac{}{\cancel{1\cdot 2\cdot\ldots\cdot(2k-3)}\cdot\cancel{(2k-2)}\cdot(2k-1)}=-\frac{x^2}{2k(2k-1)}.$$

Окрім рекурентного множника R, треба обчислити перший член ряду за $k = 1$:

$$u_1=\frac{(-1)^1 x^2}{2(2-1)!}=-\frac{x^2}{2}.$$

Текст програми

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float x, y, u, r, eps;    int i, f, k = 1;
  Memo1->Clear();
  x = StrToFloat(Edit1->Text);   eps = StrToFloat(Edit2->Text);
  u = -x*x/2;
  y = u;
  Memo1->Lines->Add(IntToStr(k)+")"+FormatFloat("0.00000",u));
  do
  {k++;
   r = -x*x/(2*k*(2*k-1));
   u *= r ;
   Memo1->Lines->Add(IntToStr(k)+")"+FormatFloat("0.00000",u));
   y += u;
  } while(fabs(u)>=eps);
  Edit3->Text=FloatToStr(y);  Edit4->Text=IntToStr(k);
}
```

2 різновид обчислення рекурентного множника

Для визначення R до формули $R=\frac{u_{k+1}}{u_k}$ слід підставити $u_k=\frac{(-1)^k x^{2k}}{2k(2k-1)!}$ та u_{k+1}. Для обчислення u_{k+1} підставимо до виразу для u_k $(k+1)$ замість k:

$$u_{k+1}=\frac{(-1)^{k+1}x^{2(k+1)}}{2(k+1)(2(k+1)-1)!}=\frac{(-1)^{k+1}x^{2(k+1)}}{2(k+1)(2k+1)!};$$

$$R=\frac{u_{k+1}}{u_k}=\frac{\dfrac{(-1)^{k+1}x^{2(k+1)}}{2(k+1)(2k+1)!}}{\dfrac{(-1)^{k}x^{2k}}{2k(2k-1)!}}=\frac{(-1)^{k+1}\cdot x^{2k+2}\cdot 2k\cdot(2k-1)!}{(-1)^{k}\cdot x^{2k}\cdot 2(k+1)\cdot(2k+1)!}=$$

$$=\frac{(-1)^{k+1-k}\cdot x^{2k+2-2k}\cancel{k}}{(k+1)}\cdot\frac{\cancel{1\cdot 2\cdot\ldots\cdot(2k-1)}}{\cancel{1\cdot 2\cdot\ldots\cdot(2k-1)}\cdot 2\cancel{k}\cdot(2k+1)}=-\frac{x^2}{2(k+1)(2k+1)}.$$

Окрім рекурентного множника *R*, треба обчислити перший член ряду за $k = 1$:

$$u_1=\frac{(-1)^1x^2}{2(2-1)!}=-\frac{x^2}{2}.$$

Текст програми

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float x, y, u, r, eps;    int i, f, k = 1;
  Memo1->Clear();
  x = StrToFloat(Edit1->Text);   eps = StrToFloat(Edit2->Text);
  u = -x*x/2;
  y = u;
  Memo1->Lines->Add(IntToStr(k)+")"+FormatFloat("0.00000",u));
  do
  {r = -x*x/(2*(k+1)*(2*k+1));
   u *= r ;
   Memo1->Lines->Add(IntToStr(k)+")"+FormatFloat("0.00000",u));
   y += u;
   k++;
 } while(fabs(u)>=eps);
  Edit3->Text=FloatToStr(y);  Edit4->Text=IntToStr(k);
}
```

Приклади консольних програм із застосуванням циклів while та do-while

Приклад 4.53 Обчислити квадрати натуральних чисел від 1 до 10 трьома варіантами, використовуючи різні оператори циклу.

Розв'язок. Оскільки натуральні числа – це цілі додатні числа 1, 2, 3, 4, ..., доцільно використати змінну `i` як параметр циклу, яка змінюватиметься від 1 до 10.

У циклі десятиразово повторюватимуться такі дії:

- ✓ обчислення квадрату числа `i`;
- ✓ виведення на екран значення змінної `i` та її квадрату;
- ✓ збільшення змінної `i` на 1.

Текст програми з циклом `for`:

```
#include <iostream.h>
#include <conio.h>
//---------------------------------------------------------
```

```
int main(int argc, char* argv[])
{int i, x;
 for(i=1; i<=10; i++)
  { x = i * i;
    cout << "Квадрат числа " << i << " дорівнює " << x << endl;
  }
 getch();
 return 0;
}
```

Результат виконання програми:

```
Квадрат числа 1 дорівнює 1
Квадрат числа 2 дорівнює 4
Квадрат числа 3 дорівнює 9
Квадрат числа 4 дорівнює 16
Квадрат числа 5 дорівнює 25
Квадрат числа 6 дорівнює 36
Квадрат числа 7 дорівнює 49
Квадрат числа 8 дорівнює 64
Квадрат числа 9 дорівнює 81
Квадрат числа 10 дорівнює 100
```

Напишемо цю програму з циклами `while` та `do-while` замість `for`. Перше значення змінної `i` слід задати до циклу. Також слід пам'ятати, що умовні оператори циклу не збільшують автоматично значення параметра, а тому збільшення параметра слід прописати у тілі циклу.

Текст програми з циклом `while`:

```
int main()
{ int i=1, x;
  while(i<=10)            // Якщо число не перевищує 10,
   {x = i * i;            // обчислюватиметься квадрат його значення
                          // і виводитиметься число разом із квадратом
    cout << "Квадрат числа " << i << " дорівнює " << x << endl;
    i++;                  // Збільшення змінної циклу на 1
   }
 getch();  return 0;
}
```

Текст ідентичної програми з циклом `do-while`:

```
int main(int argc, char* argv[])
{ int i = 1, x;
  do
   {x = i * i;
    cout << "Квадрат числа " << i << " дорівнює " << x << endl;
    i++;
   } while (i <= 10);
  getch();  return 0;
}
```

Приклад 4.54 Уводити цілі числа, допоки не буде введене трицифрове число, і обчислити кількість введених чисел та суму чисел, які належать до інтервалу [–4, 11).

Розв'язок. Трицифрове число – це число, модуль якого перебуває поміж 100 та 999. Отже, вираз умови продовження виконання циклу матиме вигляд:

```
abs(x)<100 || abs(x)>999
```

Якщо для розв'язання поставленого завдання застосовувати цикл `while`, то ще до початку циклу слід знати перше значення змінної `x`, яка буде вводитись згодом у циклі, або присвоїти `x` будь-яке нетрицифрове значення до циклу, наприклад, 0, щоб можна було увійти до циклу.

Текст програми з циклом while:

```
int main()
{ int x=0, k=0, s=0;
  while(abs(x)<100 || abs(x)>999)    // Допоки число x не є трицифрове,
   { cout << "Ввести число: ";
     cin >> x;
     k++;                            // збільшується кількість уведених чисел і,
     if (x>=-4 && x<11)              // якщо x належить до проміжку [–4, 11),
     s += x;                         // додається його значення до суми.
   }
  cout << "Введено чисел: " << k << endl;
  cout << "Сума чисел з проміжку [-4, 11): " << s << endl;
  getch();
  return 0;
}
```

Результат виконання програми:

```
Ввести число: 4
Ввести число: 8
Ввести число: -1
Ввести число: 3
Ввести число: 9
Ввести число: 54
Ввести число: 2
Ввести число: -87
Ввести число: 4
Ввести число: 443
Введено чисел: 10
Сума чисел з проміжку [-4, 11): 29
```

У цьому прикладі більш оптимально використовувати цикл `do-while`, що надасть можливість не хвилюватися стосовно першого значення `x` – цикл розпочнеться поза всяких умов.

Текст основної програми:

```
int main()
{ int x, k=0, s=0;
```

```
  do
  { cout << " Ввести число: ";      cin >> x;
    k++;
    if (x>=-4 && x<11) s += x;
  } while(abs(x)<100 || abs(x)>999);
  cout << " Введено чисел:  " << k << endl;
  cout << " Сума чисел з проміжку [-4, 11): " << s << endl;
  getch();  return 0;
}
```

Приклад 4.55 Генерувати й виводити випадкові числа з проміжку [–5, 10), допоки числа збільшуються. Обчислити середнє арифметичне цих чисел.

Розв'язок. Для генерування випадкових цілих додатних чисел існує функція **random()**, а для ініціалізації генератора випадкових чисел – функція **randomize()** (див. докладніше підрозд. 3.7). Параметром функції `random()` у дужках зазначається верхня межа чисел. Приміром, для генерування чисел з проміжку [–5, 10) слід записати цю функцію в такий спосіб:

```
x = random(15) - 5;
```

Числа мають зростати, тобто кожне наступне має бути більше за попереднє. Ми повинні зберігати попереднє число у певній змінній.

Текст основної програми:

```
int main()
{ int x, y, s, k = 1;
  randomize();          // Ініціалізація генератора випадкових чисел
  x = random(15) - 5; // Перше випадкове число x
  cout<<"Число: "<<x<<endl;
  s = x;
  do{ y = x;            // У циклі запам'ятовується число x у змінній y,
      x = random(15) - 5; // і генерується наступне випадкове число x,
      cout<<"Число: "<<x<<endl;  // яке виводиться на екран.
      if(x>y)           // Порівнюється число з попереднім і, якщо
       { s+=x;          // воно є більше за попереднє, обчислюється сума s
         k++;   }       // та кількість k таких чисел.
     } while(x>y);      // Вихід з циклу відбудеться за умови
                        // введення порівняно меншого числа.
  cout << "Кількість чисел: " << k << endl;
  cout << "Середнє арифметичне: " << (float) s/k << endl;
  getch();  return 0;
}
```

Результати виконання програми:

```
Число: 0
Число: 6
Число: 8
Число: 5
Кількість чисел:  3
Середнє арифметичне: 4.66667
```

Приклад 4.56 Написати програму обчислення найбільшого спільного дільника (НСД) двох цілих чисел, тобто реалізувати алгоритм Евкліда.

Текст програми:

```
#include <stdio.h>
#include <conio.h>
//---------------------------------------------------------
int main(int argc, char* argv[])
{printf("\nОбчислення найбільшого спільного дільника");
 printf("двох цілих чисел.\n");
 printf("Уведіть в один рядок два числа і натисніть <Enter>");
 printf("\n -> ");
 int nod;        // Найбільший спільний дільник (НДС)
 int nl,n2;      // Числа, НСД яких треба обчислити
 int r;          // Остача від ділення nl на n2
 scanf("%i %i", &nl, &n2) ;
 printf("НСД чисел %i і %i - це ", nl, n2);
 while (nl % n2)
  {r = nl % n2; // Остача від ділення
   nl = n2;
   n2 = r;
  }
 nod =n2;
 printf("%i\n", nod);
 printf("\nДля завершення натисніть <Enter>");
 getch();
 return 0;
}
```

Результати виконання програми:

```
Обчислення найбільшого спільного дільника двох цілих чисел.
Уведіть до одного рядка два числа і натисніть <Enter>
 -> 121 33
НСД чисел 121 і 33 – це 11

Для завершення натисніть <Enter>
```

Приклад 4.57 Увести цілі числа до `Memo1` і скопіювати парні числа з `Memo1` до `Memo2` допоки їхня сума не стане більшою за 10.

Розв'язок. При копіюванні чисел слід зупинитись, якщо вже скопійовано всі числа чи то якщо сума скопійованих чисел є більша за 10.

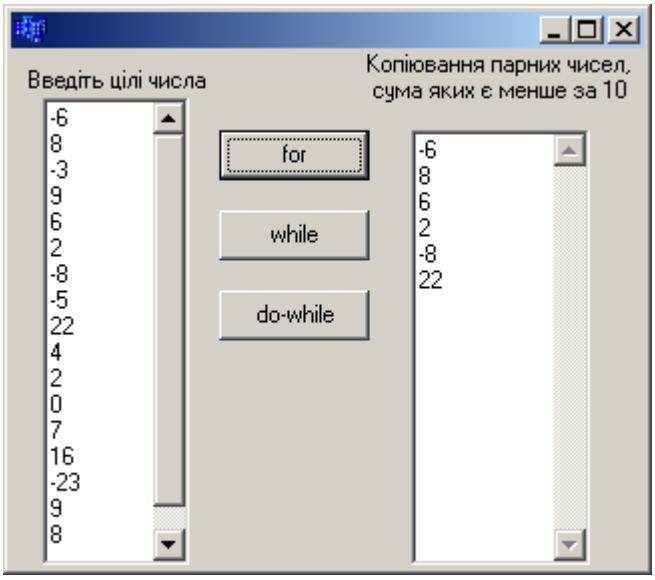
Наведемо три варіанти розв'язання, використовуючи три цикли: `for`, `while`, `do-while`.

Текст програми:

```
// Розв'язання за допомогою циклу for
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int x, i, n, S=0;
   n=Memo1->Lines->Count;
```

```
  if(n==0) {ShowMessage ("Введіть числа до Memo1"); return;}
  Memo2->Clear();
  // У циклі умова i<n означає, що в Memo1 ще є числа для зчитування і копіювання;
  //  умова S<=10 означає, сума чисел, котрі було скопійовано, є менше за 10.
  for(i=0; i<n && S<=10; i++)
  { x=StrToInt(Memo1->Lines->Strings[i]);
    if(x%2==0)
     { Memo2->Lines->Add(IntToStr(x));
       S+=x;
     }
  }
}
//---------------------------------------------------------
// Розв'язок за допомогою  циклу  while
void __fastcall TForm1::Button2Click(TObject *Sender)
{ int x, i=0, n, S=0;
  n=Memo1->Lines->Count;
  if(n == 0)
  { ShowMessage ("Введіть числа до Memo1"); return; }
  Memo2->Clear();
  while(i<n && S<=10)
  { x=StrToInt(Memo1->Lines->Strings[i]);
    if(x%2 == 0)
     { Memo2->Lines->Add(IntToStr(x));
       S+=x;
     }
    i++;
  }
}
//---------------------------------------------------------
// Розв'язок за допомогою  циклу  do-while
void __fastcall TForm1::Button3Click(TObject *Sender)
{ int x, i=0, S=0, n=Memo1->Lines->Count;
  if(n == 0)
  { ShowMessage("Введіть числа до Memo1"); return;}
  Memo2->Clear();
  do
  { x=StrToInt(Memo1->Lines->Strings[i]);
    if(x%2 == 0)
     { Memo2->Lines->Add(IntToStr(x));
       S += x;
     }
    i++;
  }
  while (i<n && S<=10);
}
```

Форма з результатами роботи наведеної програми:



Приклад 4.58 Зчитати числа з Memo, допоки не зустрінеться число зі значенням 0. Віднайти найменше з цих чисел.

Розв'язок. Для пошуку мінімального числа використовуватимемо такий алгоритм:

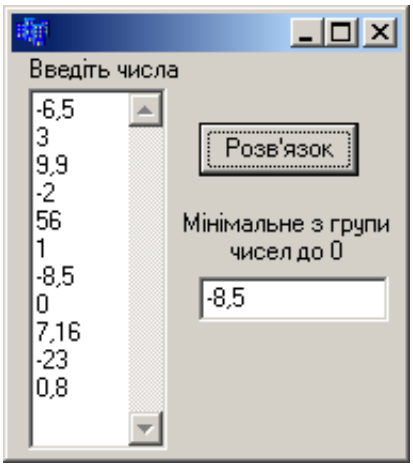
1) поточному мінімуму присвоїти перше число;

2) пройти всіма числами і, якщо зустрінеться число, більше за поточний мінімум, присвоїти це число поточному мінімуму.

Після того, як всі числа переглянуто, поточний мінімум можна вважати за остаточний мінімум.

Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float min, x;
 int i=0, n=Memo1->Lines->Count;
 x= StrToFloat(Memo1->Lines->Strings[0]);
 // Вважатимемо, що перше число – мінімальне
 min = x;
 while(x != 0 && i < n)
 {i++;
  x=StrToFloat(Memo1->Lines->Strings[i]);
  if(x < min ) // Якщо віднайшли менше число,
    min = x;    // записати це число у min
 }
Edit1->Text =FloatToStr(min);
}
```



Приклад 4.59 Зчитати числа з Memo, допоки не зустрінеться 0. Віднайти мінімальне додатне число.

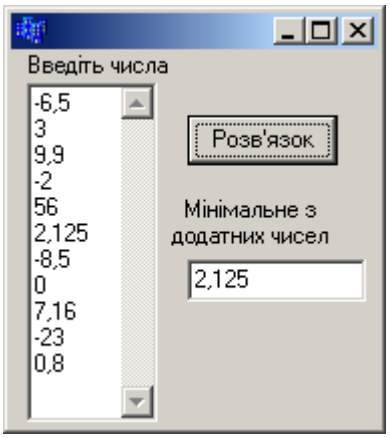
Розв'язок. Відмінність цього прикладу від попереднього полягає в тому, що мінімальне число треба віднайти з-посеред додатних чисел і присвоїти поточному мінімуму перше число не можна, оскільки воно може бути від'ємним.

Розглянемо два способи розв'язання цього завдання.

Перший спосіб

Спочатку пройдемо по числах в Memo1 і відшукаємо перше додатне число (за допомогою циклу do-while). Після цього пройдемо по решті чисел і віднайдемо мінімальне з додатних.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float min, x;
  int i=0, n=Memo1->Lines->Count;
  do
  { x=StrToFloat(Memo1->Lines->Strings[i]);
    i++;
  }
  while(x<=0 && i<n);
  min = x;
  while(x != 0 && i<n)
  { x=StrToFloat(Memo1->Lines->Strings[i]);
    if(x>0 && x<min) min = x;
    i++;
  }
  Edit1->Text=FloatToStr(min);
 }
```



Другий спосіб

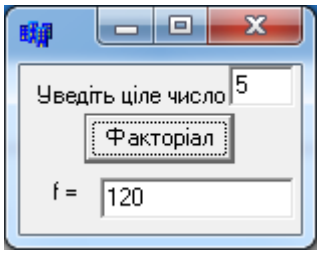
Спочатку присвоюємо змінній min значення 0. Зчитуємо числа з Memo. Якщо число є від'ємне, зігноруємо його. Якщо число є додатне та min=0, це означає, що ще не зустрічалось додатних чисел, це – перше і його слід присвоїти min. Якщо число є додатне і min≠0, це означає, що додатні числа вже зустрічались і min вже має певне додатне значення (тимчасовий мінімум). В цьому разі порівнюємо число з min і, якщо воно є менше за min, присвоюємо його.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float min=0, x; int i=0,n=Memo1->Lines->Count;
 do
 { x=StrToFloat(Memo1->Lines->Strings[i]);
   if(x>0)
    if(min==0 || x<min) min = x;
   i++;
 }
 while(x != 0 && i<n);
 Edit1->Text=FloatToStr(min);
}
```

Приклад 4.60 Написати програму обчислення факторіала введеного цілого числа за допомогою оператора `while`.

Розв'язок. У порівнянні з обчисленням факторіала оператором `for` (див. п. 4.4.1) чи за допомогою рекурсивної функції (див. підрозд. 8.5) наведене розв'язання є самим лаконічним і елегантним.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int f = 1, n;
  n = Edit1->Text.ToInt();
  while ( n >= 1 ) f *= n--;
  Edit2-> Text = IntToStr(f);
}
```



4.4.5 Оператори переривання виконання

Для завчасного переривання повторювань операторів циклу будь-якого типу в тілі циклу можна застосовувати оператор **`break`**, який перериває виконування оператора, в якому розміщено цей `break`, і передає керування на наступний оператор.

Наведемо приклад застосування оператора `break` на програмному фрагменті обчислення у змінній `k` кількості різних (неоднакових) елементів для масиву `a` з `m` елементів. Кожен черговий елемент `a[i]` порівнюється з наступними елементами масиву. Якщо він не збігається з жодним із цих елементів, лічильник `k` збільшується на одиницю. Інакше, внутрішній цикл переривається оператором `break` і починається нова ітерація зовнішнього циклу.

```
int k=1;
for(int i=0;i<m-1;i++)
{ for(int j=i+1;j<m;j++)
   if(a[i]==a[j]) break;
   if(j==m)k++;
}
```

Зауважимо, що всередині вкладених операторів `do-while`, `for`, `while` чи `switch` оператор `break` завершує лише самий внутрішній з цих операторів, тому `break` неможна використовувати для виходу з декількох вкладених циклів. Навіть повторення підряд двох операторів `break` не забезпечить вихід з вкладених циклів у такому фрагменті:

```
for (i=0; i<100; i++)
for (j=0; j<100; j++)
{ . . .
if (a[i][j]<0) { break; break; }
  . . .
}
```

Так після виконання умови `a[i][j]<0` відбудеться вихід лише з внутрішнього циклу по змінній `j`, а виконання зовнішнього циклу по змінній `i` продовжиться, незважаючи на те, що оператор `break` записано двічі.

По своїй суті оператор `break` є оператором переходу і оператори записанні після нього виконуватись не будуть, якщо тільки їм не буде передано керування за допомогою інших операторів переходів.

Для виходу з обох циклів можна застосувати додаткову змінну, яка буде набувати ненульового значення лише за потреби виходу із зовнішнього циклу:

```
for (i=0; i<100; i++)
{ for (br2=j=0; j<100; j++)
  { . . .
    if ( a[i][j]<0 ) { br2=1; break;}
    . . .
  }
  if (br2) break;
  . . .
}
```

У наведеному прикладі відразу після завершення внутрішнього циклу перевірятиметься умова завершення зовнішнього циклу.

Крім того, передавати керування за межі вкладеної структури, можна використовуючи оператори **`return`** (вихід з поточної функції) (див. підрозд. 8.1) та **`goto`** (безумовний перехід) (див. п. 4.3.2).

Для переходу до наступної ітерації циклу можна застосовувати оператор **`continue`.** Цей оператор, подібно до оператора `break`, використовується лише всередині операторів циклу `for`, `while`, `do-while`, але, на відміну від останнього, оператор `continue` не припиняє подальше виконання циклу, а переходить до чергової ітерації того циклу, у тілі якого він виявився. Він як би імітує безумовний перехід на кінцевий оператор циклу, але не за межі самого циклу.

Оператор `continue`, так само як і оператор `break`, перериває найглибинний з циклів.

```
for(int a = 1, b = 0;  a < 100;  b += a, a++)
{  if(b % 2) continue;
   . . .                    // Опрацювання парних сум
}
```

У вищенаведеному прикладі оператор `continue` передає керування на чергову ітерацію циклу `for` за умови, коли сума чисел від `1` до `a` є непарною, не виконуючи операторів опрацювання парних сум.

Наведемо ще один програмний фрагмент з застосуванням оператора `continue` для обчислення суми додатних елементів масиву `a` з `m` елементів:

```
int s=0;
for(int i=0; i<m; i++)
{ if(a[i]<=0) continue; // Пропустити елемент
  s+=a[i];
}
```

У наведеному прикладі оператор `continue` використовується для пропускання від'ємних елементів масиву, підсумовуючи лише додатні.

Питання та завдання для самоконтролю

1) Який процес називають лінійним?
2) Який процес називають розгалуженим?
3) Які оператори в С++ використовуються для організації розгалужень?
4) Перелічіть базові логічні операції.
5) Вкажіть послідовність дій при виконанні оператора
`bool w=2*5<=17%3;`.
6) Наведіть результат виконання логічного виразу `(7+3 > 16-4*3)`.
7) Обчисліть логічний вираз `(-3 >= 5) || (7 < 9) && (0 < 3)`.
8) Вкажіть значення `w` після виконання оператора `bool w=2*5<=17%3;`.
9) Засобами умовного оператора `if` скороченої і повної форми, а також умовної операції `?:` запишіть три варіанти обчислення

$$y=\begin{cases}\sin x^2 & \text{за } x>0.5;\\ \cos^2 x & \text{за } x\le 0.5.\end{cases}$$

10) Назвіть оператор без помилок:

а) `if(x<=6)y=2*x; else y=cos(x);`
б) `if y<=x then y:=exp(x*y);`
в) `if(a<>0) if(b<>0) y=2*x;`
г) `if(x>0)y=ln(x) else y=x;`

11) Вкажіть значення `x` після виконання фрагментів програми:

а)
```
float x=1.5;
if(x<=0.5) x=7.7;
```
б)
```
float x=1.5;
if(x<=0.5) x=7.7; else x=3;
```

12) Наведіть значення змінної `z` після виконання операторів:
```
x=2.5;
if(x>=0.5) z=7.7; else z=5.5;
```
13) Вкажіть значення змінної `f` після виконання операторів
```
f = 1; n = 3; i = 2;
M1: if (i > n)  goto PP;
    f = f * i; i++; goto M1;
PP : ;
```
14) Вкажіть значення *y* після виконання фрагментів програми:

а)
```
float y=0; int n=1;
switch (n)
{case 1: { y=n/4.; break; }
 case 2: { y=n*n; break; }
 case 3: { y=n; break;   }}
```
б)
```
float y=0; int n=3;
switch (n)
{case 1: { y=n/4.; }
 case 3: { y=n*n; }
 case 5: { y=n+1; }}
```
б)
```
float y=0; int n=4;
switch (n)
{case 2: { y=n/4.; break; }
 case 5: { y=n*n; break; }
 case 9: { y=n; break;   }}
```
в)
```
float y=0; int n=1;
switch (n)
{case 1: { y=n/4; }
 case 3: { y=n*n; }
 case 5: { y=n+1; }}
```
15) У наведеному фрагменті програми
```
int  nom = pow(2, 3);
 switch (nom)
 { case  2 : y=d;     break;
   case  8 : y=d*exp(x); break;
   case 10 : y=d*x;   break;          }
```
оператор `switch` обчислюватиме вираз … (запишіть увесь вираз).

16) Який процес називають циклічним?

17) Які оператори циклу використовуються е мові С++?

18) Скільки разів виконуватиметься оператор у циклі, тобто вкажіть значення s:

```
for(int k = -1, s=0; k <= 5; k++) s++;
```

19) Вкажіть помилки в таких фрагментах програм:

а)
```
int k, m=2, n=3;
  for(k=1; k<=n; k++)n=n+m;
```

б)
```
int n=-7, m=2;
  for(int k=n; k<=m; k--)k++;
```

20) Вкажіть значення m після виконання фрагментів програми:

а)
```
int k, m=1;
  for(k=1; k<=5; k++)m++;
```

б)
```
int m=1, n=5;
  for(int k=n; k>=1; k--)m*=k;
```

21) Назвіть правильну, на Ваш погляд, послідовність номерів, для запису елементів оператора циклу while:

а) логічний вираз умови;

б) оператори тіла циклу;

в) while.

22) Вкажіть значення n після виконання фрагментів програми:

а)
```
int k=0, n=17;
  while(k<7){ k++;  n--; }
```

б)
```
int m=1, n=1;
  while(m<5){ m+=2; n=n*m; }
```

23) Якими є структура і порядок виконання оператора циклу do-while?

24) Вкажіть значення y після виконання фрагментів програми:

а)
```
int i=1, y=1;
  do {y*=i++;} while(y<7);
```

б)
```
int k=1; float y=0;
do{k+=2;y+=1./k;}while(k<5);
```

25) Запишіть трьома операторами циклу варіанти обчислення $S=\sum_{i=1}^{6} i^2$.

26) Вкажіть значення s після виконання операторів

```
s=0.5; i=0;
while(i<5) i++;
s+=1.0/i;
```

27) Назвіть номер фрагмента програми з вкладеним циклом

а)
```
for(k=1;k<=10;k++)
  p=k;
 for(j=1;j<=5;j++)
  s+= p*j;
```

б)
```
for(k=1;k<=10;k++)
 { p=k;
   for(j=1;j<=5;j++)
    s+= p*j;  }
```

в)
```
for(k=1;k<=10;k++)
 { p=k; }
 for(j=1;j<=5;j++)
 { s+= p*j;  }
```

28) Якого значення набуде змінна s після виконання операторів

```
for(s=0,k=1; k<=3; k++)
  for(j=1; j<=k; j++)  s+=j;
```

29) Вкажіть значення s після виконання операторів

```
for(s=0,k=1; k<=2; k++)
{ m=k;
  do { m++; s+=m; } while( m <= 2);
 }
```

30) Назвіть оператори, використовувані для завчасного переривання повторювань циклу.

Розділ 5

Масиви в С++

5.1 Поняття масиву

Змінні, якими ми оперували до цих пір, могли зберігати лише одне значення одночасно. За присвоювання їм іншого значення попереднє втрачалося. Часто виникає потреба зберігати велику кількість однотипних значень, які мають опрацьовуватись однаково. Приміром, екзаменаційні оцінки студентів, які слід вводити, виводити, аналізувати (порівнювати з “2” чи “5”), змінювати за потреби тощо. Такі однотипні значення є сенс зберігати в одній змінній, перенумерувати їх усередині цієї змінної, надати доступ до цих значень за номером і опрацьовувати ці значення у циклі (номер значення збігатиметься з параметром циклу).

Масив у програмуванні – це упорядкована сукупність однотипних елементів. Масиви широко застосовуються для зберігання і опрацювання однорідної інформації, приміром таблиць, векторів, матриць, коефіцієнтів рівнянь тощо.

Кожен елемент масиву однозначно можна визначити за ім’ям масиву та індексами. *Ім’я масиву* (ідентифікатор) добирають за тими самими правилами, що й для змінних. *Індекси* визначають місцезнаходження елемента в масиві. Приміром, елементи вектора мають один індекс – номер по порядку; елементи матриць чи таблиць мають по два індекси: перший означає номер рядка, другий – номер стовпчика. Кількість індексів визначає вимірність масиву. Приміром, *вектори* у програмах – це одновимірні масиви, *матриці* – двовимірні.

Індексами можуть бути лише змінні, константи чи вирази цілого типу. Значення індексів записують після імені масиву в квадратних дужках. При оголошенні масивів у квадратних дужках зазначається кількість елементів, а нумерація елементів завжди розпочинається з нуля.

Відмінності масиву від звичайних змінних:

- ✓ спільне ім’я для всіх значень;
- ✓ доступ до конкретного значення за його номером (індексом);
- ✓ можливість опрацювання у циклі.

5.2 Одновимірні масиви

5.2.1 Оголошення одновимірних масивів

Одновимірний масив (вектор) оголошується у програмі в такий спосіб:

<*тип_даних*> <*ім’я_масиву*> [<*розмір_масиву*>];

Тип_даних задає тип елементів масиву. Елементами масиву не можуть бути функції й елементи типу `void`. *Розмір_масиву* у квадратних дужках задає кількість елементів масиву. На відміну від інших мов, у С++ не перевіряється вихід за межі масиву, тому, щоб уникнути помилок у програмі, слід стежити за

розмірністю оголошених масивів. Значення *розмір_масиву* при оголошенні масиву може бути не вказано в таких випадках:

- ✓ при оголошенні масив ініціалізується;
- ✓ масив оголошено як формальний параметр функції (докладніше див. розд. 8);
- ✓ масив оголошено як посилання на масив, явно визначений в іншому модулі.

Використовуючи ім'я масиву та індекс, можна звертатися до елементів масиву:

<*ім'я_масиву*> [<*значення_ індексу*>]

Значення індексів мають перебувати в діапазоні від нуля до величини, на одиницю меншу за розмір масиву, визначений при його оголошенні, оскільки в C++ нумерація індексів розпочинається з нуля.

Наприклад,

```
int A[10];
```

оголошує масив з ім'ям `A`, який містить 10 цілих чисел; при цьому виділяє і закріплює за цим масивом оперативну пам'ять для усіх 10-ти елементів відповідного типу (`int` – 4 байти), тобто 40 байтів, у такий спосіб:

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Отже, при оголошенні масиву виділяється пам'ять, потрібна для розташування усіх його елементів. Елементи масиву з першого до останнього запам'ятовуються у послідовно зростаючих адресах пам'яті. Поміж елементами масиву в пам'яті проміжків немає. Елементи масиву запам'ятовуються один за одним поелементно.

Зауважте на звертання у програмі до елементів масиву: `A[0]` – перший елемент, `A[1]` – другий, `A[9]` – останній.

Так, для розміщення елементів одновимірного масиву `int B[5]` виділяється по 4 байти під кожен з 5-ти елементів масиву – тобто усього 20 байт:

Елементи	B[0]				B[1]				B[2]				B[3]				B[4]			
Байти	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Для отримання доступу до `i`-го елемента масиву `B`, можна написати `B[i]`, де `i` – змінна циклу, яка може набувати значення від 0 до 3. При цьому величина `i` перемножується на розмір типу `int` і являє собою адресу `i`-го елемента масиву `B` від його початку, після чого здійснюється вибір елемента масиву `B` за сформованою адресою.

Змінна (ідентифікатор), оголошена як масив, по суті є вказівником на перший елемент масиву (див. розд. 6). Слід зауважити, що у перебігу виконування програми адресу масиву не може бути змінено, хоча значення його елементів можуть змінюватись.

Ось кілька прикладів оголошення масивів:

```
char  N[20];
int  grades[125];
```

```
float  mass[30];
double v[1500];
```

Перший з масивів `N` містить 20 символів. Звертання до елементів масиву може бути таким: `N[0]`, `N[1]`, ..., `N[19]`.

Другий масив `grades` містить 125 цілих чисел. Звертання до елементів такого масиву може бути таким: `grades[0]`, `grades[1]`, ..., `grades[124]`.

Третій масив `mass` містить 30 дійсних чисел. Звертання до елементів масиву може бути таким: `mass[0]`, `mass[1]`, ..., `mass[29]`.

Четвертий масив `v` містить 1500 дійсних чисел з подвійною точністю. Звертання до елементів такого масиву може бути таким: `v[0]`, `v[1]`, ..., `v[1499]`.

При оголошуванні масивів можна елементам масиву (необов'язково всім) присвоювати початкові значення, які у подальшому в програмі може бути змінено. Якщо реальна кількість ініціалізованих значень є менше за розмірність масиву, то решта елементів масиву набуває значення 0.

Наприклад,

```
int a[5]={9,33,-23,8,1};      //a[0]=9, a[1]=33, a[2]=–23, a[3]=8, a[4]=1
float b[10] = {1.5, -3.8, 10};
                        //  b[0]=1.5, b[1]=-3.8, b[2]=10, b[3]=b[4]=...=b[9]=0
char s[5]="Фото";       //  s[0]= 'Ф', s[1]='o', s[2]='m', s[3]= 'o', s[4]= '\0'
char code[]="abc";      //code[0]='a', code[1]= 'b', code[2]= 'c', code[3]='\0'
```

В останньому прикладі ініціалізується `code` як масив символів з чотирьох елементів. Четвертим елементом є символ `'\0'`, який завершує усі рядки символів. Якщо рядок є коротше за оголошений розмір масиву символів, то решта елементів масиву ініціалізується нулем (символом `'\0'`).

Масиви в програмах можна оголошувати також *зі створенням типу користувача* (докладніше див. розд. 11):

`typedef` <*тип_даних*> <*ім'я_типу*> [<*розмір_масиву*>];
<*ім'я_типу*> <*ім'я_масиву*>;

Наприклад, створимо тип з ім'ям `mass` як масив з 10-ти додатних цілих чисел і оголосимо два масиви – `a` та `b` – створеного типу:

```
typedef unsigned short mass[10];
mass a, b;
```

Для оголошування масивів можна використовувати *типізовані константи-масиви,* які дозволяють водночас і оголосити масив, і задати його значення як константи:

`const` <*тип_даних*> <*ім'я_масиву*> [<*розмір_масиву*>] =
{<*значення_елементів_масиву*>};

Слід пам'ятати, що змінювати значення елементів констант-масивів не припустимо.

Приклади створювання констант-масивів:

```
const int arr[5]={9,3,-7,0,123}; //масив з 5-ти цілих чисел
const float f[5]={1.5,6,8,7.4,-1.125}; //масив з 5-ти дійсних чисел
const C[5]={15,-6,546};          //масив з 5-ти цілих чисел типу int,
                        // де C[0]=15, C[1]= –6, C[2]=546, C[3]=0 і C[4]=0
```

5.2.2 Введення-виведення одновимірних масивів

Виведення елементів одновимірних масивів до різних компонентів

Виводити значення одновимірних масивів можна до файла чи на форму, використовуючи різноманітні компоненти C++ Builder. При цьому виводити значення елементів масивів можна лише поелементно, для чого слід організовувати цикли зі зміненням значень індексу зазначеного масиву. Організацію виведення масивів до файла буде розглянуто у розд. 11. Тут розглянемо організацію виведення одновимірних масивів на форму за допомогою компонентів `Edit`, `Label`, `Memo`, `ListBox` та функції `ShowMessage()`.

У подальших прикладах буде використано такі змінні:

```
float A[10];
int i; AnsiString st;
```

Виведення до компонента Edit елементів одновимірних масивів можна організовувати, відокремлюючи елементи пробілами (" ") чи іншими символами.

Приклад фрагмента програми виведення масиву `A`:

```
st = "" ;              // Очищення рядка st
for(i=0; i<=9; i++)    // Початок циклу за індексами масиву
                       // для накопичення в рядку значень масиву
 {  st += FormatFloat("0.00", A[i])+"  ";  }
Edit1->Text=st;  // Виведення до компонента Edit1 сформованого рядка
```

Ще один спосіб виведення масиву `A` до `Edit1` (кожний елемент виводиться до `Edit1` послідовно):

```
for(i=0; i<=9; i++)
  Edit1->Text = Edit1->Tex+FormatFloat("0.00", A[i])+"  ";
```

Виводити до компонента Label можна одновимірний масив, відокремлюючи його елементи пробілами (" ") чи символами переходу до нового рядка (`'\n'`). Виведення одновимірного масиву у рядок організовують за тими самими правилами, що і до компонента `Edit`, лише в програмі замість `Edit1->Text` слід записати `Label1->Caption` (наприклад `Label1->Caption=st;`). Для виведення одновимірного масиву у стовпчик слід властивість `WordWrap` компонента `Label` встановити в `true` замість `false`, що встановлено за замовчуванням.

Виведення одновимірного масиву у вікно повідомлень за допомогою функції `ShowMessage()` організовують так само, як і в попередніх прикладах, лише замість оператора присвоювання слід записати оператор виклику функції. Наприклад, замість оператора

```
Edit1->Text = st;
```

слід записати

```
ShowMessage(st);
```

Виводити до багаторядкового компонента Memo можна одновимірні масиви з якою завгодно кількістю елементів. В такому разі є доцільним використання смуги прокручування (надати властивості `ScrollBars` значення `ssBoth` чи `ssVertical`).

Приклад фрагмента програми виведення масиву A (у стовпчик):

```
Memo1.Clear();          // Очищення компонента.
for(i=0; i<=9; i++)     // Початок циклу за індексами масиву
                        // для почергового виведення елементів масиву.
Memo1->Lines->Add(FormatFloat("0.00", A[i]));
```

Виведення масивів до компонента ListBox організовують так само, як і для компонента Memo, лише замість Memo слід записати компонент ListBox.

Наприклад, замість оператора

```
Memo1->Lines->Add(FormatFloat("0.00",A[i]));
```

слід записати

```
ListBox1->Items->Add(FormatFloat("0.00",A[i]));
```

Як і при виведенні масивів, при їхньому введенні слід організовувати цикли змінювання зі зміною значення індексу. Розглянемо введення елементів масивів, використовуючи компоненти Memo та ListBox.

Введення елементів одновимірних масивів з різних компонентів

За допомогою компонента Memo можна вводити масиви як при виконуванні програми, так і при конструюванні форми проекту програми через вікно властивості Lines (для переходу до нового рядка при введенні значень треба натиснути клавішу <Enter>).

Приклад фрагмента програми введення значень елементів одновимірного масиву A (в кожному рядку по одному числу):

```
for(i=0;i<=9;i++) A[i]=StrToFloat(Memo1->Lines->Strings[i]);
```

За допомогою компонента ListBox можна вводити масиви так само, як і з компонентом Memo, лише замість властивості Lines використовувати властивість Items.

Компонент StringGrid, який також можна використовувати для введення-виведення одновимірних масивів, буде розглянуто в підрозд. 5.3.

Введення-виведення масиву у консолі

Виведення масиву у консолі за допомогою **cout** у стовпчик:

```
for(int i=0; i<n; i++) cout << a[i] << endl;
```

у рядок через пробіл:

```
for(int i=0; i<n; i++) cout << a[i] << "  ";
cout << endl;
```

Введення масиву за допомогою **cin**:

```
for (int i=0; i<n; i++)  cin >> a[i];
```

При цьому числа при введенні можуть бути написані як у стовпчик, так і у рядок через пробіл.

Для введення-виведення символьних масивів у консолі використовуються функції **gets()** і **puts()**:

```
char s[50];
```

```
gets(s);      // Введення символьного масиву s
puts(s);      // Виведення символьного масиву s
```

Ще раз звернімося до питання стосовно доцільності використання у програмах масивів замість окремих змінних. Порівняймо застосування масиву з 7-ми цілих чисел і 7-ми окремих змінних:

Масив з 7-ми цілих чисел	7 окремих цілих змінних
Оголошення	
`int a[7];`	`int a, b, c, d, e, f, g;`
Введення до `Memo1`	

`for(int i=0; i<7; i++)`
`a[i]=StrToInt(Memo1->Lines->`
`Strings[i]);`
	`a = StrToInt(Memo1->Lines->Strings[0]);`
	`b = StrToInt(Memo1->Lines->Strings[1]);`
	`c = StrToInt(Memo1->Lines->Strings[2]);`
	`d = StrToInt(Memo1->Lines->Strings[3]);`
	`e = StrToInt(Memo1->Lines->Strings[4]);`
	`f = StrToInt(Memo1->Lines->Strings[5]);`
	`g = StrToInt(Memo1->Lines->Strings[6]);`

Обчислення суми додатних елементів	

`float s=0;`
`for(int i=0; i<7; i++)`
`  if (a[i]>0) s+=a[i];`
	`if (a>0) s=a;`
	`if (b>0) s+=b;`
	`if (c>0) s+=c;`
	`if (d>0) s+=d;`
	`if (e>0) s+=e;`
	`if (f>0) s+=f;`
	`if (g>0) s+=g;`

На практиці зазвичай у масивах зберігається набагато більше за 7 елементів і використовування окремих змінних стає, м'яко кажучи, недоцільним.

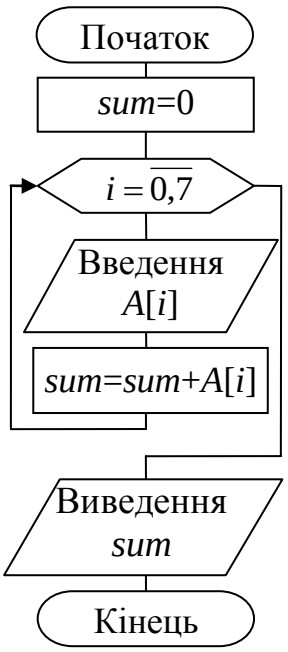
5.2.3 Програмування базових алгоритмів опрацювання одновимірних масивів

Опрацювання масиву полягає у виконанні операцій над його елементами. Окрім введення-виведення масивів існує перелік найпоширеніших базових алгоритмів опрацювання масивів:

- ✓ обчислення узагальнювальних характеристик (сум, добутків і кількості елементів);
- ✓ пошук максимального чи мінімального елемента;
- ✓ пошук заданих елементів;
- ✓ переставляння елементів;
- ✓ упорядкування масивів.

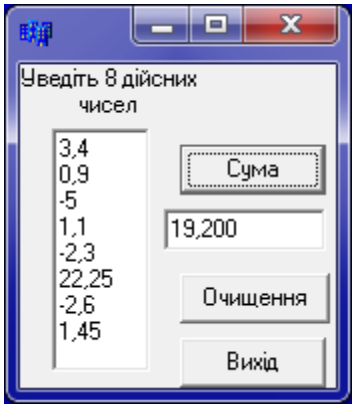
Базові алгоритми опрацювання масивів можуть бути складовими блоками при розв'язуванні більш складних задач. А тому організовувати їх доцільно у вигляді окремих функцій (див. п. 5.2.4 та розд. 8).

Приклад 5.1 Розробити схему алгоритму та проект програми для введення одновимірного масиву з 8-ми дійсних чисел та обчислення суми усіх елементів масиву.



Текст програми:

```
// Сума
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float A[8], sum = 0;
  for(int i = 0; i < 8; i++)
   { A[i]=StrToFloat(Memo1->Lines->Strings[i]);
     sum += A[i];
   }
  Edit1->Text = FormatFloat("0.000", sum);
}
//---------------------------------------
//Очищення

void __fastcall TForm1::Button3Click(TObject
*Sender)
{  Memo1->Clear();
   Edit1->Clear();
}
//---------------------------------------
void __fastcall TForm1::Button2Click(TObject
                                     *Sender) //Вихід
{  Close();
}
```



Приклад 5.2 Скласти консольну програму, яка вводить масив з 12-ти цілих чисел і обчислює кількість непарних елементів з проміжку (–10, 30).

Текст програми:

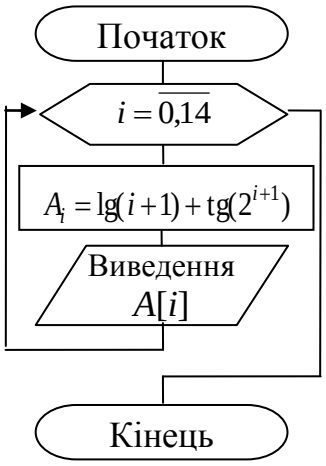
```
#include <conio.h>
#include <iostream.h>
//---------------------------------------------------------
int main(int argc, char* argv[])
{ int a[12],i,k=0;
  for(i=0; i<12; i++)
  {  cin >> a[i];
     if((a[i]%2 != 0) && (a[i] > -10) && (a[i] < 30))  k++;
  }
  cout << "k=" << a[i] << endl;
  getch();  return 0;
}
```

Результати виконання програми:

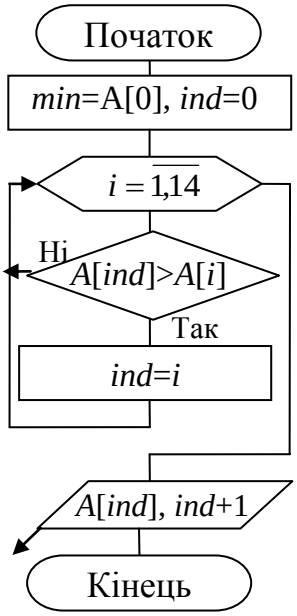
```
11 -12 0 5 15 122 23 9 3 6 12 10
 k = 6
```

Приклад 5.3 Скласти схему алгоритму і проект програми для обчислення елементів одновимірного масиву з 15-ти елементів за формулою $A_i = \lg(i) + \mathrm{tg}(2^i)$, де i = 1, 2, ..., 15, та їхнього виведення на форму, а також визначення мінімального елемента та його порядкового номера.

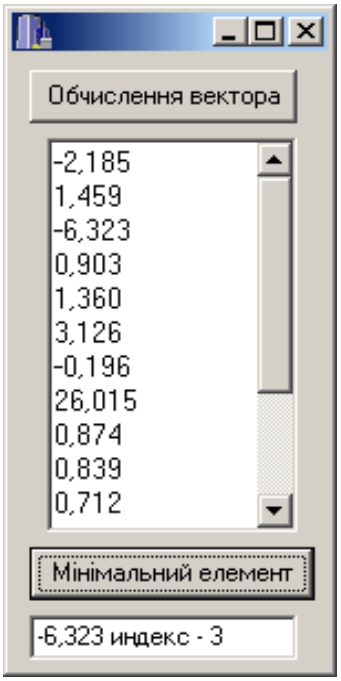
Розв'язок. Схеми алгоритмів програми та приклад вигляду форми з результатами роботи наведено нижче.



Блок-схема для кнопки "Обчислення вектора"



Блок-схема для кнопки "Мінімальний елемент"



У цій програмі змінні `A[15]` та `i` оголошено глобально перед функціями для подій `onClick`. Це зроблено для того, щоб значення елементів масиву, набуті в одній функції, були доступними для опрацювання і в другій функції. Але слід звернути увагу на те, що оголошувати без нагальної потреби глобальні змінні вважається за неоптимальний стиль програмування.

Розглянемо алгоритм пошуку мінімального елемента масиву. Припустімо, що на столі лежить купа резюме, кожне з яких, окрім іншої інформації про кандидата на посаду, містить і його рік народження. Треба віднайти найстаршого з усіх кандидатів, тобто резюме з найменшим роком народження.

Беремо до рук перше резюме. Оскільки невідомо, що міститься у решті резюме, відкладаємо перше і вважаємо його власника за найстаршого.

Одне за одним переглядаємо решту резюме. Якщо натрапляємо на резюме, в якому подано рік менше за рік у відкладеному резюме, то відкладаємо його (попереднє нас вже не цікавить). Коли всі резюме переглянуто, виявляється, що відкладене резюме належить кандидатові з найменшим роком народження.

У вигляді словесного алгоритму пошук мінімального елемента масиву і його індексу можна записати в такий спосіб:

1) вважаємо перше число за найменше. Для цього присвоюємо перший елемент масиву `A[0]` у змінну `min`, а змінній `ind` – значення 0 (тобто індекс першого елемента масиву);

2) у циклі переглядаємо всі елементи. Якщо зустрічаємо число, яке є менше за min, запам'ятовуємо це число у min, а його індекс – в ind.

Коли цикл завершиться, у змінній min буде зберігатися найменший елемент масиву, а його індекс – у змінній ind. Оскільки нумерація індексів масиву розпочинається з 0, а не з 1, як у повсякденному житті, виводиться значення індексу, збільшене на 1.

Текст програми:

```
#include <math.h>
double A[15];  int i;
//---------------------------------------------------
// Обчислення елементів вектора
void__fastcall TForm1:: Button1Click(TObject *Sender)
{ Memo1->Clear();
  for(i=0; i<15; i++)
  {  A[i]=log10(i+1)+tan(pow(2.,i+1));
     Memo1->Lines->Add(FormatFloat("0.000", A[i]));
  }
}
//--------------------------------
// Мінімальний елемент
void __fastcall TForm1::Button2Click(TObject *Sender)
{ double min=A[0]; int ind=0;
  for(i=1; i<15; i++)
    if(min > A[i]) { min=A[i]; ind=i; }
  Edit1->Text=FormatFloat("0.000", A[ind]) + " індекс - " +
                                              IntToStr(ind+1);
}
```

Якщо треба віднайти лише значення мінімального елемента масиву, то програма для Button2Click матиме спрощений вигляд:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ double min=A[0];
  for(i=1; i<15; i++)
    if(min > A[i]) min=A[i];
  Edit1->Text=FormatFloat("0.000", min);
}
```

Можна відшукати лише індекс мінімального елемента, а потім за його допомогою визначити мінімальний елемент. У цьому способі мінімальним буде елемент з віднайденим індексом: A[ind]. Цей спосіб реалізовано у такій програмі.

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ int ind=0;
  for(i=1; i<15; i++)
   if(A[ind] > A[i]) ind=i;
  Edit1->Text=FormatFloat("0.000", A[ind]) + " індекс - " +
                                              IntToStr(ind+1);
 }
```

Приклад 5.4 Розробити консольний додаток для заповнення масиву з 20-ти елементів за формулою $a_i = (-1)^i \frac{\sin(i^2)}{\sin(i+1)}$ та визначити найбільший від'ємний елемент масиву.

Розв'язок. Оскільки невідомо, чи буде перший елемент масиву від'ємним, не можна присвоїти початкове значення змінній max. Присвоїмо їй значення 0. Вважатимемо, що, якщо зустріли від'ємний елемент і змінна max дорівнює 0, то це буде перший від'ємний елемент і його слід присвоїти змінній max у якості початкового значення. Якщо змінна max вже набула певного значення, порівнюємо її з новим від'ємним елементом, і якщо новий елемент перевищує max, записуємо його до max.

Текст програми

```
#include <conio.h>
#include <math.h>
#include <stdio.h>
int main(int argc, char* argv[])
{double a[20], max=0;  int i;
 for(i=0; i<20; i++)
 { a[i]=pow(-1,i)*sin(i*i)/sin(i+1);
   printf("%5.2f ",a[i]);
 }
for(i=0; i<20; i++)
  if(a[i]<0)                          //Якщо віднайшли від'ємний елемент і,
   if(max==0 || max<0 && max<a[i])    //якщо він є перший чи більший за max,
      max=a[i];                       //присвоюємо його max
 printf("\n max = %5.2f",max);
 getch();  return 0;
}
```

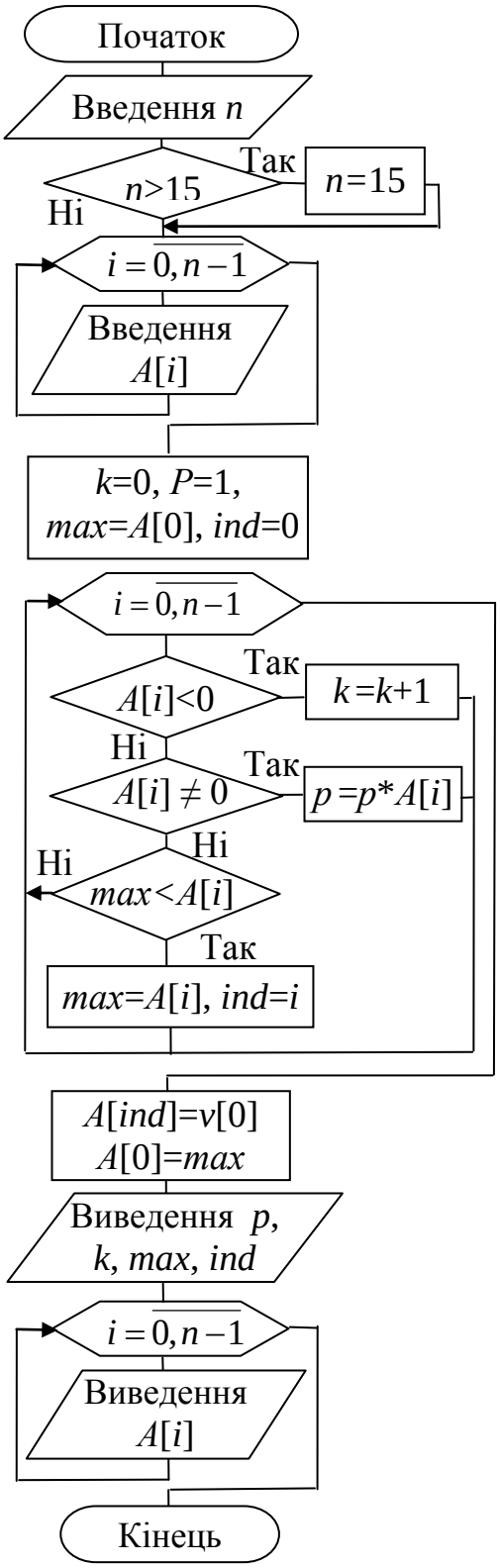
Результати виконання програми:

```
 0.00
-0.93
-10.73
 1.63
 1.20
-2.37
-9.06
 6.75
17.86
-10.42
 5.06
20.48
-14.02
 7.90
20.23
-48.46
16.63
-0.60
-48.53
-5.81

max = -0.60
```

Приклад 5.5 Розробити проект програми для введення одновимірного масиву до 15-ти елементів та обчислення:

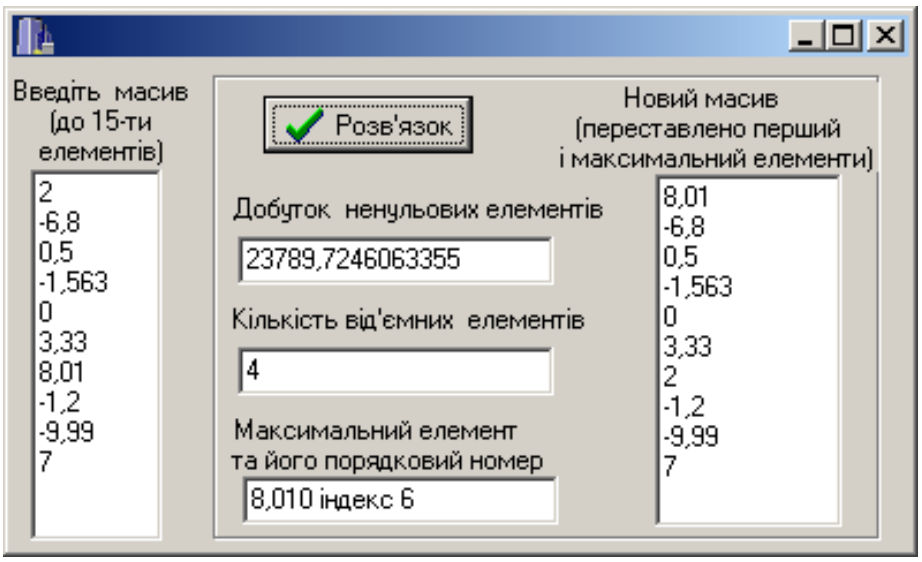
- ✓ добутку ненульових елементів;
- ✓ кількості від'ємних елементів;
- ✓ максимального елемента з переставлянням його з першим елементом вектора.

Розв'язок. Зверніть увагу, що чисел може бути введено менш за 15, але оголосити масив слід з максимально можливою кількістю елементів (15). У програмі при опрацюванні масиву індекс у циклі змінюється від 0 до n, де n – кількість реально введених чисел. Якщо введено менше за 15 чисел, зчитуються всі числа, які було введено. Якщо чисел є більше за 15, зайвими числами нехтують.



Текст програми:

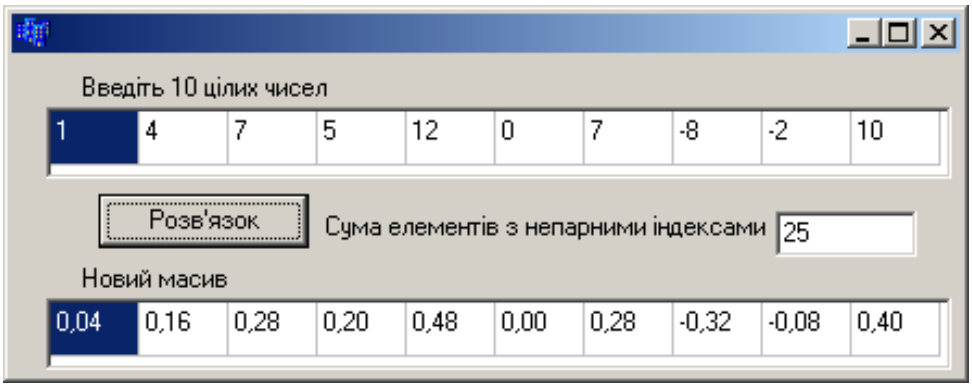
```
void __fastcall TForm1::
          BitBtn1Click (TObject *Sender)
{double A[15]; int i, n;
 n = StrToInt(Memo1->Lines->Count);
 if (n>15) n=15;
 for (i=0; i<n; i++)
  A[i]=StrToFloat(Memo1->Lines->Strings[i]);
 int k=0, ind=0;
 double p=1, max=A[0];
 for (i=0; i<n; i++)
  { //Кількість від'ємних елементів
    if (A[i] < 0) k++;
    //Добуток ненульових елементів
    if (A[i] != 0) p *= A[i];
    //Максимальний елемент та його індекс
    if (A[i] > max)
     { max=A[i];  ind=i;
     }
  }
 //Поміняти місцями максимальний елемент
 //з першим елементом вектора
 A[ind]=A[0];  A[0]=max;
 Edit1->Text=FloatToStr(p);
 Edit2->Text=IntToStr(k);
 Edit3->Text=FormatFloat("0.000", max)+
                  " індекс "+IntToStr(ind);
 Memo2->Clear();
 for(i=0; i<n; i++)
   Memo2->Lines->Add(A[i]);
}
```



Приклад 5.6 Увести масив з 10-ти цілих чисел. Створити новий масив, елементи якого обчислити, поділивши кожен з елементів початкового масиву на суму його елементів з непарними індексами.

Розв'язок. Щоб створити новий масив, спочатку обчислимо суму елементів з непарними індексами. Після цього послідовно поділимо всі елементи масиву на обчислену суму і присвоїмо результат відповідному елементові нового масиву. Після ділення елементи нового масиву матимуть тип `float`.

Будемо вводити початковий масив і виводити новий масив у компонент `StringGrid`, який міститься на панелі `Additional` (докладніше про цей компонент див. у п. 5.3). Цей компонент зазвичай використовується для таблиць. Він складається з комірок, кожна з яких є розміщена на перетині рядка і стовпчика і має два індекси. У цьому прикладі компоненти `StringGrid` матимуть імена (властивість `Name`) `SG1` та `SG2` і міститимуть по одному рядку (властивість `RowCount`=1) і 10 стовпчиків (властивість `ColCount`=10). Щоби надати користувачу можливість вводити числа у `SG1`, встановимо значення властивості `Options->goEditing` в `true`. Також встановимо для обох компонентів значення властивостей `FixedCols`=0 та `FixedRows`=0, тобто таблиці не матимуть заголовків рядків і стовпчиків:

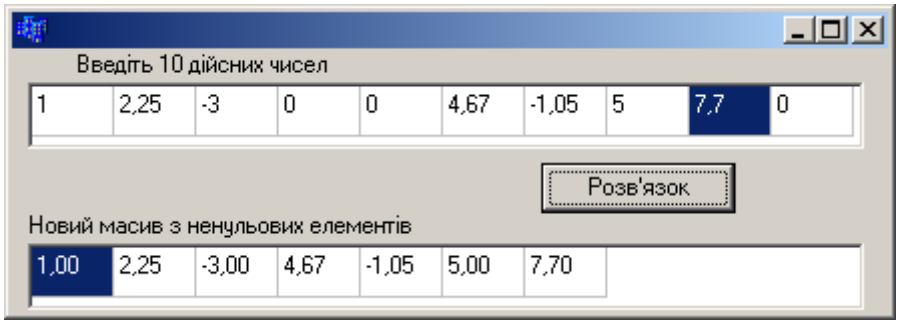


Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ const int N=10;            // Константа N для кількості елементів масиву
 int a[N], i, sum=0;         // a – початковий масив
 float b[N];                 // b – новий масив
 for(i=0; i<N; i++)          // Введення елементів початкового масиву
  a[i]=StrToInt(SG1->Cells[i][0]);// із StringGrid (стовпчик i, рядок 0)
 for(i=0; i<N; i+=2)  // Обчислення суми елементів з непарними індексами
  sum+=a[i];
 Edit1->Text=IntToStr(sum);
 for(i=0; i<N; i++)
 { b[i]=1.*a[i]/sum;          // Ділимо a[i] на sum і присвоюємо результат b[i]
   SG2->Cells[i][0]=FormatFloat("0.00", b[i]);
 }
}
```

Приклад 5.7 Увести масив до 10-ти дійсних чисел і створити новий масив з його ненульових елементів.

Розв'язок. Оскільки реальна кількість ненульових елементів початкового масиву є невідома, оголошуємо новий масив з максимально можливою кількістю елементів, тобто 10. У попередньому прикладі індекси елементів нового масиву при створенні збігалися з індексами відповідних елементів початкового масиву. У цьому прикладі нульові елементи пропускаються і не записуються до нового масиву, тому індекси початкового й результуючого масивів не збігатимуться:



Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ const int N=10;
float a[N],b[N]={0};//Спочатку елементи нового масиву мають нульові значення
int  i,j=0;          // Кількість елементів нового масиву дорівнює 0
for(i=0; i<N; i++)
 { a[i]=StrToInt(SG1->Cells[i][0]);
   if (a[i]!=0)      // Якщо елемент a[i] є ненульовий,
    { b[j]=a[i]; j++; // записати його до b[j] та збільшити j
    }
 }
SG2->ColCount=j;       // Тепер вже є відома реальна кількість комірок SG2
for(i=0; i<j; i++) SG2->Cells[i][0]=FormatFloat("0.0", b[i]);
}
```

5.2.4 Опрацювання одновимірних масивів у функціях

Більш докладно правила організації функцій розглядаються у розд. 8. Тут лише зауважимо, що *функцією* є окремо організований поіменований блок команд програми, в якому розв'язується невелика і специфічна частина загального завдання. Організація програмного проекту з розподілом великого складного завдання на набір більш простих задач і завдань, розв'язування яких виокремлено в логічні блоки – функції, дозволяє спростити читаність програми, її розуміння. Окрім того, застосовування функцій вивільняє програмний код від повторюваних блоків команд, які багаторазово реалізовують один і той самий алгоритм для різних даних. В таких випадках створюється функція з певним набором операторів, а виклик цієї функції здійснюють в основній програмі потрібну кількість разів для різних вхідних параметрів.

Якщо результатом виконання функції є одне значення, яке повертається оператором `return`, то воно підставляється у точку виклику функції. Тому тип функції і є типом результату. Коли функція має повертати масив, типом функції слід оголосити вказівник на тип елемента масиву. Наприклад, оголошена в такий спосіб функція може повертати масив цілих чисел:

```
int * function ();
```

Якщо масив використовують у якості параметра функції, то треба зазначити адресу початку масиву. Зробити це можна одним з трьох способів:

```
float fun (int a[10]);
float fun (int a[]);
float fun (int *a);
```

За першим способом явно зазначено кількість елементів масиву. У другій синтаксичній формі константний вираз у квадратних дужках є відсутній. Така форма є припустима, коли кількість елементів масиву є глобальною змінною чи константою. За третього способу передається вказівник як посилання на масив, явно визначений в основній програмі. Більш докладно роботу з вказівниками буде розглянуто в розд. 6.

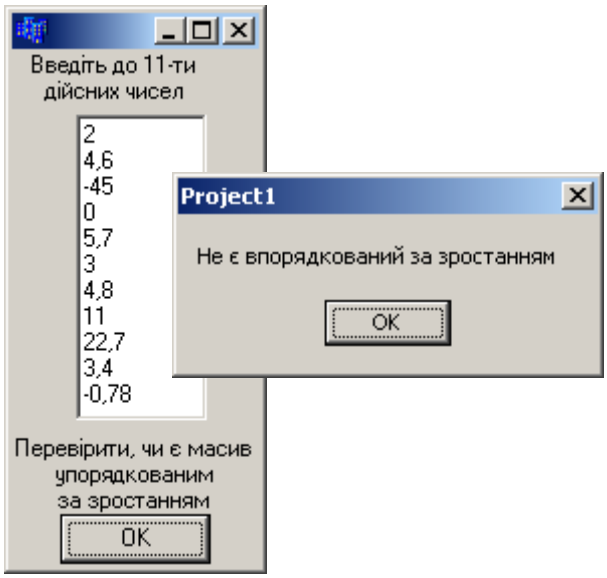
Хоча наведені записи мають різний синтаксис, насправді вони є рівнозначними, оскільки авторами стандарту С для більшої ясності було вирішено, що масив оголошений як параметр функції *є вказівником*. При цьому число у квадратних дужках ігнорується. А тому доцільно передавати розмірність масиву окремим параметром:

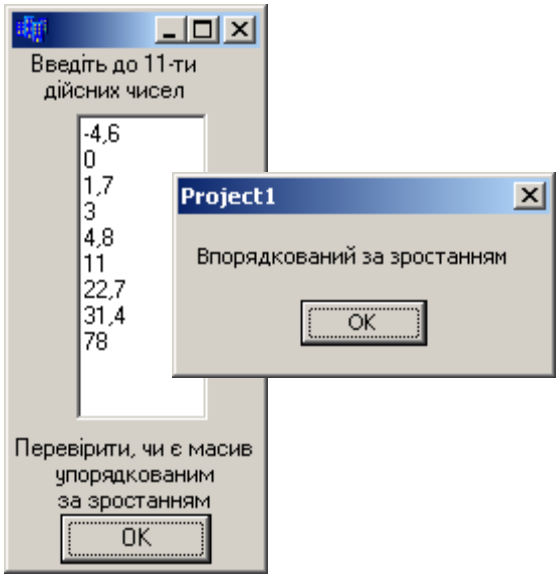
```
float fun (int a[], int n);
float fun (int *a, int n);
```

Якщо функцію організовано для опрацювання елементів масиву, наприклад для сортування елементів, у такому разі можна оголосити функцію з типом результату `void` (нема величини, що повертається). Оскільки сам масив передається у функцію за посиланням, то будь-які змінювання значень елементів масиву у функції буде видно і в основній програмі, яка викликає цю функцію (див. нижче приклади 5.9 – 5.12).

Приклад 5.8 Ввести масив до 11-ти дійсних чисел і з'ясувати, чи є він упорядкованим за зростанням.

Розв'язок 1. У циклі порівнюватимемо кожну пару сусідніх елементів. Якщо принаймні для однієї пари виконається умова (`a[i]>a[i+1]`) (наступний елемент є менший за попередній), це означатиме, що масив не є впорядкований, і в такому разі слід вивести про це повідомлення і перервати виконування функції `Button1Click`. Якщо цикл завершиться, а виконування функції перервано не було, це означатиме, що масив є впорядкований, про що виведеться відповідне повідомлення.






```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ double a[11];
  int i, N=Memo1->Lines->Count;
  if (N>11) N=11;
  for(i=0; i<N; i++)a[i]=StrToFloat(Memo1->Lines->Strings[i]);
  for(i=0; i<N-1; i++)
    if(a[i] > a[i+1])
      {ShowMessage("Не є впорядкований за зростанням"); return;}
  ShowMessage("Впорядкований за зростанням");
}
```

Розв'язок 2. Напишемо аналогічну програму зі створенням функції `sorted()`, яка повертає значення `true`, якщо масив є впорядкований за зростанням, і `false` – якщо ні. У циклі порівнюємо кожну пару елементів, і, якщо зустрінеться пара, для якої наступний елемент є менше за попередній, перериваємо виконування функції й повертаємо `false`. Якщо цикл завершено і виконування функції перервано не було, повертаємо `true`. Докладніше про специфіку організації функцій див. у розд. 8.

```
bool sorted(double a[], int n)
{ for (int i=0; i<n-1; i++)
   if (a[i] > a[i+1]) return false;
  return true;
}
```

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ double a[11];
  int i, N=Memo1->Lines->Count;
  if (N>11) N=11;
  for (i=0; i<N; i++)
    a[i]=StrToFloat(Memo1->Lines->Strings[i]);
  if (sorted(a, N)) ShowMessage("Впорядкований за зростанням");
  else  ShowMessage("Не є впорядкований за зростанням");
}
```

Приклад 5.9 Розробити схему алгоритму і проект програми для введення до 15-ти дійсних чисел із компонента `Memo` і впорядкувати їх за зростанням.

Розв'язок. Упорядкування масивів за будь-якою ознакою називається також *сортуванням*. Існують різні методи сортування. Вони розрізняються переважно швидкістю отримання результату. Розглянемо лише три з них.

Перший спосіб

В наведеній нижче функції `sort()` запропоновано найбільш поширений метод "бульбашки" (bublesort). Алгоритм сортування полягає у переставлянні двох сусідніх порівнюваних елементів масиву, якщо вони йдуть у неправильному порядку.

Перш ніж розглянемо повний текст програм, звернімо увагу на певні її фрагменти.

Для послідовного порівняння сусідніх елементів масиву `a[i]` та `a[i+1]` слід організувати цикл. При цьому останнє значення параметра циклу `i` у циклі `for` має бути на один менше за кількість усіх елементів, щоб на останній ітерації можна було порівнювати передостанній елемент з останнім.

Для того щоб переставити місцями два елементи, слід застосувати тимчасову змінну `tmp`:

```
tmp = a[i];       // Запам'ятати значення a[i]
a[i] = a[i+1];    // і в a[i] записати a[i+1],
a[i+1] = a[i];    // а в a[i+1] записати те, що спочатку було в a[i]
```

Щоб краще зрозуміти необхідність тимчасової змінної, слід уявити собі склянку соку і чашку кави і спробувати перелити сік у чашку, а каву – у склянку. Це неможливо буде зробити без додаткового посуду, до якого спочатку треба перелити вміст, наприклад, склянки.

Якщо в умові порівняння елементів записати знак ">", то елементи сортуватимуться за зростанням, а якщо знак "<" – за спаданням.

Цикл для одноразового проходження по `n` елементах масиву і за потреби переставляння елементів має вигляд

```
for (i=0; i<n-1; i++)
  if (a[i] > a[i+1])
  { tmp = a[i];
    a[i] = a[i+1];
    a[i+1] = tmp;
  }
```

Перевіримо роботу цього циклу на масиві з 7-ми елементів:

5, –1.3, –7.1, 4.8, 2, 7, 1.6.

Після виконання наведеного вище циклу цей масив набуде вигляду

–1.3, –7.1, 4.8, 2, 5, 1.6, 7

тобто масив ще не є остаточно відсортованим за зростанням і дію наведеного вище циклу слід повторювати, допоки масив не набуде вигляду

–7.1, –1.3, 1.6, 2, 4.8, 5, 7.

У поданій нижче таблиці наведено результати чотирьох проходів масивом. Після останнього проходу масив вже відсортовано.

	Елементи масиву						
	0	*1*	*2*	*3*	*4*	*5*	*6*
початок	5	–1.3	–7.1	4.8	2	7	1.6
1-й крок	–1.3	7.1	4.8	2	5	1.6	7
2-й крок	–7.1	–1.3	2	4.8	1.6	5	7
3-й крок	–7.1	–1.3	2	1.6	4.8	5	7
4-й крок	–7.1	–1.3	1.6	2	4.8	5	7

Оскільки заздалегідь невідомо, скільки разів треба повторювати проходження масивом, доцільно організувати ще один цикл. Зовнішній цикл (по `i`) дозволяє перебрати всі елементи масиву для порівняння з іншими елементами, доступ до яких організовано у вкладеному циклі (по `j`). Можна навести два різновиди організації порівнянь елементів у циклах:

1) Циклічно порівнюються два сусідні елементи – `a[j]` та `a[j+1]`. За рахунок того, що після кожного проходження циклом один з елементів ставатиме на своє місце, можна скоротити кількість повторювань вкладеного циклу до `(n – i – 1)` разів.

```
for (i=0; i<n-1; i++)
for (j=0; j<n-i-1; j++)
 if (a[j] > a[j+1])
  { tmp = a[j];
    a[j] = a[j+1];
    a[j+1] = tmp;
  }
```

Перевіримо роботу цього циклу на масиві з 7-ми елементів:

9, 8, 7, 6, 5, 4, 3.

Для наочності взято початковий масив, елементи якого розташовано у порядку, протилежному до того, який має бути набуто. У таблиці подано вигляд масиву після кожної ітерації циклу по `i`. Сірим кольором і рамкою виокремлено елементи, які беруть участь у перестановках наступної ітерації.

	0	*1*	*2*	*3*	*4*	*5*	*6*
початок	9	8	7	6	5	4	3
i=0	8	7	6	5	4	3	9
i=1	7	6	5	4	3	8	9
i=2	6	5	4	3	7	8	9
i=3	5	4	3	6	7	8	9
i=4	4	3	5	6	7	8	9
i=5	3	4	5	6	7	8	9

2) Циклічно порівнюються два елементи – `a[i]` та `a[j]`. За рахунок того, що після кожного проходження циклом один з елементів ставатиме на своє місце, можна скоротити кількість повторювань вкладеного циклу, розпочинаючи вкладений цикл не з 0-го елемента, а з (`i + 1`)-го.

```
for (i=0; i<n-1; i++)
for (j=i+1; j<n; j++)
 if (a[i] > a[j])
  { tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;  }
```

Подібно до першого фрагменту, наведемо подібну таблицю для того само початкового масиву.

	0	*1*	*2*	*3*	*4*	*5*	*6*
початок	9	8	7	6	5	4	3
i=0	3	9	8	7	6	5	4
i=1	3	4	9	8	7	6	5
i=2	3	4	5	9	8	7	6
i=3	3	4	5	6	9	8	7
i=4	3	4	5	6	7	9	8
i=5	3	4	5	6	7	8	9

Сортування елементів масиву організуємо в окремій функції. Слід звернути увагу на заголовок цієї функції:

```
void sort(float a[], int n)
```

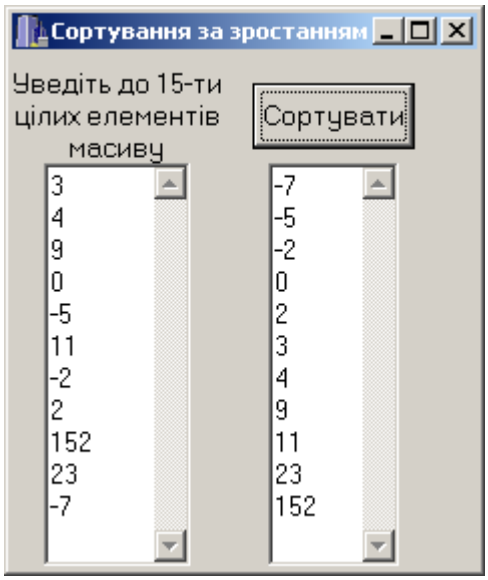
де `sort` – ім'я функції, за яким її можна буде викликати;

`float a[]`, `int n` – параметри функції: масив дійсних чисел `a` (який функція сортуватиме) і кількість елементів масиву `n`;

`void` – тип результату, який вказує на те, що функція виконуватиме сортування і змінюватиме масив, але жодних інших обчислень вона не виконуватиме і нічого, окрім зміненого масиву, повертати немає потреби.

Викликати розглянуту функцію можна, якщо написати ім'я функції й у круглих дужках зазначити два фактичні параметри: ім'я масиву та кількість його елементів, наприклад:

```
sort(a, 15);          // Якщо елементів є 15
sort(a, n);           // Якщо елементів є n
```



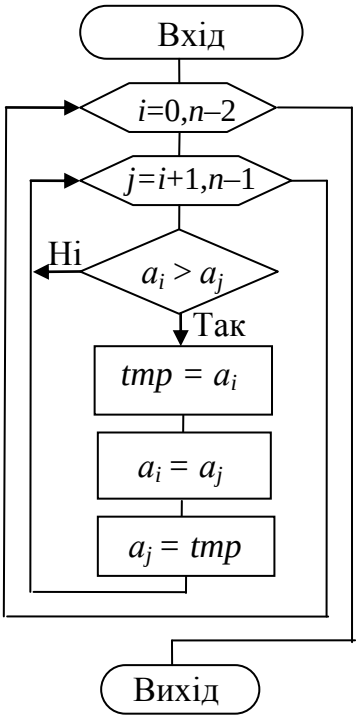


Схема алгоритму

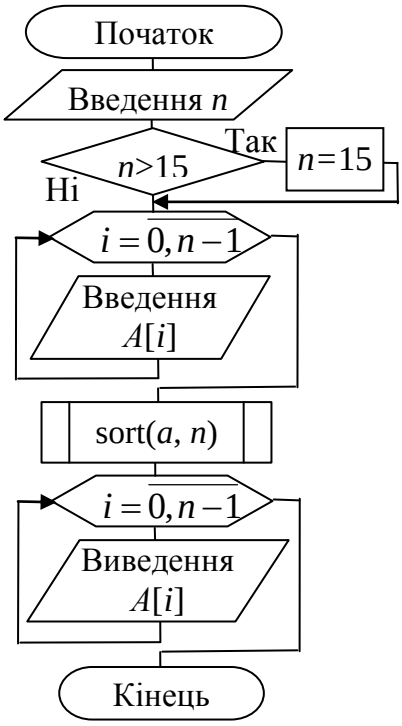


Схема алгоритму до основної програми

Текст функції та основної програми:

```
void sort(float a[], int n)
{ int i, j;     // Параметри наступних циклів
  float tmp;    // Тимчасова зміна для переставлянь елементів
  for (i=0; i<n-1; i++)
  for (j=i+1; j<n; j++)
   if (a[i] > a[j])
    { tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
    }
}
//---------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int n,i;  float a[15];
  // Зчитування кількості заповнених рядків Memo1
  n = Memo1->Lines->Count;
  if (n>15) n=15;
  for (i=0; i<n; i++)
    a[i]=StrToFloat(Memo1->Lines->Strings[i]);
  sort(a, n);   // Виклик функції
  Memo2->Clear();
  for (i=0; i<n; i++)
    Memo2->Lines->Add(FloatToStr(a[i]));
}
```

Другий спосіб

Аналізуючи попередній алгоритм сортування, можна скоротити кількість проходжень по масиву. В наведеному нижче прикладі функції `sort2()` запропоновано алгоритм сортування, де булева змінна `ok` відіграє роль своєрідного ліхтарика, який "вмикається" (`ok=1`) лише в разі переставляння сусідніх елементів, тобто є сигналом, що ще не всі елементи є упорядковано і треба повторювати перевірки.

Текст функції:

```
void sort2(float a[], int n)
{ int i; float tmp; bool ok=1;
  while (ok)
    { ok=0;
      for (j=0; j<n-1; j++)
      if (a[j] > a[j+1])
       { tmp = a[j];
         a[j] = a[j+1];
         a[j+1] = tmp;
         ok = 1;
       }
    }
}
```

Перевіримо роботу цього циклу на масиві з 7-ми елементів:

–0.9, –2.1, 6.3, 3, 8, 5.1, 30.

В таблиці показано вигляд масиву після кожного проходження масивом.

	0	*1*	*2*	*3*	*4*	*5*	*6*
початок	–0.9	–2.1	6.3	3	8	5.1	30
1-й прохід	–2.1	–0.9	3	6.3	5.1	8	30
2-й прохід	–2.1	–0.9	3	5.1	6.3	8	30
3-й прохід	–2.1	–0.9	3	5.1	6.3	8	30

Отже, на відміну від попереднього способу сортування, кількість проходжень циклом істотно скорочується, залежно від початкового розташування елементів. Останнє проходження циклом можна назвати холостим, оскільки перестановок у ньому немає, а отже, булева змінна `ok` не "вмикається" (`ok=1`), що є сигналом, що всі елементи упорядковано, – і відбувається вихід з циклу `while`.

Третій спосіб

Доволі поширеним є ще один спосіб сортування за допомогою мінімального елемента, алгоритм якого реалізовано у наведеній нижче функції `sort3()`. Його суть полягає в тому, що з послідовності обирають мінімальний (чи максимальний) елемент і переставляють його до початку (кінця) масиву, змінюючи місцями з перевірюваним. Кількість перестановок – `(n+1)/2`.

Розглянемо цей алгоритм для сортування за зростанням масиву з 7-ми елементів:

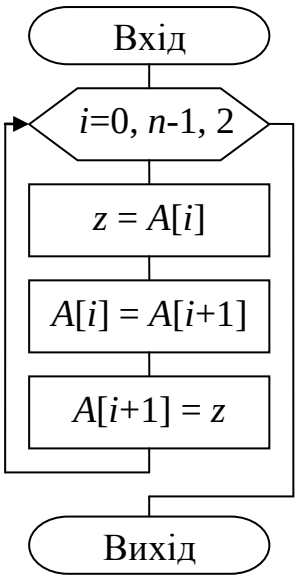
5, –1.3, –7.1, 4.8, 2, 7, 1.6.

Щоб елементи було розташовано за зростанням, слід віднайти мінімальний елемент масиву і поставити його на перше місце. Після цього повторити аналогічні дії для решти елементів (окрім першого, який вже стоїть на своєму місці). У таблиці на кожній ітерації жирним позначено мінімальний елемент, рамкою виокремлено елементі, які беруть участь у подальшому пошуку мінімального елемента і перестановках.

	0	*1*	*2*	*3*	*4*	*5*	*6*
початок	5	–1.3	**–7.1**	4.8	2	7	1.6
i=0	–7.1	**–1.3**	5	4.8	2	7	1.6
i=1	–7.1	–1.3	5	4.8	2	7	**1.6**
i=2	–7.1	–1.3	1.6	4.8	**2**	7	5
i=3	–7.1	–1.3	1.6	2	**4.8**	7	5
i=4	–7.1	–1.3	1.6	2	4.8	7	**5**
i=5	–7.1	–1.3	1.6	2	4.8	5	7

```
void sort3(float a[], int n) //n – кількість елементів масиву
{ int i, j, imin; float tmp;
  for (i=0; i<n-1; i++)
   { imin=i;
     for (j=i+1; j<n; j++)
      if (a[imin] > a[j]) imin=j;
     tmp = a[i];
     a[i] = a[imin];
     a[imin] = tmp;
   }
}
```

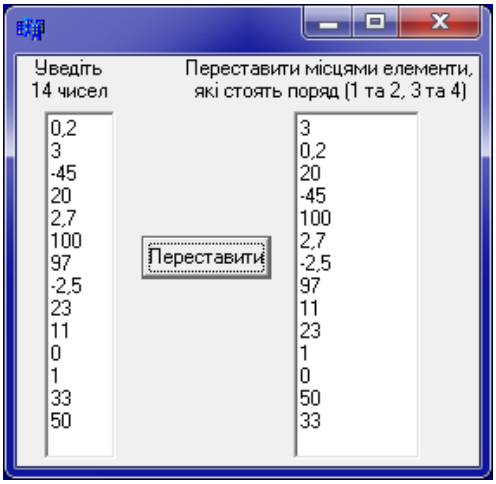
Приклад 5.10 В одновимірному масиві дійсних чисел, який складається з 14-ти (парної кількості) елементів, переставити місцями елементи, які стоять поряд (1 та 2, 3 та 4 тощо).



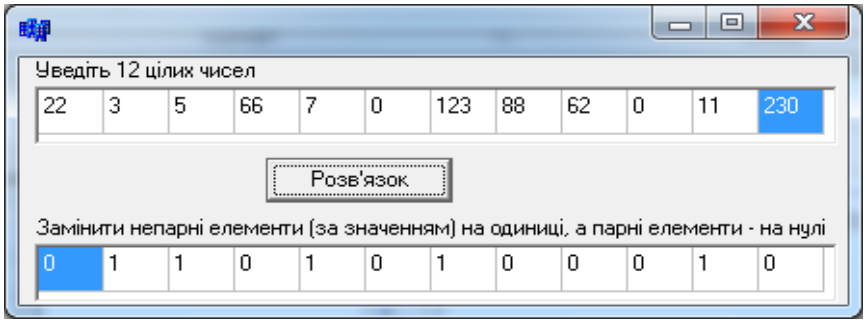
Текст функції та її виклику в основній програмі:

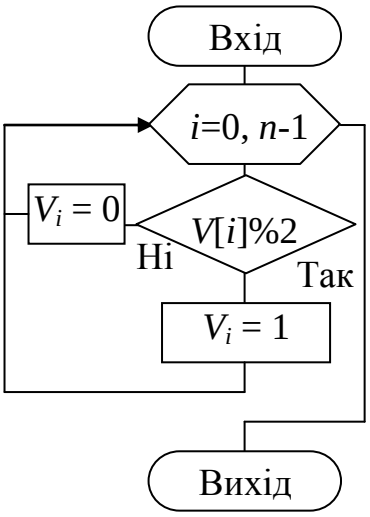
```
void Perehod (float A[], int n)
{ int i;
  float  z;
  for (i=0; i<n; i+=2)
   { z = A[i];
     A[i] = A[i+1];
     A[i+1] = z;
   }
}
//-----------------------------------------------
```

```
void __fastcall TForm1::Button1Click
                              (TObject *Sender)
{float a[14];
 for (int i=0; i<14; i++)
   a[i]=StrToFloat(Memo1->Lines->
                                  Strings[i]);
 Perehod (a, 14);
 Memo2->Clear();
 for (i=0; i<14; i++)
   Memo2->Lines->Add(FloatToStrF(a[i],
                            ffGeneral, 5, 2));
}
```



Приклад 5.11 У послідовності з 12-ти цілих чисел непарні елементи (за значенням, а не за індексом) замінити на одиниці, а парні елементи – на нулі.



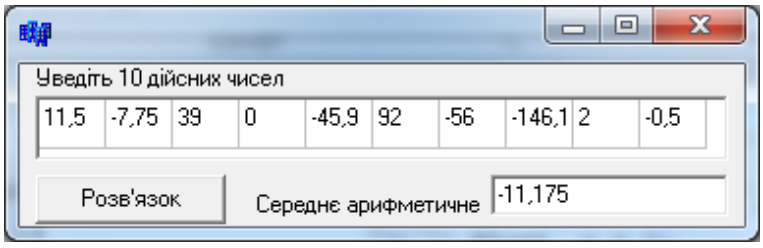


Розв'язок. Введення та виведення послідовності чисел організуємо з компонента `StringGrid`, роботу якого докладно розглянуто у п. 5.3.2.

Текст функції та основної програми:

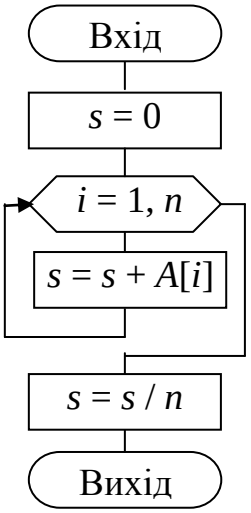
```
void Zamina(int V[12])
{ for (int i=0; i<12; i++)
   if (V[i] % 2) V[i]=1;   else  V[i]=0;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int a[12], i;
  for (i=0; i<12; i++) a[i]=StrToInt(StringGrid1->Cells[i][0]);
  Zamina(a);
  for (i=0; i<12; i++) StringGrid2->Cells[i][0]=IntToStr(a[i]);
}
```

Приклад 5.12 Обчислити середнє арифметичне усіх елементів одновимірного масиву з 10-ти дійсних чисел.



Текст функції та основної програми:

```
float middle (float *a, int n)
{ float  s = 0;
  for (int i=0; i<n; i++) s += a[i];
  return  s / n;
}
//---------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float A[10], sr;
  for (int i=0; i<10; i++)
    A[i]=StrToFloat(StringGrid1->Cells[i][0]);
  sr = middle(A, 10);
  Edit1->Text=FormatFloat("0.000", sr);
}
```



5.3 Двовимірні масиви

5.3.1 Організація двовимірних масивів

Доволі часто буває недостатньо однієї вимірності масиву для зручного зберігання даних. Приміром, якщо є потреба зберігати результати сесії цілої групи (30 студентів): оцінки з п'яти предметів. На папері ці результати зазвичай заносяться до таблиці, кожний рядок якої є оцінками одного студента по всіх предметах, а кожний стовпчик – оцінками групи з конкретного предмета:

Прізвище	Математика	Фізика	Інформатика	Історія	Фізкультура
Антонов	95	95	95	95	95
Бондарь	90	85	90	95	95
Василенко	60	75	65	80	80
…	…	…	…	…	…

Так само згадані дані можуть зберігатися і в масиві, де номери рядків означають номери студентів, а номери стовпчиків – номери предметів:

	0	*1*	*2*	*3*	*4*
0	95	95	95	95	95
1	90	85	90	95	95
2	60	75	65	80	80
…	…	…	…	…	…

Цей масив є двовимірний: номер рядка визначає оцінки кожного студента, а номер стовпчика – предмет. Якщо в одному масиві зберігати оцінки всіх груп разом, то це буде вже тривимірний масив, і елемент визначатиметься ще й номером групи.

Як зазначалось на початку розділу, вимірність масиву визначається кількістю індексів. Елементи одновимірного масиву (вектора) мають один індекс, двовимірного масиву (матриці, таблиці) – два індекси: перший з них – номер

рядка, другий – номер стовпчика. Кількість індексів у масивах є необмежена. При розміщуванні елементів масиву в пам'яті комп'ютера першою чергою змінюється крайній правий індекс, потім решта – справа наліво.

Багатовимірний масив оголошується у програмі в такий спосіб:

<тип> *<ім'я>* [*<розмір1>*] [*<розмір2>*] … [*<розмірN>*];

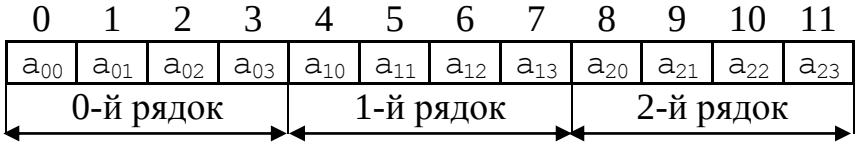
Кількість елементів масиву дорівнює добуткові кількості елементів за кожним індексом. У прикладі

```
int a[3][4];
```

оголошено двовимірний масив з 3-х рядків та 4-х стовпчиків (12-ти елементів) цілого типу:

```
a[0][0], a[0][1], a[0][2], a[0][3],
a[1][0], a[1][1], a[1][2], a[1][3],
a[2][0], a[2][1], a[2][2], a[2][3];
```

Під масив надається пам'ять, потрібна для розташування усіх його елементів. Елементи масиву один за одним, з першого до останнього, запам'ятовуються у послідовно зростаючих адресах пам'яті так само, як і елементи одновимірного масиву. Наприклад, масив `int a[3][4]` зберігається у пам'яті у такий спосіб:



Масив оцінок з п'яти предметів для групи з 30-ти студентів можна оголосити, наприклад, у такий спосіб:

```
int ocinki[30][5];
```

Якщо цей масив зберігатиме оцінки з наведеної таблиці, то до оцінки з математики першого в групі студента з прізвищем Антонов можна звернутися у такий спосіб: `ocinki[0][0]`, до оцінки Бондаря з інформатики: `ocinki[1][2]`.

Наведемо ще кілька прикладів оголошення масивів:

```
float Mas1 [5][5];      // Матриця 5×5=25 елементів дійсного типу.
char Mas2 [10][3];
                        // Двовимірний масив з 10×3=30  елементів символьного типу
double Mas3 [4][5][4];
                        // Тривимірний масив з 4×5×4=80 елементів дійсного типу
```

При оголошенні масиву можна ініціалізовувати початкові значення його елементів, причому необов'язково усіх, наприклад:

1) `int w[3][3]={ { 2, 3, 4 },{ 3, 4, 8 },{ 1, 0, 9 } };`
2) `float C[4][3]={1.1, 2, 3, 3.4, 0.5, 6.8, 9.7, 0.9};`
3) `float C[4][3]={{1.1, 2, 3},{3.4, 0.5, 6.8},{ 9.7, 0.9}};`
4) `float C[4][3]={{1.1, 2},{3, 3.4, 0.5},{6.8},{9.7, 0.9}};`

У першому прикладі оголошено й ініціалізовано масив цілих чисел `w[3][3]`. Елементам масиву присвоєно значення зі списку: `w[0][0]=2`,

`w[0][1]=3`, `w[0][2]=4`, `w[1][0]=3` тощо. Списки, виокремлені у фігурні дужки, відповідають рядкам масиву.

Записи другого й третього прикладів є еквівалентні, і в них елементи останнього рядка не є ініціалізовані, тобто не є визначені. У таких випадках у числових масивах неініціалізовані елементи набувають значення 0.

Розташування елементів у прикладах 2 та 3

1.1	2	3
3.4	0.5	6.8
9.7	0.9	0
0	0	0

Розташування елементів у прикладі 4

1.1	2	0
3	3.4	0.5
6.8	0	0
9.7	0.9	0

Багатовимірні масиви можна оголошувати й зі створюванням типу користувача (докладніше див. розд. 11):

`typedef` <*тип_даних*> <*ім'я_типу*>[<*розмір1*>][<*розмір2*>]…[<*розмір n*>];
<*ім'я_типу*> <*ім'я_масиву*> ;

Наприклад, створимо тип з ім'ям `matr` як масив додатних цілих чисел з 10-ти рядків і 7-ми стовпчиків та оголосимо два масиви – `M1` і `M2` – створеного типу:

```
typedef unsigned short matr[10][7];
matr M1, M2;
```

Для оголошення масивів можна також використовувати типізовані констант-масиви, які дозволяють водночас оголосити масив і задати його значення в розділі оголошень констант, наприклад:

```
const int arr[2][5] = {{9, 3, 0, 12, -5},{ -7, 23, 2, 4, 0}};
```

але змінювати значення елементів констант-масивів у програмі є неприпустимо.

В С++ можна використовувати *перетин масиву* (section). Це поняття подібно до поняття мінору матриці в математиці. Перетин формується внаслідок вилучення однієї чи кількох пар квадратних дужок. Пари квадратних дужок можна відкидати лише справа наліво й лише послідовно. Перетини масивів використовуються при організації обчислювального процесу в функціях мовою С++, розроблюваних користувачем.

Приклади:

1) `int x[5][3];`

Якщо при звертанні до певної функції написати `x[0]`, то передаватиметься нульовий рядок масиву `x`.

2) `int y[2][3][4];`

При звертанні до масиву `y` можна записати, наприклад, `y[1][2]` – і буде передаватися вектор з чотирьох елементів, а звертання `y[1]` надасть двовимірний масив розміром 3×4. Не можна писати `y[2][4]`, вважаючи на увазі, що передаватиметься вектор, тому що це не відповідає обмеженню, накладеному на використання перетинів масиву.

5.3.2 Введення-виведення двовимірних масивів

Виведення елементів двовимірних масивів

Здійснювати виведення значень елементів масиву можна лише поелементно, для чого слід організовувати цикли, в яких послідовно змінюватимуться значення індексів елементів.

У поданих нижче прикладах виведення елементів масиву використовуватимуться змінні, оголошені в такий спосіб:

```
int B[5][3];  int i, j;  AnsiString st;
```

До компонента Memo можна виводити масиви рядками, відокремлюючи пробілами елементи в рядку. В компоненті `Memo` можна використовувати смуги прокручування, для чого властивості `ScrollBar` слід задати значення `ssBoth` чи то `ssVertical`.

```
Memo1->Clear();
for (i=0; i<5; i++)
{ st=""  ;
  for (j=0; j<3; j++)
    st += FormatFloat("0.00",B[i][j]) + " ";
  Memo1->Lines->Add(st);
}
```

До компонента ListBox виведення масиву організовують так само, як і для компонента `Memo`, лише замість властивості `Lines` слід використовувати властивість `Items`.

Приміром, замість

```
Memo1->Lines->Add(st);
```

слід записати

```
ListBox->Items->Add(st);
```

Компонент StringGrid (таблиця рядків)  має вигляд таблиці з комірками і розташований на закладці `Additional` палітри компонентів. У кожній комірці компонента `StringGrid` можна розмістити величину рядкового типу, причому задану кількість перших рядків та стовпчиків може бути зафіксовано і при прокручуванні вони залишатимуться на місці. У перебігу створювання форми не можна задавати значення комірок таблиці, оскільки відповідна властивість `Cells` є доступна лише програмно.

Приміром, для виведення матриці `B` розміром 5×3 компонентові `StringGrid1` слід задати властивості, наведені в табл. 5.1. Для виведення матриці у комірки `StringGrid` треба організовувати цикли по номерах рядків та стовпчиків. Приклад виведення матриці `float B[5][3]` до компонента `StringGrid1`:

```
for (i=0;i<5;i++)
for (j=0;j<3;j++)
 StringGrid1->Cells[j+1][i+1]=FormatFloat("0.00", B[i][j]);
```

Таблиця 5.1

Основні властивості компонента StringGrid

Властивість	Опис	Значення за замовчуванням	Нові значення
RowCount	Кількість рядків	5	6
ColCount	Кількість стовпчиків	5	4
Cells	Програмно доступна властивість для звертання до комірок таблиці. Приміром, Cells[j][i] – це комірка в *i*-тому рядку *j*-того стовпчика (нумерація розпочинається від нуля)		
FixedCols	Кількість фіксованих стовпчиків (для заголовка рядків)	1	1
FixedRows	Кількість фіксованих рядків (для заголовка стовпчиків)	1	1
Options. goEditing	Ознака дозволу на редагування змісту комірок	false	true
Options. goTabs	Ознака дозволу на переміщення таблицею за допомогою <Tab>	false	true
Options. goColSizing	Ознака дозволу на змінення ширини стовпчиків	false	true
Options. goRowSizing	Ознака дозволу на змінення висоти рядків	false	true

Приклад створення заголовків таблиці в комірках фіксованих рядка та стовпчика (з індексом 0) під час створення форми.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{ for(int i=1; i<=5; i++)
    StringGrid1->Cells[0][i] = IntToStr(i) + "-й рядок";
  for(int j=1; j<=3; j++)
    StringGrid1->Cells[j][0] = IntToStr(j) + "-й стовпчик";
}
```

До прикладу введення матриці `float B[5][3]` з комірок `StringGrid1`

```
for(int i=0; i<5; i++)
for(int j=0; j<3; j++)
  B[i][j]=StrToFloat(StringGrid1->Cells[j+1][i+1]);
```

можна долучити перевірку на незаповнені комірки:

```
for(int i=0; i<5; i++)
for(int j=0; j<3; j++)
  if(StringGrid1->Cells[j+1][i+1] != "")
    B[i][j]=StrToFloat(StringGrid1->Cells[j+1][i+1]);
  else
    { ShowMessage("Заповніть ["+IntToStr(i+1) + ","
                              + IntToStr(j+1) + "] елемент");
      break;
    }
```

Введення матриці z розміром m×n у *консолі:*

```
for(int i=0; i<m; i++)
for(int j=0; j<n; j++) cin >> z[i][j];
```

Для *виведення* двовимірного масиву a[4][5] у *консолі* у вигляді матриці елементи слід розташувати у рівні стовпчики. Це можна зробити за допомогою маніпулятора setw(), який задає ширину виведення змінної у cout, (аналогічно до табуляції). Для використання маніпуляторів треба долучити до програми заголовний файл iomanip.h:

```
#include <iostream.h>
#include <iomanip.h>
int main()
{ int a[4][5];
  for(int i=0; i<4; i++)
  for(int j=0; j<5; j++) a[i][j]=random(15)-5;
  for(int i=0; i<4; i++)
  { for(int j=0; j<5; j++)    // Аналогічне виведення рядка
      cout<<setw(5)<<a[i][j];// з табуляцією: cout<<"\t"<<a[i][j];
    cout<<endl;              //переведення курсора на новий рядок
  }
  cin.get();
  return 0;
}
```

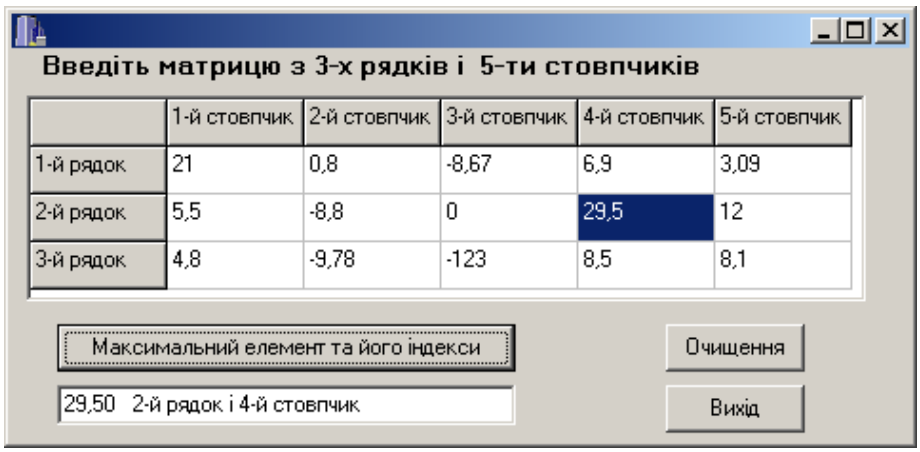
```
   -1    4    6   -1    7
   -2    6    4    1    1
   -4    9    5   -3    3
    4    8    8   -4   -5
```

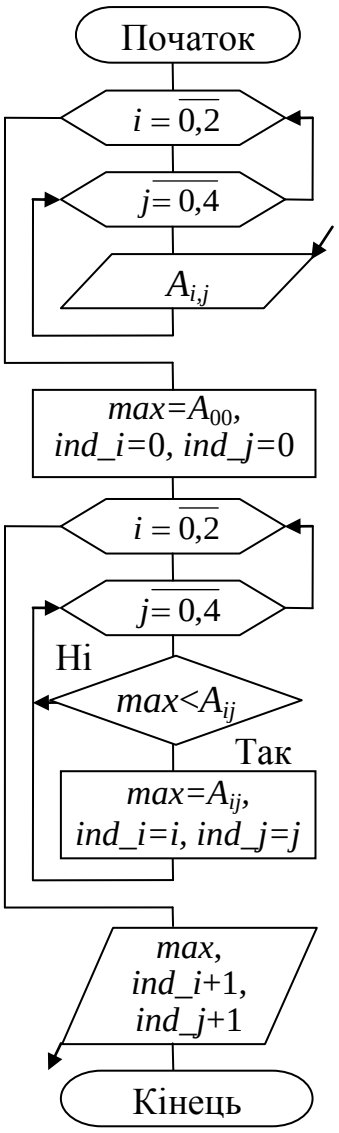
5.3.3 Програмування базових алгоритмів опрацювання двовимірних масивів

Приклад 5.13 Скласти схему алгоритму і розробити проект для введення матриці дійсних чисел розмірністю 3×5 і віднайти максимальний елемент матриці та його індекси.

Розв'язок. На формі розташовано компоненти StringGrid1, Edit1, Button1, Button2, Button3, Label1 та Label2, а їхні нові властивості зазначено в таблиці.

Компоненти	Властивості	Нові значення
StringGrid1	RowCount	4
StringGrid1	ColCount	6
StringGrid1	Options.goEditing	true
StringGrid1	Options.goTabs	true
StringGrid1	Name	SG1
Button1	Caption	Максимальний елемента його індекси
Button2	Caption	Очищення
Button3	Caption	Вихід





Текст програми:

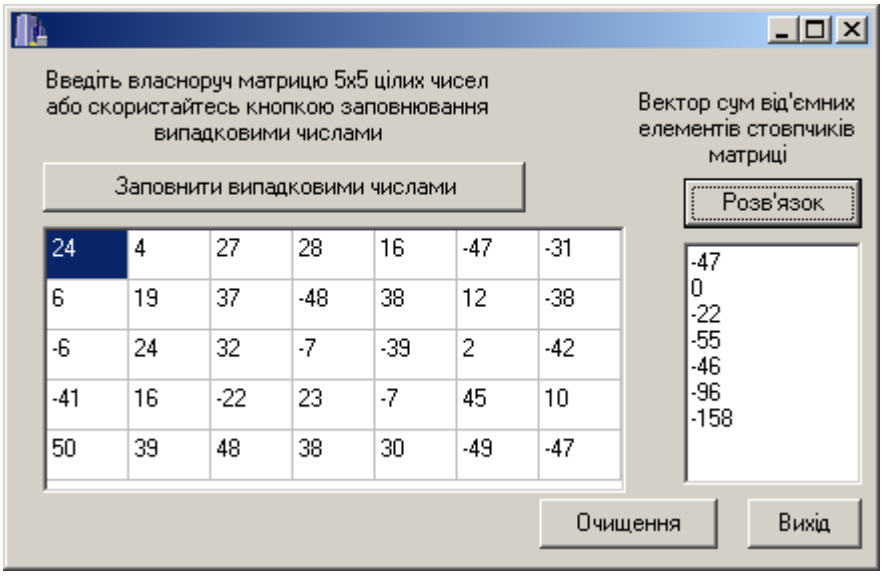
```
// Створення форми
void __fastcall TForm1::FormCreate(TObject *Sender)
{ for(int i=1; i<=3; i++)
    SG1->Cells[0][i]=IntToStr(i)+"-й рядок";
  for(int j=1; j<=5; j++)
    SG1->Cells[j][0] = IntToStr(j)+"-й стовпчик";
}
//---------------------------------------------
// Максимальний елемент та його індекси
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float A[3][5], max;   int i, j, ind_i, ind_j;
  for (i=0; i<3; i++)
  for (j=0; j<5; j++)
   if (SG1->Cells[j+1][i+1] != "")
     A[i][j] = StrToFloat(SG1->Cells[j+1][i+1]);
     else
      { ShowMessage("Введіть [" + IntToStr(i+1) + "," +
                                          IntToStr(j+1) + "] елемент");
        break;  }
  max = A[0][0];  ind_i = 0; ind_j = 0;
  for (i=0; i<3; i++)
  for (j=0; j<5; j++)
    if (max < A[i][j])  { max = A[i][j]; ind_i = i; ind_j = j; }
  Edit1->Text = FormatFloat("0.00",max) + "   " + IntToStr(ind_i+1)
              + "-й рядок і " + IntToStr(ind_j + 1) + "-й стовпчик";
}
//---------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender) // Очищення
{ for (int i=0; i<3; i++)
  for (int j=0; j<5; j++) SG1->Cells[j+1][i+1] = "";
  Edit1->Clear();
}
//---------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender) // Вихід
{  Close();  }
```

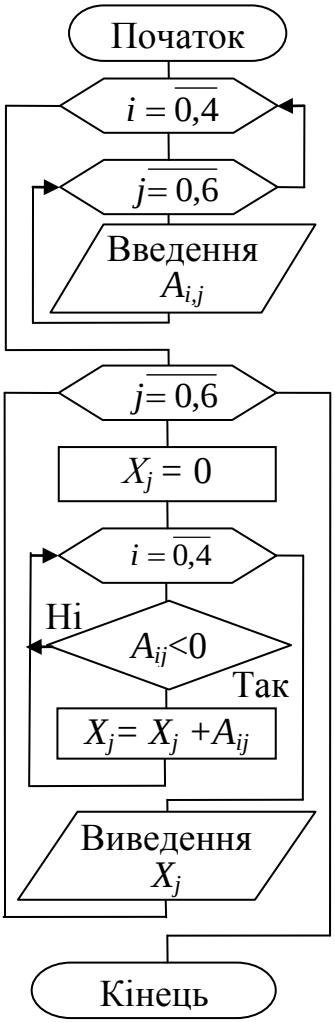
Приклад 5.14 Скласти схему алгоритму і розробити проект для введення матриці цілих чисел розмірністю 5×7 та обчислення елементів вектора сум від'ємних елементів стовпчиків матриці.

Розв'язок. Для заповнення вихідної матриці слід передбачити можливість заповнення комірок випадковими числами. Користувач може скористатись такою можливістю чи то заповнити комірки самостійно. Випадкові числа формуємо за допомогою функції генератора випадкових чисел `random(N)`, яка міститься в бібліотеці `<stdlib.h>`. Функція `random(N)` генерує випадкове ціле додатне число в діапазоні від `0` до `N - 1`. Щоб формувались і від'ємні числа, використаємо конструкцію `(50 - random(100))`. Функція `randomize()` ініціалізує генератор випадкових чисел (див. підрозд. 3.7).

Нові властивості компонента `StringGrid1`, розташованого на формі, наведено нижче в таблиці.

Компоненти	Властивості	Нові значення
StringGrid1	ColCount	7
StringGrid1	RowCount	5
StringGrid1	Options.goEditing	true
StringGrid1	Options.goTabs	true
StringGrid1	FixedCols	0
StringGrid1	FixedRows	0
StringGrid1	DefaultColWidth	40
StringGrid1	Name	SG1





Масив `A` та змінні `i`, `j` оголошено перед усіма функціями глобально, оскільки посилання до цих об'єктів є в різних функціях.

Тексти програм для усіх кнопок:

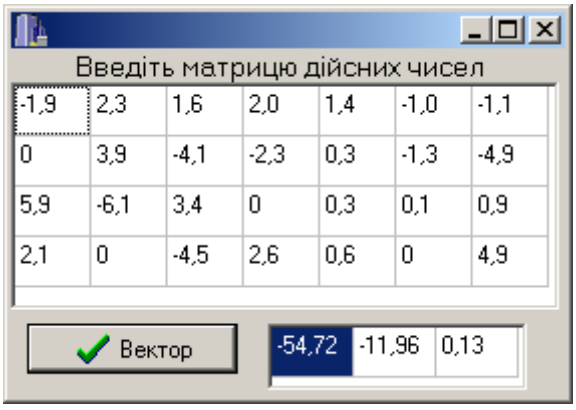
```
int A[5][7],  i, j;
void __fastcall TForm1::Button1Click(TObject *Sender)
{ randomize();// Заповнити випадковими числами
```

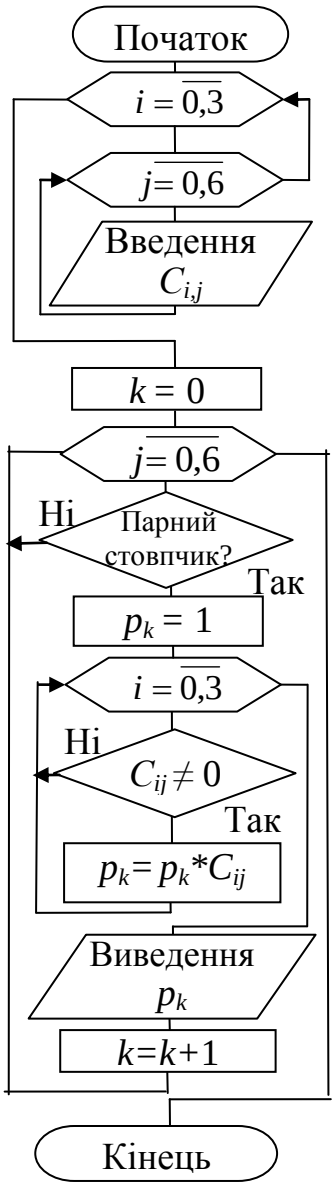
```
  for (i=0; i<5; i++)
  for (j=0; j<7; j++) SG1->Cells[j][i]=IntToStr(50-random(100));
}
//---------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)//Розв'язок
{ int X[7];   Memo1->Clear();
  for (i=0; i<5; i++)
   for (j=0; j<7; j++) A[i][j] = StrToInt(SG1->Cells[j][i]);
  for (j=0; j<7; j++)
   { X[j] = 0;
     for(i=0; i<5; i++) if(A[i][j] < 0)  X[j] += A[i][j];
     Memo1->Lines->Add(IntToStr(X[j]));
}  }
//---------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)//Очищення
{ Memo1->Clear();
  for (i=0; i<5; i++)
  for (j=0; j<7; j++) SG1->Cells[j][i] = "";
}
//---------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{ Close();   // Вихід – закриття форми
}
```

Приклад 5.15 З елементів матриці розмірністю 4×7 дійсних чисел обчислити вектор добутків ненульових елементів парних (2, 4 та 6) стовпчиків матриці.

Розв'язок. Оскільки при формуванні вектора його індекси не збігаються з індексами парних стовпчиків матриці, за якими обчислюється вектор, виникла потреба у використанні додаткової змінної для індексів вектора, наприклад *k*.

Компонент `BitBtn`, зображений на формі, є різновидом стандартної кнопки `Button`, але з можливістю виведення на кнопку, окрім надпису, піктографічних зображень за допомогою властивості `Kind` чи `Glyph`.

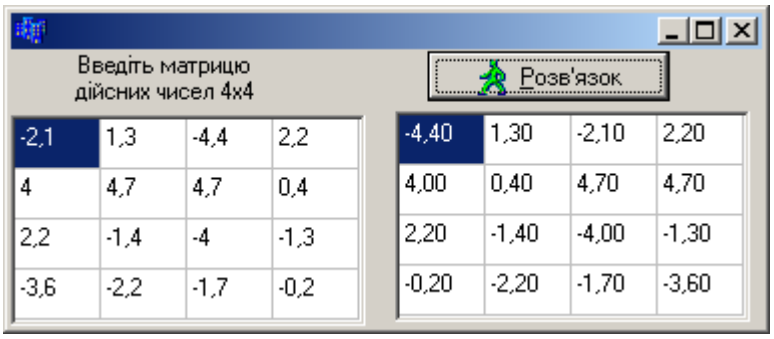


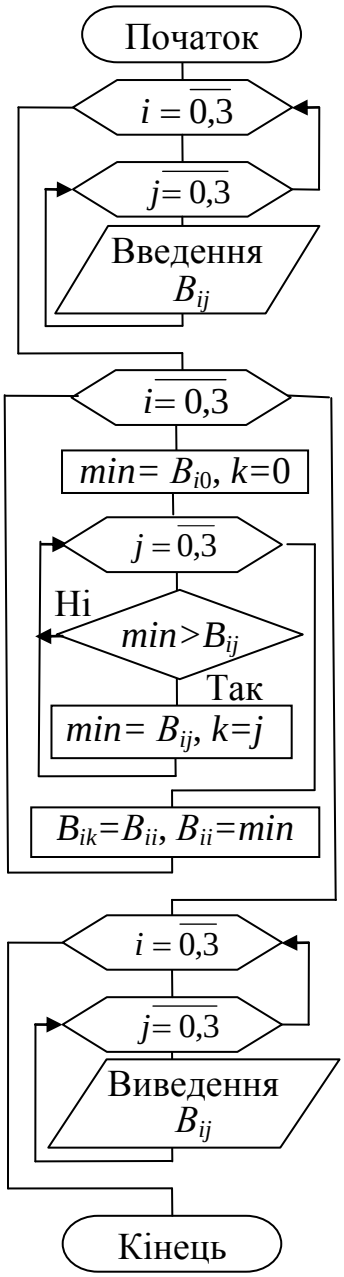


Текст програми:

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{ int i, j, k=0;   float C[4][7], p[3];
  for (i=0; i<4; i++)
   for (j=0; j<7; j++) C[i][j]=StrToFloat(SG1->Cells[j][i]);
  for (j=0; j<7; j++)
   if ((j+1)%2 == 0) //Стовпчик є парний?
    { p[k]=1;
      for (i=0; i<4; i++)
        if (C[i][j]!=0) p[k]*=C[i][j];
      SG2->Cells[k][0]=FormatFloat("0.00", p[k]);
      k++;
 }   }
```

Приклад 5.16 Увести матрицю розмірністю 4×4 дійсних чисел і замінити елементи головної діагоналі на мінімальні елементи відповідних рядків.





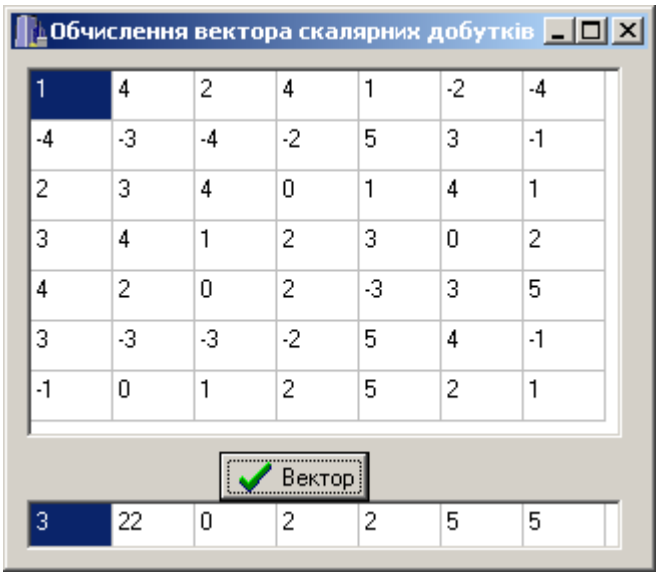
Текст програми:

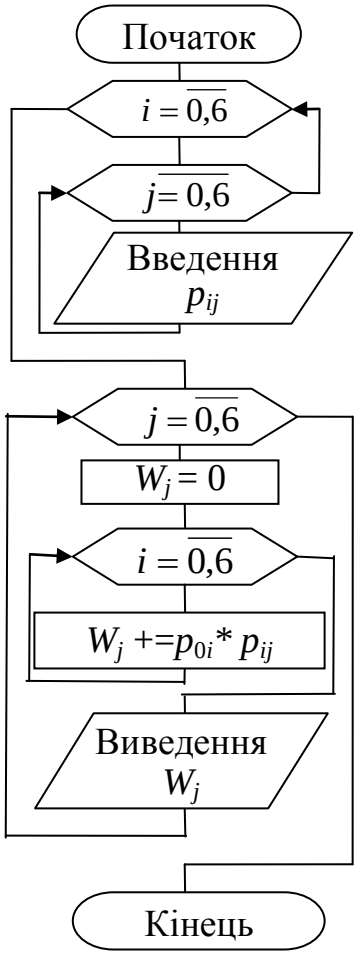
```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{ int i, j, k;   float B[4][4], X[3], min;
  for (i=0; i<4; i++)
   for (j=0; j<4; j++)
    B[i][j]=StrToFloat(SG1->Cells[j][i]);
  for (i=0; i<4; i++)
   { min=B[i][0]; // Запам'ятати перший елемент рядка
     k=0;         // та його індекс як мінімальний.
     for (j=0; j<4; j++) //Рухаючись по стовпчиках
      if(B[i][j]<min) //перевірити: якщо елемент є менший
       { min=B[i][j]; // за min, запам'ятати його значення
         k=j;         // та індекс.
       }
          // Замінити мінімальний елемент рядка на елемент
     B[i][k]=B[i][i]; // діагоналі, а елемент діагоналі –
     B[i][i]=min;     // на мінімальний елемент.
   }
  for (i=0; i<4; i++)
  for (j=0; j<4; j++) SG2->Cells[j][i]=FormatFloat("0.00", B[i][j]);
}
```

Приклад 5.17 З елементів матриці розмірністю 7×7 цілих чисел обчислити вектор скалярних добутків елементів першого рядка на стовпчики матриці.

Розв'язок. Нагадаємо, що скалярний добуток – це сума добутків відповідних елементів. У даному разі:

W[0] = p[0][0]*p[0][0] + p[0][1]*p[1][0] + p[0][2]*p[2][0] + p[0][3]*p[3][0] + ...
W[1] = p[0][0]*p[0][1] + p[0][1]*p[1][1] + p[0][2]*p[2][1] + p[0][3]*p[3][1] + ...
W[2] = p[0][0]*p[0][2] + p[0][1]*p[1][2] + p[0][2]*p[2][2] + p[0][3]*p[3][2] + ...





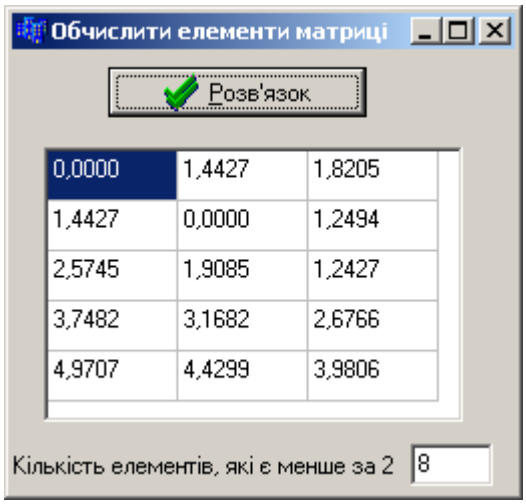
Текст програми:

```
void __fastcall TForm1::BitBtn1Click
                                (TObject *Sender)
{ int i, j, p[7][7], W[7];
  for (i=0; i<7; i++)
  for (j=0; j<7; j++)
    p[i][j] = StrToInt(SG1->Cells[j][i]);
  for (j=0; j<7; j++)
  { W[j]=0;
    for (i=0; i<7; i++)
      W[j] += p[0][i] * p[i][j];
    SG2->Cells[j][0]=IntToStr(W[j]);
  }
}
```

Приклад 5.18 Обчислити елементи матриці розмірністю 5×3 за формулою $a_{ij} = \dfrac{\sqrt{\left|i^3 - j^2\right|}}{\ln(i + j + 1)}$. Обчислити кількість елементів цієї матриці, які є менше за 2.

Текст програми:

```
#include <math.h>
void __fastcall TForm1::Button1Click(TObject *Sender)
{ double a[5][3];  int i, j, k=0;
 for (i=0; i<5; i++)
 for (j=0; j<3; j++)
 { // Обчислення елементів матриці
   if (i==0 && j==0) a[i][j]=0;
   else a[i][j]=sqrt(fabs(pow(i,3)-j*j))/
                                  log(i+j+1);
   SG1->Cells[j][i]=FormatFloat("0.0000",
                a[i][j]); // Виведення матриці
   // Обчислення кількості елементів <2
   if (a[i][j]<2) k++;
 } Edit1->Text=IntToStr(k);
}
```

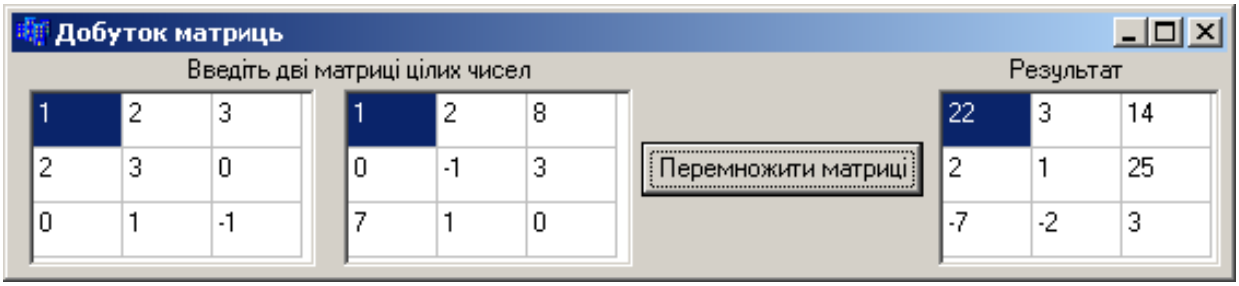


Приклад 5.19 Увести дві цілочисельні матриці 3×3 і обчислити добуток цих матриць.

Розв'язок. Кожний елемент матриці-результату обчислюється за таким алгоритмом: він дорівнює сумі попарних добутків `i`-го рядка та `j`-го стовпчика. Наприклад, другий елемент першого рядка обчислюється як сума попарних добутків відповідних елементів першого рядка і другого стовпчика:

$$\begin{bmatrix}1&2&1\\3&4&0\\1&0&-1\end{bmatrix}\cdot\begin{bmatrix}3&0&1\\-1&1&0\\2&5&7\end{bmatrix}=$$

$$=\begin{bmatrix}1\cdot3+2\cdot(-1)+1\cdot2 & 1\cdot0+2\cdot1+1\cdot5 & 1\cdot1+2\cdot0+1\cdot7\\3\cdot3+4\cdot(-1)+0\cdot2 & 3\cdot0+4\cdot1+0\cdot5 & 3\cdot1+4\cdot0+0\cdot7\\1\cdot3+0\cdot(-1)+-1\cdot2 & 1\cdot0+0\cdot1+-1\cdot5 & 1\cdot1+0\cdot0+-1\cdot7\end{bmatrix}=\begin{bmatrix}3&7&8\\5&4&3\\1&-5&-6\end{bmatrix}$$

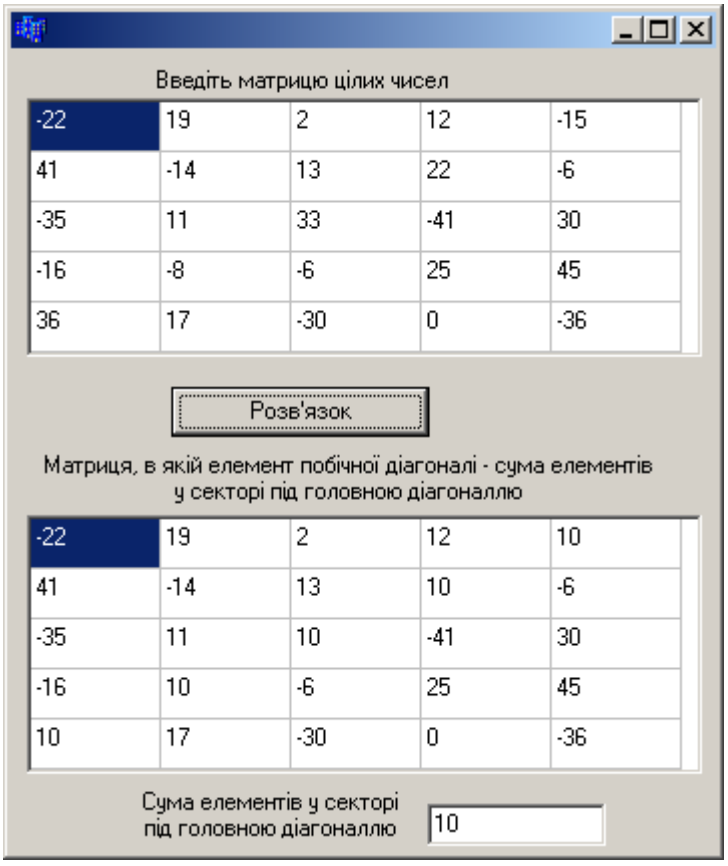


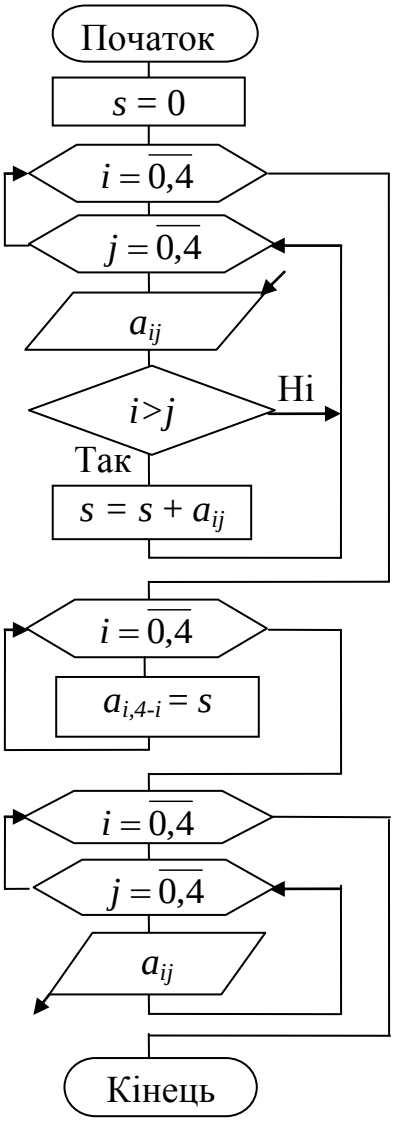
Текст програми

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ const int n=3;
  int i, j, k, a[n][n], b[n][n], c[n][n];
  for (i=0; i<n; i++)
  for (j=0; j<n; j++)
  { a[i][j]=StrToInt(StringGrid1->Cells[j][i]); // Введення
    b[i][j]=StrToInt(StringGrid2->Cells[j][i]); // матриць
  }
```

```
    //Обчислення матриці-результату
    for (i=0;i<n;i++)
    for (j=0;j<n;j++)
    { c[i][j]=0;
      for (k=0;k<n;k++) c[i][j]+=a[i][k]*b[k][j];
    }
    //Виведення результату
    for (i=0;i<n;i++)
    for (j=0;j<n;j++) StringGrid3->Cells[j][i]=IntToStr(c[i][j]);
  }
```

Приклад 5.20 Скласти схему алгоритму і розробити програму для введення матриці цілих чисел розмірністю 5×5 та обчислення суми елементів в секторі під головною діагоналлю. Замінити елементи побічної діагоналі на цю суму.





Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, j, a[5][5], s=0;
  for (i=0; i<5; i++)
  for (j=0; j<5; j++)
   { a[i][j]=StrToInt(StringGrid1->Cells[j][i]);
     if (i>j) s += a[i][j];
   }
  for (i=0; i<5; i++) a[i][4-i]=s;
```

```
  for (i=0; i<5; i++)
  for (j=0; j<5; j++) StringGrid2->Cells[j][i]=IntToStr(a[i][j]);
  Edit1->Text=IntToStr(s);
}
```

5.3.4 Опрацювання двовимірних масивів у функціях

Як буде зазначено у підрозд. 8.1, при передаванні до функції двовимірних масивів обидві розмірності зазначають чи то як константи, якщо їхнє значення відомо (див. приклади 5.21, 5.23), чи передають як окремі параметри, якщо вони не є відомі на етапі компіляції (див. приклад 8.10).

При опрацюванні у функціях як одновимірних так і двовимірних масивів, які є параметрами цих функцій, насправді у функцію передаються не самі масиви, а вказівники на їхні перші елементи.

Розглянемо кілька варіантів передавання матриці цілих чисел `A[3][4]` у функцію `print()`, яка організовує виведення цієї матриці на екран у консолі.

1) Якщо розміри обох індексів є відомі, то проблем немає:

```
void print(int A[3][4])
{ for (int i=0; i<3; i++)
  { for (int j=0; j<4; j++) cout << ' ' << A[i][j];
    cout << '\n';
} }
```

Матриця і тут передається як вказівник, а розмірності наведено просто для повноти опису. Виклик такої функції може бути таким:

```
int x[3][4];
print(x);
```

2) Перша розмірність для обчислення адреси елемента є неважлива, тому її можна передавати як параметр:

```
void print(int A[][4], int m)
{ for (int i=0; i<m; i++)
  { for (int j=0; j<4; j++) cout << ' ' << A[i][j];
    cout << '\n';
} }
```

Виклик цієї функції:

```
int x[3][4];
print(x,3);
```

3) Самий складний випадок – коли треба передавати обидві розмірності. Наведений нижче код функції є поширеною помилкою:

```
void print(int A[][], int m, int n)   // Помилка!
{ for (int i=0; i<m; i++)
  { for (int j=0; j<n; j++) cout << ' ' << A[i][j];
    cout << '\n';
} }
```

По-перше, опис параметра `A[][]` є неприпустимий, оскільки для обчислення адреси елемента двовимірного масиву слід знати другу розмірність. У такому разі найбільш поширеним і грамотним є застосування динамічних маси-

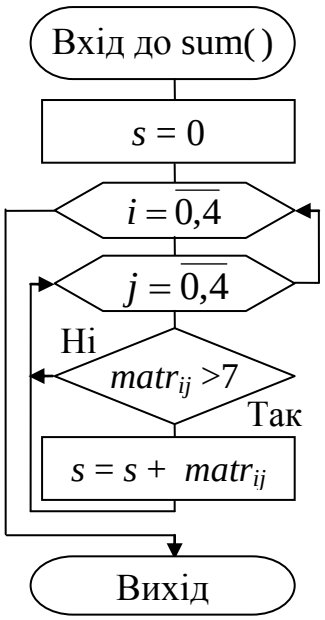
вів, створювання яких більш докладно розглянуто у розд. 6. Наведемо правильний код такої функції:

```
void print(int **A, int m, int n)
{ for (int i=0; i<m; i++)
  { for (int j=0; j<n; j++) cout << ' ' << A[i][j];
    cout << '\n';
} }
```

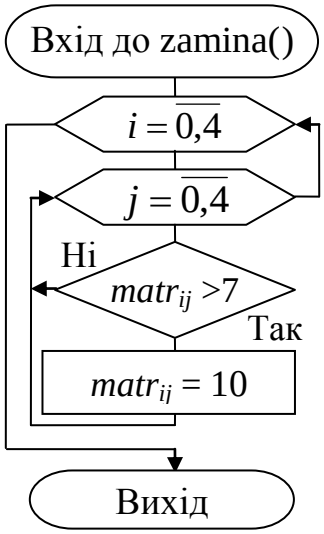
Виклик такої функції матиме вигляд:

```
int **x=new int*[3];
for (i=0; i<3; i++) x[i]=new int[4];
print(x,3,4);
```

Приклад 5.21 Розробити програму для введення матриці розмірністю 5×5 цілих чисел, обчислення суми елементів зі значенням понад 7 і замінити у вихідній матриці ці елементи на значення 10 та обчислити нову суму елементів матриці зі значенням понад 7.

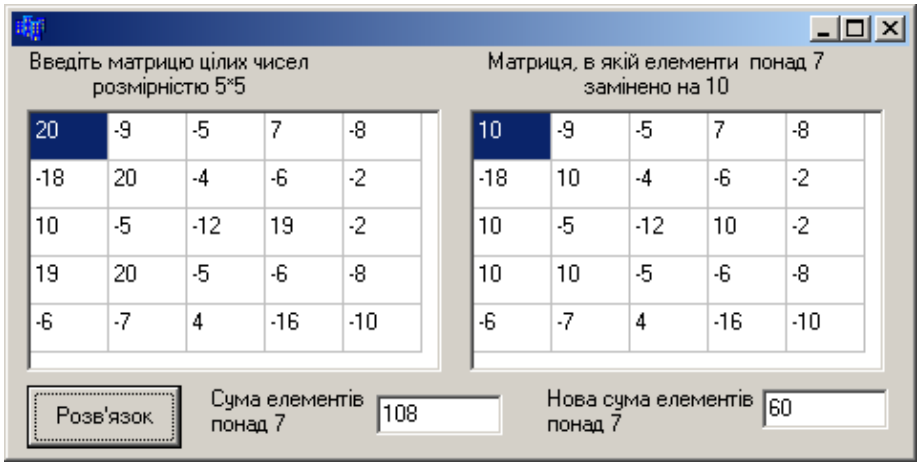


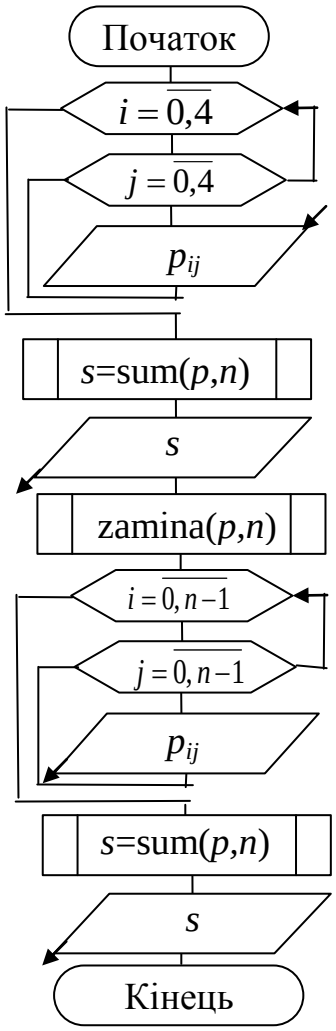
а



б

Блок-схеми функцій:
а – **sum**( ); б – **zamina**( )





Блок-схема основної програми (кнопка “Розв’язок”)

Тексти функцій і програми для командної кнопки “Розв’язок”:

```
int  sum(int matr[5][5])
{  int i, j, s=0;
   for (i=0; i<5; i++)
   for (j=0; j<5; j++)
     if (matr[i][j] > 7)
        s += matr[i][j];
   return s;
}
//---------------------------------------------
void  zamina(int matr[5][5])
{  for (int i=0; i<5; i++)
   for (int j=0; j<5; j++)
     if (matr[i][j] > 7)
       matr[i][j] = 10;
}
//---------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, j, s, p[5][5];
  for (i=0; i<5; i++)
  for (j=0; j<5; j++) p[i][j]=StrToInt(SG1->Cells[j][i]);
  s=sum(p);
  Edit1->Text=IntToStr(s);
  zamina(p);
  for (i=0; i<5; i++)
  for (j=0; j<5; j++) SG2->Cells[j][i]=IntToStr(p[i][j]);
  s=sum(p);
  Edit2->Text=IntToStr(s);
 }
```

Приклад 5.22 Розробити схему алгоритму та програмний проект для:

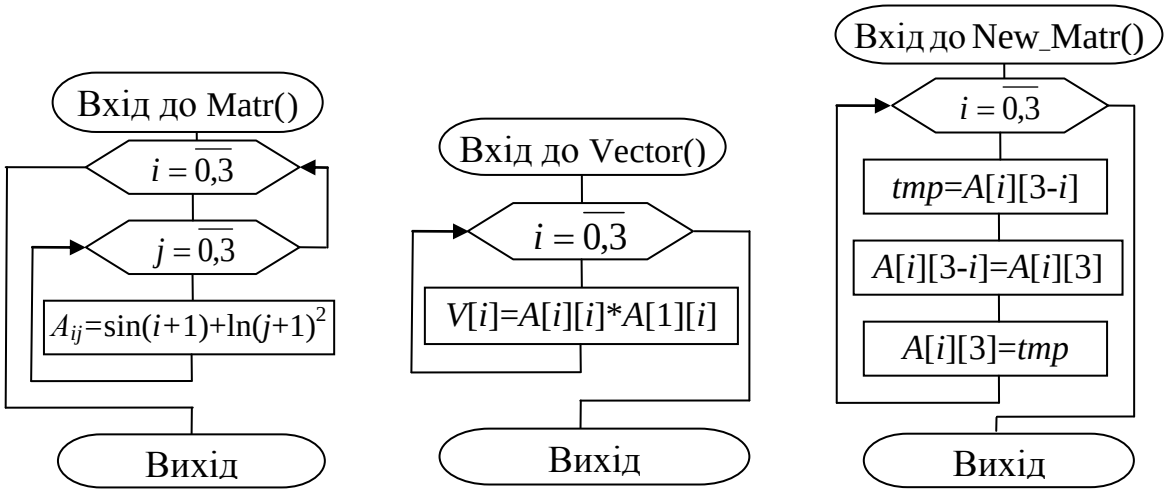
✓ обчислення елементів матриці розмірністю 4×4 за формулою $A_{ij} = \sin(i) + \ln(j^2)$, де $i = \overline{0,4}$, $j = \overline{0,4}$;

✓ обчислення елементів вектора добутків елементів головної діагоналі матриці на елементи другого рядка;

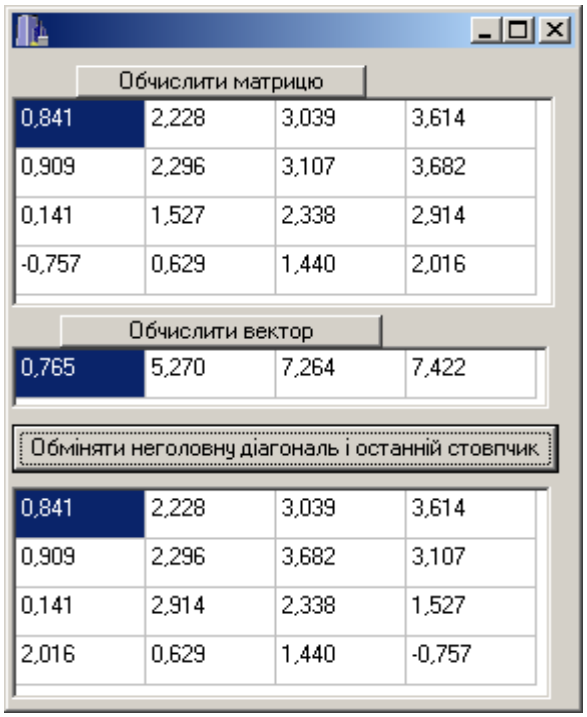
✓ переставляння елементів неголовної діагоналі й останнього стовпчика матриці.

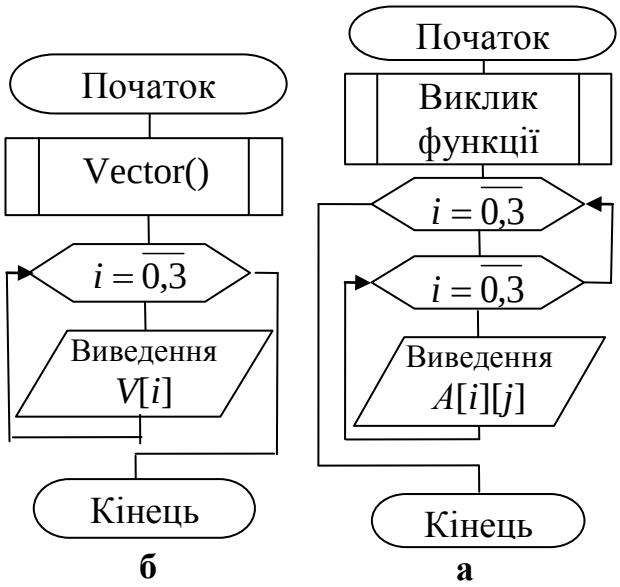
Розв’язок. Схеми алгоритмів функцій та основної програми, а також приклад вигляду форми з результатами роботи наведено нижче.

Для даного прикладу програми доцільно буде змінні, використовувані в різних функціях, оголосити глобально. В такому разі не треба буде передавати їхні значення у якості параметрів функцій, оскільки ці значення видимі в усіх функціях програмного проекту.



Блок-схеми функцій:
а – **Matr()**; б – **Vector()**; в – **New_Matr()**





Блок-схеми для кнопок:
а – `Button1` та `Button3`; б – `Button2`

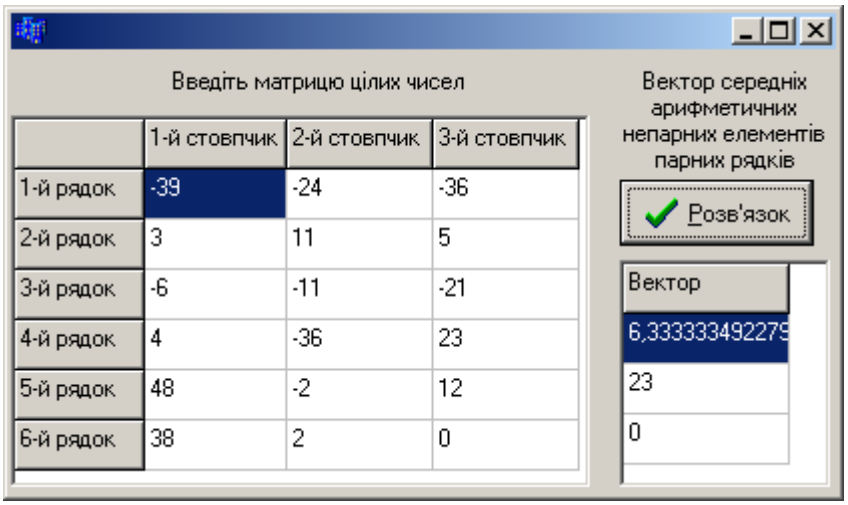
Тексти функцій і програм для усіх командних кнопок:

```
int i, j; float A[4][4], V[4]; // Глобально оголошені змінні
//---------------------------------------------------------------
void Matr()     // Функція обчислення елементів матриці
{ for (i=0; i<4; i++)
  for (j=0; j<4; j++) A[i][j]=sin(i+1)+log((j+1)*(j+1));
}
//---------------------------------------------------------------
void Vector()   // Функція обчислення елементів вектора
{ for (i=0; i<4; i++) V[i]=A[i][i]*A[1][i];
}
```

```
//---------------------------------------------------------------
// Функція  переставляння елементів неголовної діагоналі й останнього стовпчика матриці
void New_Matr()
{ float tmp;
  for (i=0; i<4; i++)
   { tmp = A[i][3-i];
     A[i][3-i] = A[i][3];
     A[i][3] = tmp;
   }
}
//---------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)//Обчислити матрицю
{ Matr();                  // Виклик функції
  for (i=0; i<4; i++)
  for (j=0; j<4; j++)
    StringGrid1->Cells[j][i]=FormatFloat("0.000", A[i][j]);
}
//---------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)//Обчислити вектор
{ Vector();                // Виклик функції
  for (i=0; i<4; i++)
    StringGrid2->Cells[i][0]=FormatFloat("0.000", V[i]);
}
//---------------------------------------------------------------
// Обміняти місцями неголовну діагональ і останній стовпчик
void __fastcall TForm1::Button3Click(TObject *Sender)
{ New_Matr();              // Виклик функції
  for (i=0; i<4; i++)
  for (j=0; j<4; j++)
   StringGrid3->Cells[j][i]=FormatFloat("0.000", A[i][j]);
}
```

Приклад 5.23 Розробити програму для введення матриці цілих чисел розмірністю 6×3 і обчислення елементів вектора середніх арифметичних непарних елементів парних рядків.

Розв'язок організуємо в окремій функції `Vektor()`. Для обчислення елементів вектора спочатку слід організувати цикл для переміщення по парних рядках (2, 4, 6). У кожному з парних рядків, рухаючись по стовпчиках, підсумовуємо тільки парні елементи та їхню кількість. Перед обчисленням середнього арифметичного, тобто перед діленням, перевіряємо можливість відсутності в даному парному рядку непарних елементів, тобто виключаємо поділ на нуль. Змінна k є потрібна для послідовного формування індексів вектора, оскільки їхня нумерація не збігається з нумерацією рядків у матриці.



Текст функції та програми для командної кнопки:

```
void Vektor(int M[6][3], float X[3])
{ int i, j, kol, k = 0;
  for(i=1; i<6; i+=2)
   { X[k] = kol = 0;
     for(j=0; j<3; j++)
       if(M[i][j] % 2 == 1)
        { X[k] += M[i][j];
          kol++;
        }
     if(kol) X[k] /= kol ;
       else X[k] = 0;
     k++;
   }
}
//---------------------------------------------------------------------
void __fastcall TForm1::BitBtn1Click (TObject *Sender)
{ int i, j, A[6][3];   float V[3];
  for(i=0; i<6; i++)
  for(j=0; j<3; j++)
    A[i][j] = StrToInt(SG1->Cells[j + 1][i + 1]);
  Vektor(A, V);          // Виклик функції
  for(i=0; i<3; i++)
    SG2->Cells[0][i + 1] = FloatToStr(V[i]);
}
//---------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{ int i, j;
  for(i=1; i<=6; i++) SG1->Cells[0][i]=IntToStr(i)+"-й рядок";
  for(j=1; j<=3; j++) SG1->Cells[j][0]=IntToStr(j)+"-й стовпчик";
  SG2->Cells[0][0] = "Вектор";
  randomize();
  for(i=1; i<=6; i++)
  for(j=1; j<=3; j++) SG1->Cells[j][i]=IntToStr(50-random(100));
}
```

Питання та завдання для самоконтролю

1) Дайте означення масиву у програмуванні?

2) В який спосіб задають розмірність масиву?

3) Яке призначення індексів масиву? Змінні яких типів можна використовувати для індексів? В який спосіб записують індекси масиву?

4) Які з наведених оголошень одновимірного масиву з 10-ти елементів помилкові і чому?

а) `int A[1..10];` в) `int A[9];`

б) `float A[10];` г) `float A[0, 9];`

5) Запишіть оголошення одновимірного масиву з 25-ти дійсних чисел.

6) Наведіть оператор, який присвоює першому елементу масиву `A` з 20-ти цілих чисел нульове значення.

7) Запишіть оголошення одновимірного константа-масиву з послідовності символів Вашого прізвища.

8) Які компоненти C++ Builder для введення елементів одновимірних масивів на форму Ви знаєте?

9) Запишіть оператор оголошення символьного масиву `S` з 10-ти елементів.

10) Запишіть оператори виведення масиву `W` з 17-ти дійсних чисел до трьох різних компонентів.

11) Оберіть правильні твердження:

а) якщо список ініціалізації містить початкових значень менше за розмірність масиву, це є помилка;

б) оголошення `int A[23];` резервує пам'ять під масив `A` з 23 елементів цілого типу;

в) оголошення `float A = {-1, 2.2, 5};` резервує пам'ять для трьох елементів масиву `A` типу `float`;

г) масив може зберігати дані різних типів.

12) В яких випадках при оголошенні масиву можна не задавати його розмірність?

13) Віднайдіть і розтлумачте помилки у твердженнях:

а) індекси масиву можуть бути якого завгодно типу;

б) при оголошенні масиву програма автоматично ініціалізує його нульовими значеннями;

в) щоб звернутися до конкретного елемента масиву, слід записати ім'я масиву і значення цього елемента;

г) кількість елементів масиву завжди збігається з кількістю займаних ним байтів оперативної пам'яті.

14) Оберіть ім'я змінної, значенням якої є сума елементів масиву в наведених трьох фрагментах програм:

а)
```
s1=0;
for(i=0;i<10;i++)
  s1 += a[i];
```

б)
```
s2=1;
for(i=0;i<10;i++)
  s2 *= a[i];
```

в)
```
s3=0;
for(i=0;i<10;i++)
  if(a[i]>0) s3++;
```

15) Запишіть ім'я змінної, значенням якої є кількість додатних елементів масиву в наведених трьох фрагментах програм:

а)
```
S1=0;
for(i=0;i<10;i++)
 S1 += a[i];
```
б)
```
S2=1;
for(i=0;i<10;i++)
 S2 *= a[i];
```
в)
```
S3=0;
for(i=0;i<10;i++)
 if(a[i]>0) S3++;
```

16) Запишіть ім'я змінної, значенням якої є максимальний з елементів масиву в наведених трьох фрагментах програм:

а)
```
S1=0;
for(i=0;i<10;i++)
 S1 += a[i];
```
б)
```
S2=a[0];
for(i=1;i<10;i++)
if(a[i]>S2)S2=a[i];
```
в)
```
S3=0;
 for(i=0;i<10;i++)
 if(a[i]>0)S3=S3+1;
```

17) Які з наведених оголошень двовимірних масивів неправильні й чому?

а) `int C[1..5, 1..5];`
в) `float C[1..5][1..5];`
б) `float C[5][5];`
г) `int C: [5][5];`

18) В який спосіб розміщуються елементи двовимірних масивів в оперативній пам'яті?

19) Запишіть фрагмент програми для введення з компонента `StringGrid1` елементів матриці `W` з 3-х рядків і 5-ти стовпчиків цілих чисел.

20) Задайте при оголошенні матриці цілих чисел `z` з 3-х рядків і 2-х стовпчиків початкові значення її елементам.

21) Запишіть оператор оголошення символьної матриці `S` розміром 7×3.

22) Під яким номером записано оператор, що обчислює суму елементів головної діагоналі матриці `A` цілих чисел розміром 5×5?

а) `for(i=0, s=0; i<5; i++) s++;`
б) `for(i=0, s=0; i<5; i++) s+=A[i][i];`
в) `for(i=0, s=0; i<5; i++) for(j=0; j<5; j++) s+=A[i][j];`
г) `for(i=0, s=0; i<5; i++) A[i][i]=0;`

23) Запишіть фрагмент програми обчислення кількості елементів масиву `A` з 10-ти цілих чисел, значеннями яких є двоцифрові числа.

24) Запишіть фрагмент програми для обчислення середнього арифметичного, максимального і мінімального елементів масиву `Z` з 15-ти цілих чисел.

25) Оберіть правильні оголошення масивів:

а) `mas[7]={1,3,0,-2,0,1,9};`
б) `char s[10];`
в) `const int m=9; int A[m];`
г) `float C[5]={12,4,82};`
д) `int B[]={1,20,3,0,-1};`
е) `int W[5]={-1,2,40,8,20,9};`

26) Оберіть номер правильного оператора виведення на екран у консольному режимі масиву `mas` з 7-ми цілих чисел:

а) `cout << mas;`
б) `for(int i=0; i<7; i++) cout << mas[i];`
в) `for(int i=0; i<=7; i++) cout << mas[i];`
г) `for(int i=1; i<7; i++) cout << mas[i];`
д) `for(int i=1; i<=7; i++) cout << mas[i];`

27) Віднайдіть і розтлумачте помилки у фрагментах програм для `int s, i, a[10];`:

а)
```
for(s=0, i=0; i<10; i++) { cin >> a[i]; s += a[i];}
cout << "Середнє арифметичне введених чисел =" << s/10;
```

б)
```
for(i=0; i<10; i++) { cin >> a[i]; s += a[i];}
cout << "Сума елементів масиву = " << s;
```

в)
```
for(s=0, i=0; i<5; i+=2) { cin >> a[i]; s += a[i];}
cout << "Сума непарних елементів масиву =" << s/5.;
```

г)
```
for(s=0, i=0; i<5; i++) { cin >> a[i]; s *= a[i];}
cout << "Добуток елементів масиву = " << s;
```

27) Запишіть фрагмент програми обчислення остачі від ділення максимального на мінімальний елемент масиву з 10-ти цілих чисел.

28) Запишіть фрагмент програми упорядкування за спаданням масиву з 20-ти цілих чисел.

29) Чи можна виконувати опрацювання двовимірного масиву, організувавши зовнішній цикл по стовпчиках, а внутрішній – по рядкам?

30) Чи варто організовувати вкладені цикли для опрацювання лише головної діагоналі квадратної матриці?

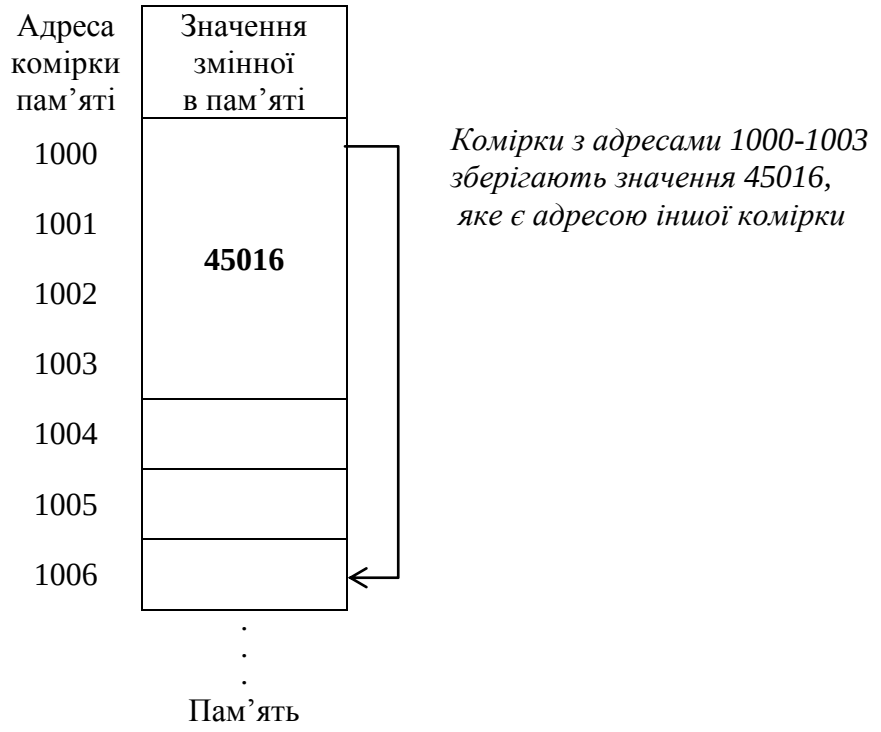
Розділ 6

Вказівники. Динамічна пам'ять

6.1 Вказівники

Пам’ять комп’ютера являє собою сукупність комірок для зберігання інформації, кожна з яких має власний номер. Ці номери називаються *адресами*. Розмір однієї комірки пам’яті становить 1 байт. При оголошенні змінної в пам’яті виділяється ділянка обсягом, відповідним до типу змінної, наприклад 4 байти для типу `int`. Ця ділянка пам’яті пов’язується з іменем змінної. Можна сказати, що адреса змінної – це адреса першої комірки цієї ділянки. Існують спеціальні змінні – *вказівники*, в яких можна зберігати адреси комірок пам’яті, тобто адреси змінних.

Слід звернути увагу на те, що *адреса* і *значення* змінної – це зовсім різні поняття. Зазвичай адреси змінних не є важливими, найголовніше – значення цих змінних. Вказівники використовуються для прямого доступу до даних у пам’яті.



Вказівник оголошується за допомогою *****:

<тип> `*`*<ім’я>*`;`

Тут *тип* – це базовий тип вказівника, котрим може бути який завгодно тип. *Ім’я* є ідентифікатором змінної-вказівника. Слід звернути увагу на те, що *тип* – це не тип вказівника, а тип даних, адресу яких буде записано у вказівнику.

Наприклад, вказівник на пам’ять, в якій зберігатиметься ціле число:

```
int *A;
```

Вказівник на дійсне число:

```
float *B;
```

При цьому змінна A – це адреса ділянки пам'яті, в якій розташоване ціле число, B – адреса ділянки пам'яті, в якій розташоване дійсне число. Обидві змінні – A та B – є вказівниками, тобто їхніми значеннями є адреси. Окрім того, адреси змінних типу int та float займають у пам'яті однакову кількість байтів. Але насправді змінні A та B мають різний тип: A – вказівник на ціле, B – вказівник на дійсне.

Базовий тип вказівника визначає тип об'єкта, адресу якого містить вказівник. Фактично вказівник будь-якого типу може посилатися на яке завгодно місце в пам'яті. Однак операції, які можуть виконуватися з вказівником, суттєво залежать від його типу. Наприклад, якщо оголошено вказівник типу int *, компілятор припускає, що адреса, на яку він посилається, містить змінну типу int, хоча це може бути і не так. Отже, при оголошенні вказівника слід переконатися, що його тип є сумісним з типом об'єкта, на який він буде посилатися.

Вказівникові можна присвоїти лише адресу змінної відповідного типу, оскільки C++ не виконує автоматичного перетворення типів вказівників. Тому такі команди є помилковими:

```
A = B;  // Не можна переприсвоювати один одному вказівники на різні типи;
A = 10; // не можна присвоювати числа вказівникам.
```

У мові C++ визначено дві операції для роботи з вказівниками:

1) ***** – *розадресація*, чи розіменування вказівника;
2) **&** – *адресація*, тобто отримання адреси змінної.

Для прикладу у програмі оголосимо цілу змінну X зі значенням 7:

```
int X = 7;
```

тоді, щоб її *адресу* записати до вказівника A, слід записати команду:

```
A = &X;  // Значенням змінної A є адреса змінної X.
```

Якщо пам'ять, у якій зберігається змінна X, починається з комірки під номером 2000, тоді змінній A після цієї команди буде присвоєно значення 2000.

Тепер до значення 7, яке зберігається у змінній X, можна звернутися не лише за ім'ям X, але й через вказівник. При цьому слід застосувати операцію *розадресації* вказівника (вона позначається символом "*"), наприклад: *A. Цей запис означає: "значення, яке зберігається за адресою, записаною у змінній A". Щоб збільшити на 2 оголошену раніше змінну X, можна записати так:

```
X = X + 2;   /* За ім'ям X комп'ютер визначить адресу ділянки пам'яті, в якій
                вона розташована, візьме значення, що зберігається у пам'яті
                за цією адресою, збільшить його на 2 і запише назад у ту ж
                саму ділянку пам'яті */
```

чи так:

```
*A = (*A) + 2; /* Комп'ютер з ділянки пам'яті за адресою, записаною
                  у змінній A, візьме значення, яке зберігається в ній, збільшить
                  це значення на 2 і запише в ту ж саму ділянку */
```

У першому разі відбувається звертання до змінної з ім'ям X, у другому – до змінної через її адресу, яку записано у змінній-вказівнику A.

Звернімо увагу на те, що "зірочки" при оголошенні вказівника та операції розадресації – це зовсім різні речі, які лише позначаються однаковим символом.

Вказівник, який не вказує на жодне значення, називається порожнім, чи нульовим. Такий вказівник має значення 0 чи NULL.

Вказівники використовуються при роботі з динамічними змінними і при передаванні параметрів до функцій (див. розд. 8).

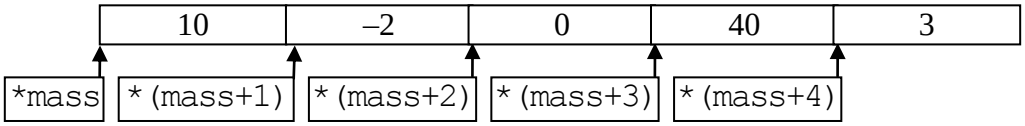
6.2 Вказівники на одновимірні масиви

Масиви і вказівники у C++ тісно пов'язані і можуть використовуватись майже еквівалентно. Ім'я масиву можна сприймати як константний вказівник на перший елемент масиву. Його відмінність від звичайного вказівника полягає у тому, що його неможна модифікувати.

Здійснимо оголошення масиву `mass` з п'яти цілих чисел з ініціалізацією значень елементів і вказівника на ціле `ptr`:

```
int mass[5] = {10,-2,0,40,3}, *ptr;
```

При такому оголошенні масиву пам'ять виділяється не лише для п'яти елементів масиву, а й для вказівника з ім'ям `mass`, значення якого дорівнює адресі першого елемента масиву `mass[0]`, тобто сам масив залишається безіменним, а доступ до елементів масиву здійснюється через вказівник з ім'ям `mass`.



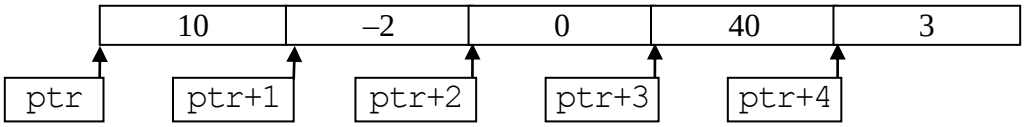
Для того, щоби звернутися до 3-го елементу цього масиву, можна записати чи то `mass[2]` чи `*(mass+2)`. При реалізації на комп'ютері перший з цих способів зводиться до другого, тобто індексний вираз перетвориться до адресного. Оскільки операції над вказівниками виконуються швидше, тому, якщо елементи масиву обробляються почергово, то доцільніше використовувати другий спосіб. Якщо ж вибір елементів є випадковим, то, щоб уникнути помилок, прийнятнішим є перший спосіб. Крім того, перший спосіб є більш наочним і звичним для сприйняття, що сприяє кращій читабельності програм.

Оскільки ім'я масиву є вказівником на перший елемент масиву, можна надати вказівнику адресу першого елементу масиву за допомогою оператора:

```
ptr = mass;
```

Цей запис є еквівалентним присвоюванню адреси першого елемента масиву (тобто до елемента з нульовим індексом)у вигляді оператора

```
ptr = &mass[0];
```



Після цього звернутися до першого елемента масиву і надати йому значення 2 можна будь-яким з шести операторів:

```
* mass = 2;     mass[0] = 2;        *(mass+0) = 2;
```

```
* ptr = 2;      ptr[0] = 2;          *(ptr+0) = 2;
```

Усі ці оператори за дією є тотожними, але швидше за всі виконуватимуться присвоювання `*mass = 2` та `*ptr = 2`, оскільки в них не треба виконувати операції додавання.

Вказівники можна індексувати так само, як і масиви. Наприклад, вираз `ptr[3]` посилається до четвертого елемента масиву `mass[3]`.

Зауважимо, що не слід плутати такі оголошення:

```
int *p1[10];                    // приклад 1
int (*p2)[10];                  // приклад 2
```

У першому прикладі оголошено масив вказівників з ім'ям `p1`. Масив складається з 10 елементів, кожний з яких є вказівником на змінну типу `int`.

У другому прикладі оголошено змінну-вказівник з ім'ям `p2`, яка вказує на масив з 10 цілих чисел типу `int`.

6.3 Арифметика вказівників

Вказівники можуть застосовуватися як операнди в арифметичних виразах, виразах присвоювання і виразах порівняння. Проте, не усі операції, зазвичай використовувані у таких виразах, дозволені для вказівників.

З вказівниками може виконуватися обмежена кількість арифметичних операцій. Вказівник можна збільшувати (++), зменшувати (––), додавати до вказівника цілі числа (+ чи +=), віднімати від нього цілі числа (– чи –=) або віднімати один вказівник від іншого.

Здебільшого арифметичні операції застосовуються до вказівників на масиви. Більше того, арифметика вказівників втрачає всякий сенс, якщо вона виконується не над вказівниками на масив. Змінення (збільшення/зменшення) значення вказівника на масив по суті є зсувом адреси у межах цього масиву. Перетворення цілої величини до адресного зсуву припускає, що у межах зсуву щільно розташовані елементи однакового розміру. Це припущення справедливо для елементів масиву, оскільки масив визначається як набір величин однаково типу, а його елементи розташовані у суміжних комірках пам'яті. Додавання і віднімання адрес, які посилаються на будь-які величини, крім елементів масиву, дає непередбачуваний результат, оскільки в такому разі не гарантується щільне заповнення пам'яті.

Розглянемо на прикладах дію арифметичних операцій на вказівники:

✓ *Додавання й віднімання цілого числа до вказівника* відрізняється від звичайної арифметики. При їх виконанні адреса, яка зберігається у вказівникові, змінюється кратно до розміру даних, на які вказує вказівник. Для оголошеного вище масиву `mass` і вказівника `ptr`, який вказує на початок масиву, команда

```
ptr += 3;
```

збільшить адресу, яка міститься у `ptr`, на 4*3=12 байти, і `ptr` вказуватиме на четвертий елемент масиву зі значенням 40.

Загальна формула, яка обчислює значення вказівника при додаванні (відніманні) цілого числа `N`:

*<Нова адреса> = <стара адреса> +/- <розмір базового типа>*N;*

✓ *Операції інкремента* (++) і *декремента* (--), тобто збільшення чи зменшення вказівника на 1. При цьому адреса, яка зберігається у цьому вказівнику, збільшується на розмір базового типу в байтах. Наприклад, команда

```
ptr++;
```

збільшить його значення на 4, оскільки розмір змінної типа `int` зазвичай дорівнює 4, а сам вказівник вказуватиме на наступний елемент масиву.

Зменшення вказівника на 1 означає віднімання розміру в байтах базового типу від адреси, яка зберігається у вказівнику.

✓ *Різниця однотипних вказівників*. Вказівники можна віднімати один від одного. Наприклад, якщо `ptr1` вказує на перший елемент масиву `mass`, а вказівник `ptr2` – на четвертий, то результат виразу `ptr2 – ptr1` має тип `int` і дорівнює 3 – різниці індексів елементів, на які вказують ці вказівники. І так буде, не дивлячись на те, що адреси, які містяться у цих вказівниках, розрізняються на 12 байтів (якщо елемент масиву займає 4 байти). Тобто, результат такої операції дорівнює кількості елементів вихідного типу між зменшуваним і від'ємником, причому якщо перша адреса молодше, то результат має від'ємне значення.

Загальна формула при відніманні однотипних вказівників:

<Кількість елементів>=(<перший вказівник>–<другий вказівник>)/<розмір типу>

Наприклад:

```
int *ptr1, *ptr2, mass[5]={10,-2,0,40,3}, i;
ptr1 = a + 4;
ptr2 = a + 9;
i = ptr1 - ptr2;  /*   дорівнює –5    */
i = ptr2 - ptr1;  /*   дорівнює  5    */
```

✓ *При порівнянні вказівників* порівнюються адреси, які містяться у вказівниках, а результат порівняння дорівнює 0 (`false`) або 1 (`true`). Порівняння вказівників операціями ">", "<", ">=", "<=" мають сенс лише для вказівників на один той самий масив. Проте, операції відношення "==" та "!=" мають сенс для будь-яких вказівників. При цьому вказівники є рівнозначними, якщо вони вказують на одну і ту саму адресу в пам'яті. Наприклад, для розглянутих вище вказівників `ptr1` та `ptr2` у команді

```
if(ptr1 > ptr2) mass[3]=4;
```

значення `ptr1` є менше за значення `ptr2` і тому оператор `mass[3]=4` не буде виконаний.

Приклад порівняння вказівника `ptr` з ненульовим значенням:

```
if(ptr != NULL) ShowMessage("Ненульовий вказівник");
```

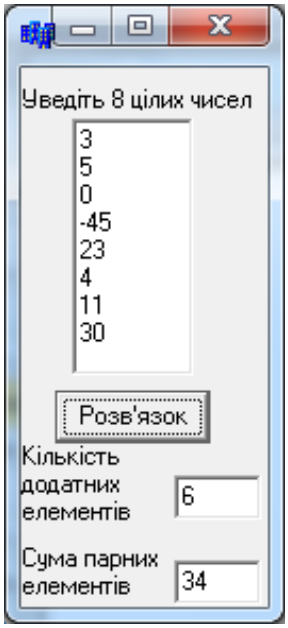
Приклад 6.1 В одновимірному масиві з 8 цілих чисел обчислити кількість додатних та суму парних елементів.

Розв'язок. Для розв'язання організуємо функцію, параметрами якої буде вказівник на масив, розмірність масиву та адреса одного з обчислюваних ре-

зультатів. Інший результат передаватимемо з функції через оператор `return`. Звернення `*(a+i)` є аналогічним до `a[i]` і тому можна однаково використовувати як те, так і те.

Текст функції та основної програми:

```
int fun(int *a, int n, int &s)
{ int k=0; s=0;
  for(int i=0; i<n; i++)
    { if (*(a+i) > 0) k++;
      if (*(a+i)%2 == 0) s += *(a+i);
    }
  return k;  }
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, a[8], sum, kol;
  for(i=0; i<8; i++)
    *(a+i) = StrToInt(Memo1->Lines->Strings[i]);
  kol = fun(a,8,sum);
  Edit1->Text = IntToStr(kol);
  Edit2->Text = IntToStr(sum);
}
```



6.4 Вказівники на багатовимірні масиви

Вказівники на багатовимірні масиви у мові С++ – це масиви масивів, тобто такі масиви, елементами яких є масиви. При оголошені таких масивів у пам'яті комп'ютера створюється декілька різних об'єктів. Приміром, при виконанні оголошення двовимірного масиву `int arr[4][3]` у пам'яті виділяється ділянка для зберігання значення змінної `arr`, яка є вказівником на масив з чотирьох вказівників. Для цього масиву з чотирьох вказівників теж виділяється пам'ять. Кожний з цих чотирьох вказівників містить адресу масиву з трьох елементів типу `int`, і, отже, у пам'яті комп'ютера виділяється чотири ділянки для зберігання чотирьох масивів чисел типу `int`, кожна з яких складається з трьох елементів. Такий розподіл пам'яті показано на схемі:

```
arr
 ↓
arr[0]   →   arr[0][0]   arr[0][1]   arr[0][2]
arr[1]   →   arr[1][0]   arr[1][1]   arr[1][2]
arr[2]   →   arr[2][0]   arr[2][1]   arr[2][2]
arr[3]   →   arr[3][0]   arr[3][1]   arr[3][2]
```

Отже, оголошення `arr[4][3]` породжує в програмі три різних об'єкти: вказівник з ідентифікатором `arr`, безіменний масив з чотирьох вказівників і безіменний масив з дванадцяти чисел типу `int`. Для доступу до безіменних масивів використовується адресні вирази з вказівником `arr`. Доступ до елементів масиву вказівників здійснюється із зазначенням одного індексного виразу у формі `arr[2]` чи `*(arr+2)`. Для доступу до елементів двовимірного масиву чисел типу

int має бути використано два індексні вирази у формі arr[1][2] чи еквівалентних їй *(*(arr+1)+2) та (*(arr+1))[2].

Нагадаємо, що елементи двовимірних масивів розміщуються у пам'яті підряд і між його рядками немає ніяких проміжків (див. п. 5.3.1). Такий порядок дає можливість звертатися до будь-якого елементу багатовимірного масиву, використовуючи адресу його початкового елемента та лише один індексний вираз. Так для матриці arr звертання *(arr) є посиланням на елемент arr[0][0], звертання *(arr+2) – на елемент arr[0][2], а звертання *(arr+3) – на елемент arr[1][0] і т. д. Тобто, застосовуючи оператори циклу, можна організувати поелементне опрацювання елементів матриці, приміром введення усієї матриці з компонента StringGrid1:

```
for(int i=0;i<4;i++)
for(int j=0;j<3;j++)
   *(arr+i*4+j)=StrToInt(StringGrid1->Cells[j][i]);
```

Тут вираз *(arr+i*4+j) є аналогічний до arr[i][j].

Розміщення тривимірного масиву відбувається аналогічно й оголошення

```
float W[3][4][5];
```

породжує у програмі крім самого тривимірного масиву з шістдесяти елементів типу float масив з чотирьох вказівників на тип float, масив з трьох вказівників на масив вказівників на тип float, і вказівник на масив масивів вказівників на тип float.

Для звертання до елемента W[2][3][4] тривимірного масиву також можна використовувати вказівник, оголошений як float *p=W[0][0] з одним індексним виразом у формі p[2*4*5+3*5+4] чи p[59].

Приклад 6.2 Увести матрицю цілих чисел розміром 4×4 і транспонувати її.

Розв'язок. Щоби створити функцію, яка прийматиме у якості аргументу двовимірний масив, слід враховувати, що ім'я масиву опрацьовується як його адреса. Тому відповідний формальний параметр є вказівником, як і для одновимірних масивів. Секрет успіху криється у правильному оголошенні вказівника. Наведемо два різні способи такої організації функції до якої для транспонування передається матриця.

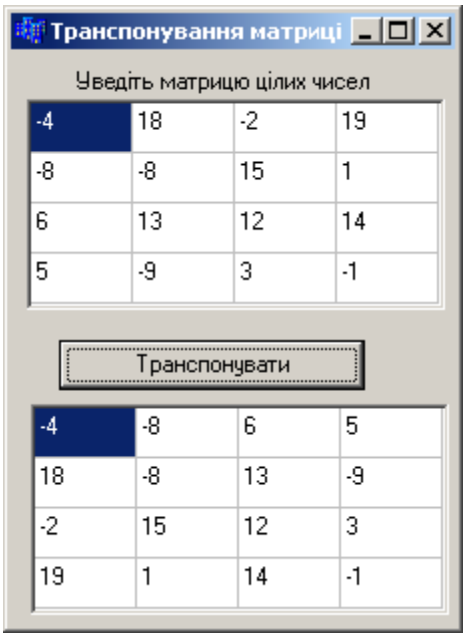
Перший спосіб розв'язання

У цьому способі, щоби кожного разу не зазначати розмірність квадратної матриці, створимо на початку відповідну константу n, і тоді зникає потреба передавання окремим параметром розмірності цієї матриці до функції.

У наведеній тут функції транспонування матриці transp() матриця передається до функції як вказівник і тоді більш звичне звертання до елементів матриці u[i][j] слід замінити виразом *(u+n*i+j) чи u[n*i+j].

Текст функції та її виклику в основній програмі:

```
const int n=4;
// Функція транспонування матриці
void transp(int *u)
{ int i,j;
  for(i=0;i<n-1;i++)
  for(j=i+1;j<n;j++)
  { int p=*(u+n*i+j);
    *(u+n*i+j)=*(u+n*j+i);
    *(u+n*j+i)=p;
} }
void __fastcall TForm1::Button1Click
                              (TObject *Sender)
{ int i, j, a[n][n];
for(i=0;i<n;i++)
for(j=0;j<n;j++)
 a[i][j]=StrToInt(StringGrid1->Cells[j][i]);
transp(&a[0][0]);
for(i=0;i<n;i++)
for(j=0;j<n;j++) StringGrid2->Cells[j][i]=IntToStr(a[i][j]);
}
```



Другий спосіб розв'язання

У цьому способі, на відміну від першого, передаватимемо розмірність квадратної матриці окремим параметром до функції. Крім того, використаємо ту особливість, що наша матриця `u` по суті є вказівником на масив з чотирьох значень типу `int`, і тому для функції є припустимим такий прототип:

```
void transp(int (*u)[4], int n);
```

У цьому запису дужки є обов'язковими, оскільки оголошення `int *u[4]` є масивом чотирьох вказівників на данні типу `int`, а параметр функції не може бути масивом. Існує альтернативний формат запису цього прототипу, який має таке саме тлумачення, але сприймається для розуміння краще:

```
void transp(int u[][4], int n);
```

Обидва варіанти означають, що параметр `u` є вказівником на масив чотирьох значень типу `int`, а не безпосередньо масивом. Число 4 у квадратних дужках задає кількість стовпців, а кількість рядків можна передавати окремим параметром.

Текст функції та її виклику в основній програмі:

```
void transp(int (*u)[4], int n)
{ for(int i=0;i<n-1;i++)
  for(int j=i+1;j<n;j++)
  { int p=u[i][j];
    u[i][j]=u[j][i];
    u[j][i]=p;
  }
}
void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{ const int n=4;
  int i, j, a[n][n];
  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
   a[i][j]=StrToInt(StringGrid1->Cells[j][i]);
  transp(a, n);
  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
   StringGrid2->Cells[j][i]= IntToStr(a[i][j]);
}
```

Третій різновид передавання до функції двовимірних масивів, зазначений у п. 5.3.4 наведено у розд. 6.7 після того як буде розглянуто оператори динамічного виділення пам'яті для таких масивів.

6.5 Динамічна пам'ять

Окрім звичайної пам'яті (стека), в якій автоматично розміщуються змінні за їхнього оголошення, існує ще й *динамічна пам'ять* (heap − купа), в якій змінні можуть розміщуватися динамічно. Це означає, що пам'ять виділяється під час виконання програми, й лише тоді, коли у програмі зустрінеться спеціальна інструкція. Основна потреба у динамічному виділенні пам'яті виникає, коли розмір або кількість даних заздалегідь є невідомі, а визначаються в процесі виконання програми.

У C++ існує кілька команд для виділення динамічної пам'яті. Найчастіше використовується оператор `new`, який у загальному вигляді записується як:

*<тип> *<ім'я_вказівника>* = **new** *<тип>*;

Приклади:

1) `int *p = new int;`

Тут виділяється місце в пам'яті під ціле число й адреса цієї ділянки пам'яті записується у змінну-вказівник `p`. Звернутися до цього числа можна буде через вказівник на нього: `*p = 2;`

2) `int *p = new int (5);`

Ця команда не лише виділить місце у пам'яті під ціле число, а й запише в цю ділянку пам'яті значення 5. Адреса першої комірки виділеної ділянки пам'яті присвоюється змінній-вказівнику `p`. Звернутися до числа, на яке вказує `p`, можна аналогічно до попереднього прикладу. Приміром, щоб збільшити таке число на 2, слід написати: `(*p) += 2;`

3) `int *p = new int [5];`

У цьому разі виділяється пам'ять під 5 цілих чисел, тобто фактично створюється так званий *динамічний масив* з 5-ти елементів. Звернутися до кожного з цих чисел можна за його номером: `p[0]`, `p[1]` і т. д. або через вказівник: `*p` − те ж саме, що `p[0]`; `*(p+1)` − те ж саме, що `p[1]` і т. д.

Пам'ять, яку було виділено динамічно, автоматично не звільнюється, тому програміст повинен обов'язково звільнити її самостійно за допомогою спе-

ціальної команди. При виділенні пам'яті за допомогою оператора `new`, для звільнення пам'яті використовується `delete`:

`delete` <*вказівник*>`;`

Наприклад:

```
delete a;
```

Якщо оператором `new` було виділено пам'ять під кілька значень водночас (як у розглянутому вище прикладі 3), використовується форма команди `delete`

`delete []`<*вказівник*>`;`

Квадратні дужки повинні бути порожніми, операційна система контролює кількість виділеної пам'яті і при звільненні їй відома потрібна кількість байтів. Наприклад:

```
delete [] p;
```

Окрім операторів `new` та `delete`, існують функції, які перейшли до C++ з C, але вони використовуються на практиці набагато рідше.

Функція

```
void *malloc(size_t size);
```

(від англ. "memory allocation" – виділення пам'яті) робить запит до ядра операційної системи щодо виділення ділянки пам'яті заданої кількості байтів. Єдиний аргумент цієї функции `size` – кількість байтів, яку треба виділити. Функція повертає вказівник на початок виділеної пам'яті. Якщо для розміщення заданої кількості байтів є недостатньо пам'яті функція `malloc()` повертає `NULL`. Вміст ділянки лишається незмінним, тобто там може залишитися "бруд". Якщо аргумент `size` дорівнює 0, функція повертає `NULL`. Наприклад, команда

```
int *a = (int*) malloc(sizeof(int));
```

виділяє пам'ять під ціле число й адресу початку цієї ділянки пам'яті записує у вказівник `a`.

Виділення пам'яті під 10 дійсних чисел за допомогою цієї функції:

```
float *a = (float*) malloc(sizeof(float)*10);
```

Функція

```
void * calloc(size_t num, size_t size);
```

виділяє блок пам'яті розміром `num×size` (під `num` елементів по `size` байтів кожен) і повертає вказівник на виділений блок. Кожен елемент виділеного блока ініціалізується нульовим значенням (на відміну від функції `malloc`). Функція `calloc()` повертає `NULL`, якщо не вистачає пам'яті для виділення нового блока, або якщо значення `num` чи `size` дорівнюють 0.

Виділення пам'яті під 10 дійсних чисел за допомогою функції `calloc()`:

```
float *a = (float*) calloc(10,sizeof(float));
```

Функція

```
void *realloc(*bl, size);
```

змінює розмір виділеного раніш блока пам'яті з адресою `*bl` на новий обсяг, який становитиме `size` байтів. Якщо змінення відбулося успішно, функція `realloc()` повертає вказівник на виділену ділянку пам'яті, а інакше повертається `NULL`. Якщо `*bl` є `NULL`, функція `realloc()` виконує такі самі дії, що й `malloc()`. Якщо `size` є 0, виділений за адресою `*bl` блок пам'яті звільнюється – і функція повертає `NULL`. Для ілюстрації роботи функції `realloc()` наведемо приклад змінення розміру раніш виділеної пам'яті під одновимірний масив з 10-ти дійсних елементів до 20-ти елементів:

```
a = (float*) realloc(a, 20);
```

Пам'ять, виділену за допомогою функцій `malloc()` і `calloc()`, звільнюють за допомогою функції `free()`:

```
void free(*bl);
```

де `*bl` – адреса початку виділеної раніше пам'яті, наприклад:

```
free(a);
```

Використання згаданих функцій разом з вказівниками надає можливість керувати розподілом динамічної пам'яті.

Якщо не звільняти динамічно виділену пам'ять, коли вона стає більш не потрібною, у системі може виникнути нестача вільної пам'яті. Іноді це називають «витіком пам'яті».

6.6 Динамічні одновимірні масиви

Відмінності *динамічного масиву* від звичайного полягають у тому, що:

✓ память під динамічний масив виділяється динамічно за допомогою вищерозглянутих функцій;

✓ кількість елементів динамічного масиву може бути задано змінною (але у програмі її неодмінно має бути визначено до виділення пам'яті під масив).

Синтаксис оголошення динамічного одновимірного масиву за допомогою оператора `new` є такий:

<базовий тип> `*`*<ім'я>*`=new` *<базовий тип>*`[`*<кількість елементів>*`];`

Приклад оголошення дійсного динамічного масиву зі змінною кількістю елементів:

```
int N;
N=StrToInt(Edit1->Text);
float *a=new float [N];
```

Тут кількість елементів масиву є змінною і вводиться з клавіатури з `Edit1` перед оголошенням масиву. Звільнення пам'яті від цього масиву `a` матиме вигляд:

```
delete []a;
```

Синтаксис оголошення динамічного одновимірного масиву за допомогою функції `malloc()` є такий:

<тип> `*`*<ім'я>* `= (`*<тип>*`*) malloc(sizeof(`*<тип>*`)*`*<кільк. ел.>*`);`

Приклад оголошення дійсного динамічного масиву зі змінною кількістю елементів за допомогою функції `malloc()`:

```
int N;
N=StrToInt(Edit1->Text);
float *a=(float *)malloc(sizeof(float)*N);
```

Приклад оголошення дійсного динамічного масиву зі змінною кількістю елементів за допомогою функції `calloc()`:

```
int N;
N=StrToInt(Edit1->Text);
float *a=(float *)calloc(sizeof(float), N);
```

Звільнення пам'яті з-під цього масиву `a`:

```
free (a);
```

Приклади програм з динамічними масивами

Приклад 6.3 Увести дійсні числа у `Memo` і створити з них масив ненульових чисел. Обчислити кількість від'ємних елементів цього масиву з непарними індексами.

Розв'язок. Оскільки заздалегідь невідомо кількість чисел, які будуть введені у `Memo1`, тому має сенс організувати динамічний масив. При розв'язанні подібних завдань зазвичай дотримуються такої послідовності дій:

1) Визначити кількість елементів майбутнього масиву. У даному прикладі для цього треба зчитати послідовно всі числа з `Memo` і обчислити кількість ненульових з них.

2) Оголосити динамічний масив і виділити під нього пам'ять, використовуючи чи то `new`, чи `malloc()`, чи `calloc()`.

3) Заповнити масив значеннями. У розглядуваному прикладі для цього слід ще раз послідовно зчитати всі числа з `Memo` і присвоїти елементам масиву лише ненульові з цих чисел. Слід звернути увагу, що при заповнюванні масиву числами з `Memo1`, індекс елемента масиву, який заповнюється на поточному кроці, й індекс рядка `Memo1`, з якого зчитується поточне число, не збігаються. Тому для індексів масиву створено окрему змінну `j`.

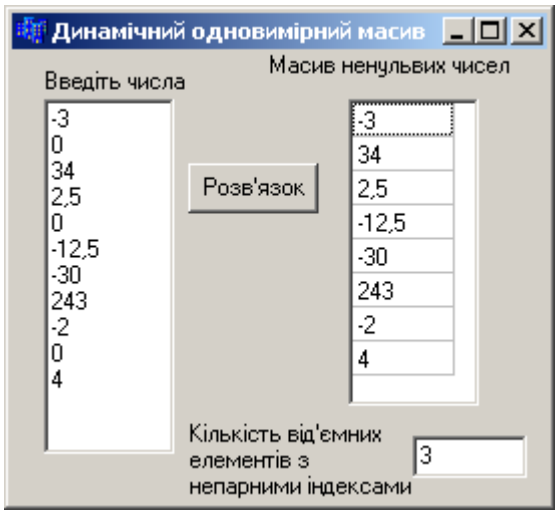
4) Вивести створений масив. У наведеній нижче програмі виведення масиву організовано у `StrinGrid1`, який при створюванні форми для зручності перейменовано на `SG1`.

Наведемо два варіанти розв'язання цього завдання з використанням оператора `new` та функції `malloc()`.

Перший спосіб розв'язання з використанням оператора `new`:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, j=0;        // i – індекс числа Memo1, j – індекс елемента масиву,
  int k=0;           // змінна k для кількості від'ємних елементів масиву.
  int n=Memo1->Lines->Count;   // Кількість чисел в Memo1
  int N=0; float x;
```

```
  for(i=0; i<n; i++) // Зчитування у циклі чисел з Memo1
    { x= StrToFloat(Memo1->Lines->Strings[i]);
      if (x!=0) N++;}      // й обчислення кількості ненульових чисел.
  SG1->RowCount=N;         // Встановлення  кількості рядків у SG1
  float *a=new float[N];  // Виділення пам'яті під динамічний масив
  for(i=0; i<n; i++)
  { x= StrToFloat(Memo1->Lines->Strings[i]);
    if(x!=0)               // Записування ненульових чисел до масиву
     { a[j]=StrToFloat(Memo1->Lines->Strings[i]);
       SG1->Cells[0][j]=a[j];
   // Збільшення індекса елемента масиву.
       j++;
     }
  }
  // Обчислення кількості від'ємних
  // елементів з непарними індексами.
  for(i=0; i<N; i++)
   if(a[i]<0 && (i+1)%2!=0) k++;
  Edit1->Text=IntToStr(k);
  // Звільнення пам'яті від масиву.
  delete []a;
}
```



Другий спосіб розв'язання з використанням функції `calloc()`:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, j=0;
  int k=0;
  int n=Memo1->Lines->Count;
  int N=0; float x;
  for (i=0; i<n; i++)
  { x= StrToFloat(Memo1->Lines->Strings[i]);
    if (x!=0) N++;
  }
  SG1->RowCount=N;
  // Виділення пам'яті під динамічний масив
  float *a=(float*)calloc(N, sizeof(float));
  for(i=0; i<n; i++)
  { x= StrToFloat(Memo1->Lines->Strings[i]);
    if(x!=0)
    { a[j]=StrToFloat(Memo1->Lines->Strings[i]);
      j++;
  }  }
  for(i=0; i<N; i++)
  { SG1->Cells[0][i]=a[i];      // Виведення масиву до SG1
    if(a[i]<0 && (i+1)%2!=0) k++;
  }
  Edit1->Text=IntToStr(k);
  free (a);                    // Звільнення пам'яті від масиву
}
```

Існує можливість звертатися до елементів масиву без індексації за допомогою вказівників. У циклі за допомогою індексу поточний елемент масиву записується як `a[i]`. Згадаймо, що ім'я масиву `a` можна використовувати як вказівник на початок (нульовий елемент) масиву. Тоді, згідно з арифметикою вказівників, `a+i` – це вказівник на елемент, який міститься на `i` комірок далі від початку масиву, тобто вказівник на `a[i]`. Значення елемента `a[i]` можна записати за допомогою операції розадресації: `*(a+i)`. Якщо у програмі замінити всі звертання до елементів масиву за допомогою індексу на звертання до елементів за допомогою вказівника, то програма першого способу розв'язання даного прикладу матиме вигляд

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, j=0, k=0, N=0;
  int n=Memo1->Lines->Count;
  for (i=0; i<n; i++)
  { x= StrToFloat(Memo1->Lines->Strings[i]);
    if(x!=0) N++;
  }
  SG1->RowCount=N;
  float *a=new float[N];
  for(i=0; i<N; i++)
  { x= StrToFloat(Memo1->Lines->Strings[i]);
    if(x!=0)
    { *(a+j)=StrToFloat(Memo1->Lines->Strings[i]);
      j++;
  } }
  for(i=0; i<N; i++)
  { SG1->Cells[0][i]=*(a+j);
    if(*(a+i)<0 && i%2!=0) k++;
  }
  Edit1->Text=IntToStr(k);
  delete []a;
}
```

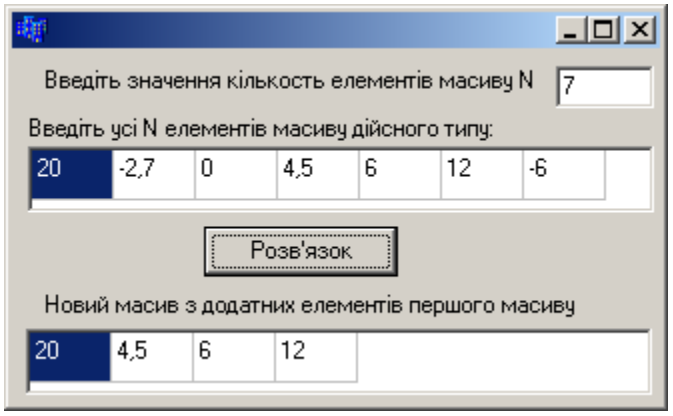
Приклад 6.4 Увести ціле число `N` та масив з `N` дійсних чисел у `StringGrid` і створити новий масив з додатних елементів першого масиву.

Розв'язок. Оскільки кількість елементів має бути введено з форми, заздалегідь вона є невідома, тому пам'ять під обидва масиви слід виділяти динамічно.

Текст програми:

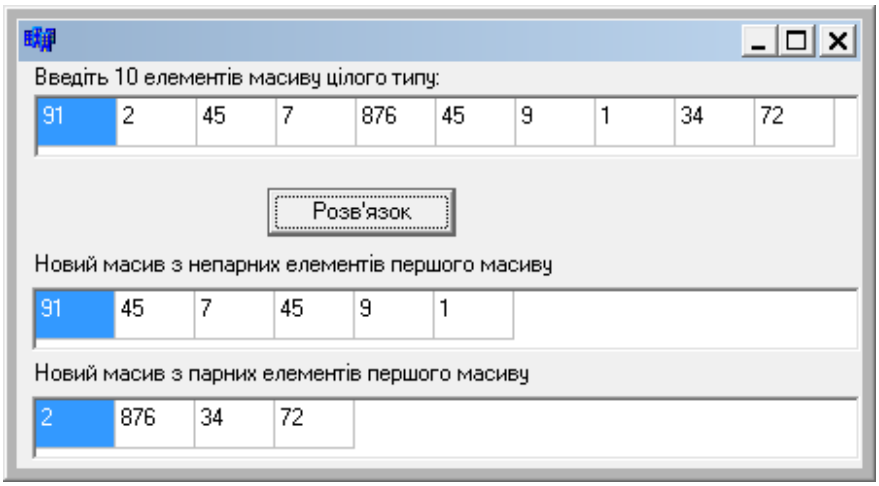
```
// Подія Edit1Change виникає при змінюванні вмісту вікна  Edit1.
void __fastcall TForm1::Edit1Change(TObject *Sender)
{ int N=StrToInt(Edit1->Text); // Введення кількості елементів масиву
  SG1->ColCount=N;             // для визначення кількості комірок у SG1.
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int N=StrToInt(Edit1->Text);
```

```
  SG1->ColCount=N;
  // Виділення пам'яті під масив a; у цей момент ще неможна виділити пам'ять
  // під масив b, оскільки кількість його майбутніх елементів ще є невідома.
  float *a=new float[N];
  int int i, k=0, j=0;
  for(i=0; i<N; i++) a[i]=StrToFloat(SG1->Cells[i][0]);
  for(i=0; i<N; i++) if(a[i]>0) k++;
  // Виділення пам'яті під масив b
  float *b=new float[k];
  SG2->ColCount =k;
  for(i=0; i<N; i++)
   if(a[i]>0){b[j]=a[i]; j++;}
  for(i=0; i<k; i++)
   SG2->Cells[i][0] =
              FloatToStr(b[i]);
  delete []a;
  delete []b;
}
```



Приклад 6.5 Увести масив з 10-ти цілих чисел і створити з нього два нових масиви: перший масив з непарних елементів, а другий – з парних.

Розв'язок. Кількість елементів вихідного масиву є відома (10), тому цей масив у програмі буде оголошено в звичайний спосіб: `int a[10];`. Два нових масиви `b1` та `b2` доцільно створити динамічно, оскільки кількість елементів цих масивів `n1` та `n2` буде обчислено у програмі.



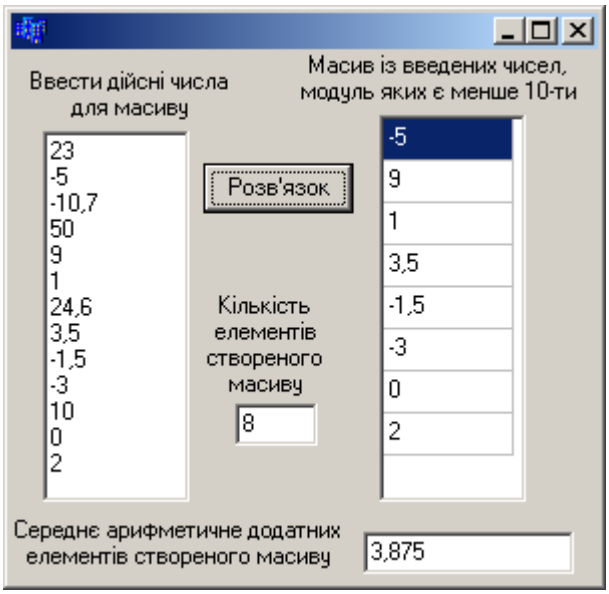
Текст програми

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int a[10];
  int i, n1=0, n2=0, j1=0, j2=0;
  for(i=0; i<10; i++) a[i]=StrToInt(SG1->Cells[i][0]);
  for(i=0; i<10; i++)
   if(a[i]%2!=0) n1++;     // Обчислення кількості непарних елементів n1
    else n2++;             // та кількості парних елементів n2.
  SG2->ColCount=n1;        // Встановлення потрібної кількості комірок
  SG3->ColCount=n2;        // у компонентах для виведення створюваних масивів.
```

```
  int *b1 = new int [n1],  *b2 = new int [n2];
  for(i=0; i<10; i++)     // Формування масивів b1 та b2
   if(a[i]%2!=0){ b1[j1]=a[i]; j1++;}
    else { b2[j2]=a[i]; j2++;}
  // Виведення масивів в окремих циклах, оскільки масиви мають різну кількість елементів
  for(i=0; i<n1; i++) SG2->Cells[i][0]=IntToStr(b1[i]);
  for(i=0; i<n2; i++) SG3->Cells[i][0]=IntToStr(b2[i]);
  // Звільнення пам'яті від b1 та b2
  delete []b1;  delete []b2;
}
```

Приклад 6.6 Увести дійсні числа в Memo і створити масив з тих чисел, модуль яких є менше за 10. Для обчислення середнього арифметичного додатних елементів створеного масиву організувати окрему функцію.

Текст програми:




```
// Функція обчислення середнього
// арифметичного додатних елементів масиву
double sr_dod(double a[], int n)
{ double s=0;  int i,k=0;
  for (i=0; i<n; i++)
    if (a[i]>0) {s+=a[i]; k++;}
  return s/k;
}
//Кнопка "Розв'язок"
void __fastcall TForm1::Button1Click
                    (TObject *Sender)
{ // N – кількість усіх чисел у  Memo1
  int N=Memo1->Lines->Count;
  int i, k=0, j=0;
  double x;
  for (i=0; i<N; i++)
  { x=StrToFloat(Memo1->Lines->Strings[i]);
    if(fabs(x)<10) k++;       // k – кількість чисел, які за модулем є менше за 10
  }
  Edit1->Text=IntToStr(k);
  double *a = new double [k];  // Оголошення динамічного масиву
  // Введення елементів динамічного масиву з Memo1
  for (i=0; i<N; i++)
  { x=StrToFloat(Memo1->Lines->Strings[i]);
    if (fabs(x)<10) { a[j]=x; j++; }
  }
  StringGrid1->RowCount=k; // Встановлення кількості рядків у StringGrid1
  for (i=0; i<k; i++)       // Виведення створеного масиву
    StringGrid1->Cells[0][i]=FormatFloat("0.0",a[i]);
  double sum=sr_dod(a, k); // Обчислення суми додатних елементів масиву
  Edit2->Text=FormatFloat("0.0", sum);
  delete []a;               // Звільнення пам'яті від масиву
}
```

Приклад 6.7 Надати можливість уведення цілого числа (від 1 до 10) та динамічно створити і розмістити на формі відповідну кількість компонентів для введення елементів динамічно створюваного масиву і подальшого обчислення суми додатних елементів цього масиву.

Розв'язок. Як приклад динамічного створювання та розміщування компонентів на формі розглянемо створення певної кількості `Edit`-ів та `Label`-ів. Причому наведемо два різні підходи для роботи зі створеними компонентами як з масивом і як з окремими об'єктами. Аналогічним чином можна працювати з будь-якими компонентами.

Окрім зазначених алгоритмів динамічного створювання компонентів у цій програмі застосовано діалогове вікно `InputBox()` для введення цілого числа, синтаксис функції формування якого є такий:

```
AnsiString InputBox(AnsiString Caption, AnsiString Prompt,
                    AnsiString Default);
```

де `Caption` – текст надпису вікна;

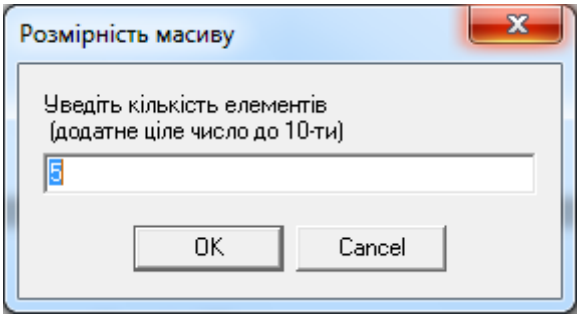
`Prompt` – пояснювальний текст (підказка) у вікні;

`Default` – пропоноване значення для введення, з яким можна погодитись або змінити його на нове значення.

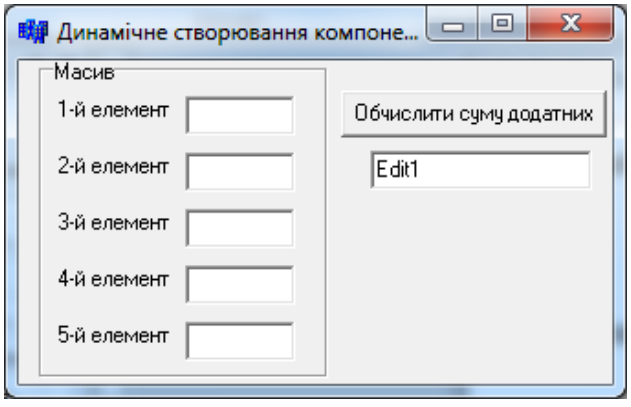
Вікно введення, створене у програмі командою

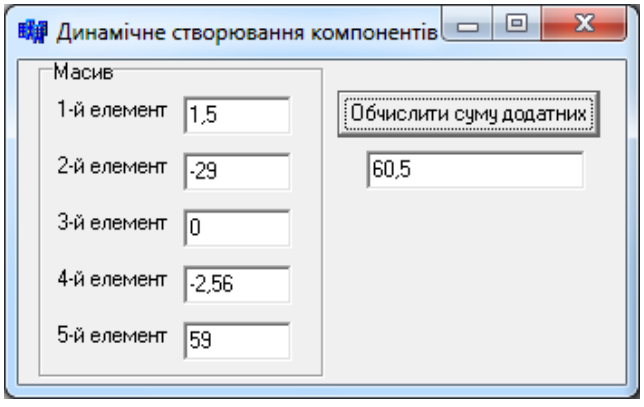
```
int n=StrToInt(InputBox("Розмірність масиву", "Уведіть кількість
                  елементів \n (додатне ціле число до 10-ти)", "5"));
```

матиме вигляд:



Приклад вигляду форми з відповідною кількістю динамічно створених компонентів на ній наведено ліворуч, а праворуч від нього показано вигляд цієї форми після введення довільних значень елементів динамічно створеного масиву та обчислення суми додатних його елементів.



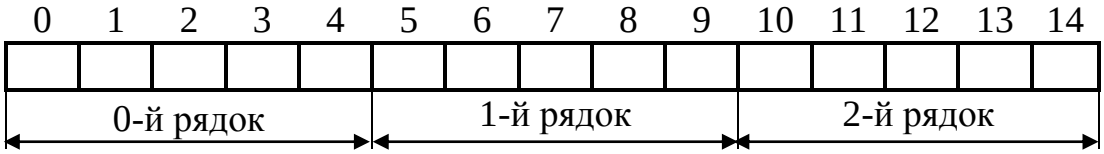


Текст програми

```
//---------------------------------------------------------------------
int n;                      // Кількість елементів масиву
TEdit *ed[10];              // Масив десяти вказівників на клас TEdit
TLabel *L;                  // L – вказівник на клас TLabel
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{ ww:
   n=StrToInt(InputBox("Розмірність масиву", "Уведіть кількість
                елементів \n (додатне ціле число до 10-ти)", "5"));
   if(n>10 || n<=0)
    { ShowMessage("Уведено некоректне значення!");   goto ww; }
   for(int i=0;i<n;i++)
    { // Створити новий компонент класу TEdit на формі як елемент масиву ed
      ed[i] = new TEdit(Form1);
      L = new TLabel(Form1); // Створення компонента класу TLabel на формі
      ed[i]->Parent=GroupBox1;// Для розміщення створених компонентів
      L->Parent=GroupBox1;    // задається їхній предок – компонент GroupBox1
      L->Caption=IntToStr(i+1)+"-й елемент";  // Надпис мітки
      // Розміщення відносно верхнього краю форми
      L->Top=i*30+20; ed[i]->Top=i*30+20;
      // Розміщення відносно лівого краю форми
      L->Left=10;     ed[i]->Left=80;
      ed[i]->Width=80;        // Ширина компонента
      ed[i]->Clear();         // Очищення вікна
    }
   // Відповідно до кількості створених компонентів встановлення висоти
   GroupBox1->Height=ed[n-1]->Top+30;  // GroupBox1
   Form1->Height=GroupBox1->Height+40; // та форми
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ double S=0, *a = new double [n];
  for (int i=0; i<n; i++)
  { a[i]=StrToFloat(ed[i]->Text);
    if (a[i]>0)  S+=a[i];
  }
  Edit1->Text=FloatToStr(S);
  delete []a;
}
//---------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{ delete []ed;
  delete L;
}
```

6.7 Динамічні двовимірні масиви (матриці)

Двовимірний динамічний масив з `m` рядків і `n` стовпчиків займає в пам'яті сусідні `m*n` комірок, тобто зберігається так само, як і одновимірний масив з `m*n` елементів. При розміщенні елементи двовимірних масивів розташовуються в пам'яті підряд один за одним з першого до останнього без проміжків у послідовно зростаючих адресах пам'яті. Наприклад, дійсний масив 3×5 зберігається у пам'яті в такий спосіб:



У такому масиві перші п'ять елементів належать до першого рядка, наступні п'ять – до другого і останні п'ять – до третього.

Нагадаємо, що `a` – вказівник на початок масиву, тобто на елемент `a[0][0]`. Щоб звернутися, наприклад, до елемента `a[1][3]`, слід "перестрибнути" від початку масиву через 5 елементів нульового рядка й 3 елементи першого рядка, тобто написати: `*(a+1*5+3)`. У загальному випадку до елемента `a[i][j]` можна звернутися в такий спосіб: `*(a+i*5+j)`. Але цей спосіб роботи з двовимірним масивом є не надто зручний, тому що в програмі при звертанні до елемента масиву доводиться розадресовувати вказівник і обчислювати індекс елемента.

Оголосити дійсний динамічний масив 3×5 можна як одновимірний з 15-ти елементів:

```
float *a=new float [3*5];
```

чи то

```
float *a=(float *)calloc(3*5, sizeof(float));
```

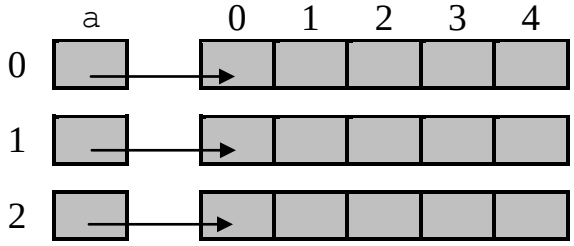
Пам'ять від створеного в такий спосіб масиву очищується за допомогою операцій відповідно **`delete`** й **`free`**:

```
delete []a;
```

чи то

```
free(a);
```

Розглянемо інший спосіб роботи з динамічним двовимірним масивом. Для цього розмістимо в пам'яті машини матрицю 3×5:



При цьому буде виділено пам'ять під кожний рядок матриці окремо, тобто буде утворено три різні одновимірні масиви. Адреси нульових елементів цих масивів зберігатимуться в допоміжному масиві `a` (пам'ять під нього слід виділити

заздалегідь). Елементами цього масиву будуть адреси дійсних чисел, тому вони матимуть тип "вказівник на дійсне число", тобто **float***. Нагадаємо, що загальний вигляд оголошення вказівника на динамічний масив є такий:

<тип елемента> * *<ім'я масиву>*;

Тоді при оголошенні масиву a слід записати дві зірочки:

```
float ** a = new float* [3];
// Оголошення й розміщення в пам'яті допоміжного масиву з 3-х елементів типу float*
for(int i=0; i<3; i++) a[i] = new float [5];
// У циклі виділяється пам'ять під 3 масиви по 5 елементів (рядки матриці) й адреси
// нульових елементів цих масивів записуються у відповідні елементи масиву a
```

Після цього можна працювати з матрицею як зі звичайним двовимірним масивом, звертаючись до кожного елемента за його індексом, наведеним у квадратних дужках: a[i][j], – що є більш природним і зручним, аніж попередній спосіб.

Аналогічне є оголошення за допомогою calloc():

```
float ** a = (float**) calloc (3, sizeof(float*));
for (int i=0; i<3; i++)
  a[i] = (float*) calloc (5, sizeof(float));
```

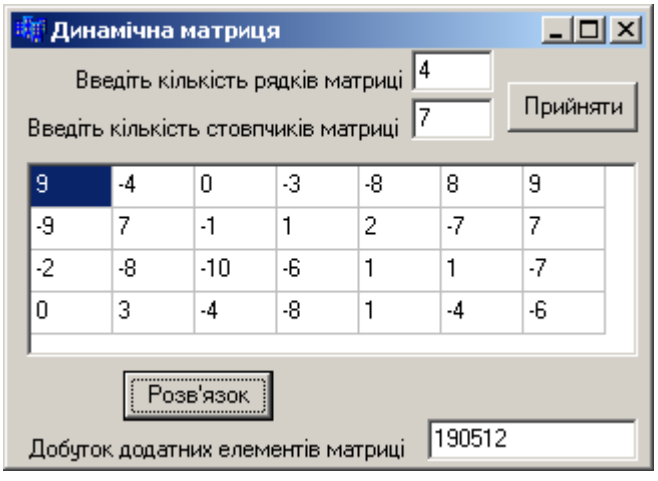
Приклад 6.8 Увести кількість рядків і стовпчиків матриці та елементи самої матриці. Обчислити добуток її додатних елементів.

Розв'язок. Оскільки кількість рядків та стовпчиків матриці заздалегідь є невідомі, матрицю доцільно організувати динамічно.

Наведемо два способи розв'язування.

Перший спосіб

Пам'ять виділяється під кожний рядок матриці окремо. При роботі з елементами матриці явно зазначаються обидва індекси (як при роботі з елементами нединамічної матриці), що вважається більш зручним.



Текст програми:

```
int m, n;
// Кнопка "Прийняти"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ m = StrToInt(Edit1->Text); // Введення розмірності матриці
  n = StrToInt(Edit2->Text);
  StringGrid1->RowCount = m;  StringGrid1->ColCount = n;
}
//Кнопка "Розв'язок"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ double p=1;
```

```
  float **a = new float* [m]; // Виділення пам'яті під допоміжний масив
  // Виділення пам'яті під кожний рядок матриці окремо
  for(int i=0; i<m; i++)  a[i]=new float[n];
  for(int i=0; i<m; i++)
  for(int j=0; j<n; j++)
  { a[i][j]=StrToFloat(StringGrid1->Cells[j][i]);
    if(a[i][j]>0) p*=a[i][j];  // Обчислення добутку додатних елементів
  }
  Edit3->Text=FloatToStr(p);
  for(int i=0; i<m; i++)
   delete []a[i];          // Очищення пам'яті від рядків матриці
  delete []a;              // Очищення пам'яті від допоміжного масиву
}
```

Другий спосіб

Змінюється лише `Button2Click` "Розв'язок". Пам'ять виділяється одразу під увесь двовимірний масив, звертання до елементів матриці виконується за допомогою одного індексу, який обчислюється залежно від індексів рядка та стовпчика. Звертатимемося до елементів масиву за допомогою вказівника, тобто для доступу до елемента матриці використовуватимемо операцію розадресації вказівника.

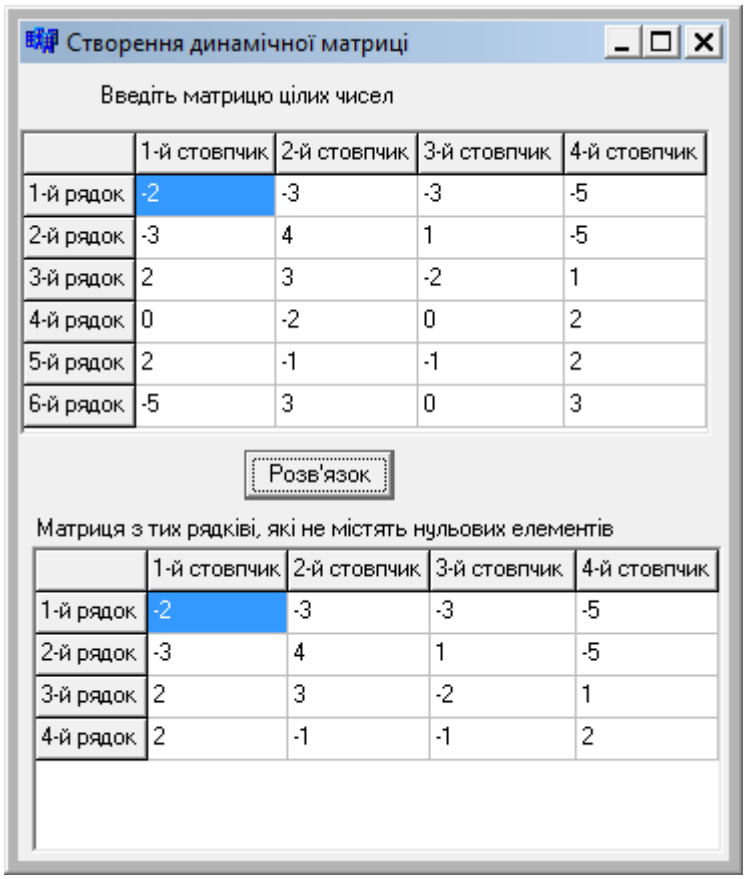
Текст програми:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ double p=1;
  float *a = new float [m*n]; // Виділення пам'яті під увесь масив одразу;
  // інакше це можна записати так: float *a = (float*)calloc(m*n, sizeof(float));
  for(int i=0; i<m; i++)
  for(int j=0; j<n; j++)
  { *(a+i*n+j)=StrToFloat(StringGrid1->Cells[j][i]);
    //При звертанні до елементів використовується вказівник і обчислюється індекс
    if(*(a+i*n+j)>0)  p *= *(a+i*n+j);
    // За допомогою операції індексації це можна записати так:
    // if (a[i*n+j]>0) p *= a[i*n+j];
  }
  Edit3->Text=FloatToStr(p);
  free(a);
}
```

Приклад 6.9 Увести цілочисельну матрицю 6×4 і створити нову матрицю з тих рядків уведеної матриці, які не містять нульових елементів.

Розв'язок. На відміну від першої матриці, друга матриця буде динамічною, оскільки кількість її рядків заздалегідь є невідома. Для обчислення кількості нульових елементів кожного рядка організуємо тимчасовий допоміжний масив `r`. Кількість рядків другої матриці визначатимемо як кількість ненульових елементів масиву `r` і у нову матрицю будемо копіювати лише ті рядки першої, для яких `r`=0.

Робочий вигляд форми після введення матриці та натиснення кнопки “Розв’язок”:



Текст програми

```
// Формування “шапки” SG1 індексами рядків і стовпчиків матриці
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{for(int i=1;i<=6;i++)SG1->Cells[0][i]=IntToStr(i)+" -й рядок";
 for(int j=1;j<=4;j++)SG1->Cells[j][0]=IntToStr(j)+" -й стовпчик";
}
//---------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, j, rowc=0,k=0;   double a[6][4], s=0;
  int r[6]={0};   // r – масив кількості нульових елементів рядків матриці
  for(i=0; i<6; i++)
  for(j=0; j<4; j++)
   { a[i][j]=StrToFloat(SG1->Cells[j+1][i+1]);
     if(a[i][j]==0)  r[i]++;
   }
  for(i=0; i<6; i++)   // Обчислення кількості рядків без нульових елементів
   if(r[i]==0) rowc++;     // r[i]=0 означає, що в і-тому рядку немає нулів
     SG2->RowCount=rowc+1;
  double **b=new double *[rowc];   // Оголошення матриці b
  for(i=0; i<rowc; i++) b[i]=new double [4];
  for(i=0; i<6; i++)
   if(r[i]==0)              // Якщо в і-тому рядку нулів немає,
```

```
      { for(j=0; j<4; j++) // відбувається
          b[k][j]=a[i][j]; // копіювання рядка у нову матрицю
        k++;               // і збільшення індексу рядка нової матриці.
      }
    for(i=0; i<rowc; i++)
    for(j=0; j<4; j++)  SG2->Cells[j+1][i+1]=FloatToStr(b[i][j]);
    for(int i=0; i<rowc; i++)
      delete [] b[i];       // Звільнення пам'яті від динамічної матриці
    delete []b;
    for(i=1;i<=rowc;i++) SG2->Cells[0][i]=IntToStr(i)+"-й рядок";
    for(j=1;j<=4;j++) SG2->Cells[j][0]=IntToStr(j)+"-й стовпчик";
  }
```

Питання та завдання для самоконтролю

1) Що являє собою пам'ять комп'ютера?

2) Що таке адрес комірки?

3) Що таке вказівник?

4) Наведіть синтаксис оголошення вказівника?

5) Яке тлумачення має значення вказівника NULL?

6) В який спосіб можна отримати адресу змінної?

7) Яке призначення операції `&`?

8) В який спосіб можна дізнатися значення, на яке посилається вказівник?

9) Поясніть відмінності поміж змінними `a` та `b`, оголошення яких має вигляд:

а) `int a; int b;` б) `int *a; float *b;`

10) Як можна виділити пам'ять під цілу змінну?

11) В який спосіб можна звільнити пам'ять, виділену за допомогою оператора `new`?

12) Як можна ініціалізувати пам'ять, виділену за допомогою оператора `new`?

13) В який спосіб можна звернутися до даних, для яких пам'ять було виділено динамічно?

14) Поясніть відмінності динамічного масиву від звичайного.

15) Оберіть правильні оголошення динамічного масиву з 5-ти цілих чисел:

a) `int a[5];`
b) `int *a[5];`
c) `int *a=malloc(5);`
d) `int *a=(int*) malloc(20);`
e) `int *a=(int*) malloc(5*sizeof(int));`
f) `int *a=new int [5];`
g) `int *a=new [5];`
h) `int *a=new int (5);`

16) Припустімо, у програмі оголошено масив: `int *a=new int[7];` Оберіть правильні команди і поясніть, що вони виконують:

a) `a=5;`
b) `a++;`
c) `a[3]++;`
d) `*a=1;`
e) `*(a+4)=3;`
f) `*a[3]=2;`

17) Оберіть правильні команди звільнення пам'яті від динамічного масиву `a` з 5-ти елементів: 14)

a) `delete a[5];`
b) `delete a[];`
c) `delete []a;`
d) `free (a);`
e) `free a[5];`

18) Припустімо, у програмі є наступні оголошення:

```
int *a; float *b; int *c(10); int d;
```

Які з наведених операцій присвоювання є помилковими і чому:

a) `a=5;` d) `*b=*c;` g) `a=d;`
b) `b=a;` e) `*a=3;` h) `a=&d;`
c) `c=a;` f) `*c=a;` i) `b=&d;`

19) Розглянемо масив `a`:

–5	30	8	–1	4

Яке значення має кожен з наступних виразів:

a) `*a` b) `*(a+4)`

20) Припустімо, у програмі оголошено масив: `int *a=new int[4];`
Оберіть команди, які призведуть до помилки, і поясніть:

a) `a=5;` c) `a[3]++;` e) `*(a+4)=3;`
b) `a++;` d) `*a=1;` f) `*a[3]=2;`

21) Як розташовуються у пам'яті елементи матриць?

22) Наведіть усі способи виділення пам'яті під динамічну матрицю?

23) Поясніть використання двох зірочок поряд у команді:

```
int **d = new int* [15];
```

24) У програмі оголошено матрицю 3×6: `int *W=new int [3*6];`
Запишіть звернення до елемента з наведеними значеннями індексів `i` та `j`:

a) `i=2, j=4` b) `i=0, j=3` c) `i=1, j=5`

25) У програмі двомірний масив 4×5 оголошено так:

```
double *a=new int [4*5];
```

Запишіть значення індексів рядка та стовпчика при таких звертаннях до елементів цієї матриці:

a) `*a` b) `*(a+2)` c) `*(a+15)`

26) У програмі двомірний масив 4×5 оголошено так:

```
float **A = new float* [4];
for(int i=0; i<4; i++)  A[i]=new float[5];
```

Запишіть індекси `i` та `j` елементів, звертання до яких у програмі має вигляд:

a) `*(A[1]+2)` b) `A[1][2]` c) `*(*(A+1)+2)`

27) Розтлумачте, чи є рівнозначними прототипи функцій:

```
float fun(float (*z)[5], int n);
float fun(float z[][5], int n);
```

28) У чому полягає відмінність роботи функцій `malloc()` та `calloc()`?

29) Яка функція застосовується для звільнення пам'яті, виділеної функцією `calloc`?

30) Яким є призначення функції `realloc()`?

Розділ 7

Символи і рядки

При виконуванні програм комп'ютер витрачає, за деякими оцінками, до 70 % часу на маніпулювання з текстовими рядками: копіює їх з одного місця пам'яті в інше, перевіряє наявність у рядках певних слів, поєднує чи відсікає рядки тощо. У C++ є два основні види рядків: С-рядки, які по суті є символьними масивами, і клас стандартної бібліотеки C++ `string`. Окрім того, C++ Builder надає свій доволі зручний в опрацюванні тип рядків `AnsiString`. Тому на початку розділу буде докладно описано особливості символьного типу `char`, потім буде розглянуто різновиди організації й опрацювання символьних масивів, які і є, власне, С-рядками, після чого буде наведено інформацію про `string`-рядки.

7.1 Символьний тип даних

Символами вважаються: великі й малі літери, цифри, знаки арифметичних дій (`'+'`, `'-'`, `'*'`, `'/'`, `'='`), пробіл, розділові знаки (`'.'`, `','`, `';'`, `':'`, `'!'`, `'?'`, `'-'`), службові символи, що відповідають клавішам <Enter>, <Esc>, <Tab> тощо. В C++ значення символьних констант записуються у одинарних лапках: `'3'`, `'f'`, `'+'`, `'%'`.

Як було зазначено у підрозд. 1.4, для кодування усіх символів використовується восьмирозрядна послідовність 0 і 1, тобто один байт. Наприклад: символ цифри `'9'` кодується послідовністю бітів 0011 1001, символ літери латиниці `'W'` – 0101 0111. За допомогою одного байта можна закодувати $2^8 = 256$ різних комбінацій бітів, а отже, 256 різних символів.

Щоб не було розходжень у кодуванні символів, існує єдиний міжнародний стандарт – так звана таблиця ASCII-кодів (American Standard Code for Information Interchange – американський стандартний код для обміну інформацією, див. додаток А). Символи ASCII мають коди від 0 до 127, тобто значення першої половини можливих значень байта, хоча часто кодами ASCII називають всю таблицю з 256 символів. Перші 128 ASCII-кодів є єдині для всіх країн, а коди від 128 до 255 називають розширеною частиною таблиці ASCII, де залежно від країни розташовується національний алфавіт і символи псевдографіки.

У таблиці ASCII всі символи пронумеровано, тобто вони мають власний унікальний код. Так само як у кожній мові людського спілкування існує алфавіт (перелік усіх літер у чітко визначеному порядку), усі комп'ютерні символи теж є суворо упорядкованими. Символ `' '` (пробіл) має код 32, цифри мають коди від 48 для `'0'` до 57 для `'9'`, великі латинські літери – від 65 для `'A'` до 90 для `'Z'`, малі літери латиниці – від 97 для `'a'` до 122 для `'z'`. Кодування другої половини таблиці ASCII має різні варіанти. Найпоширенішими є DOS-кодування (866 кодова сторінка) і кодування 1251, яке є основним для Windows (див. додаток А). Звернімо увагу на різницю поміж цифрами і їхнім символьним зображенням. Наприклад, символ цифри `'4'` має ASCII-код 52 і не має безпосереднього відношення до числа 4.

Керувальні символи таблиці ASCII не мають символьного подання, тобто не мають візуального зображення, тому їх інколи називають недрукованими (non-printed), наприклад: <Esc>, <Enter>, <Tab> тощо. Ці символи розташовано в перших 32-х кодах таблиці ASCII-кодів. Звертатися до таких символів можна через їхній код чи за допомогою так званої ескейп-послідовності (escape). Ескейп-послідовність – це спеціальна комбінація символів, яка розпочинається зі зворотної скісної риси і записується в одинарних лапках (табл. 7.1), наприклад: `'\0'`, `'\n'`. Кожна з наведених у таблиці комбінацій символів вважається за один символ.

Таблиця 7.1

Деякі поширені у застосуванні ескейп-послідовності

Символьне подання	Опис
`\n`	символ переведення курсора на початок наступного рядка
`\r`	переведення каретки
`\t`	символ переведення курсора на наступну позицію табуляції (відповідає клавіші <Tab>),
`\b`	символ вилучення попереднього символу перед курсором (відповідає клавіші <BackSpace>),
`\a`	символ звукового сигналу системного динаміка
`\\`	символ \ (зворотна скісна риса)
`\?`	символ ? (знак запитання)
`\'`	символ ' (одинарні лапки)
`\"`	символ " (подвійні лапки)
`\0`	символ з кодом 0 є завершальним символом рядка символів
`\0x`Вісімкове число	символ, код якого зазначено у вісімковій системі числення
`\0x`Шістнадцяткове число	символ, код якого зазначено у шістнадцятковій системі числення

Тип символьних змінних у C++ називається **`char`**. Так, при оголошенні

```
char c, s, g;
```

в оперативній пам’яті для кожної з цих трьох змінних буде відведено по одному байту. Коли символьні змінні набувають певних значень, то комп’ютер зберігатиме в пам’яті не власне символи, а їхні коди. Наприклад, замість літери `'A'` зберігатиметься її код 65. Тому, якщо присвоїти символьній змінній певне число, то C++ сприйме його як код символу з таблиці ASCII-кодів. Це поширюється лише на цілі числа. Зважаючи на різні кодування розширеної частини ASCII-таблиці для уникнення помилок вважається за оптимальне при роботі з символами використовувати їхнє символьне подання замість кодів. Наприклад, при оголошенні

```
char c = 115;
```

змінна `c` набуде значення символу `'s'`, який має код 115.

Зауважимо, що у C++, на відміну від більшості мов програмування, дані типу `char` змінюються у діапазоні –128 ... 127, причому додатні числа 0 ... 127

зайняті символами спільної частини ASCII-таблиці, а символи розширеної частини ASCII-таблиці у C++ відповідають від'ємним числам. Наприклад, літера кирилиці `'ч'` – має код –9, а кодом літери `'я'` є –1.

Окрім типу `char`, існує його беззнакова модифікація `unsigned char`. Дані типу `unsigned char` мають значення у діапазоні 0 ... 255. В ASCII-таблиці значення кодів літер кирилиці є більшими за 127, тому, якщо треба мати справу зі змінними, значеннями яких є літери кирилиці, їх слід оголошувати типом `unsigned char`:

```
unsigned char c;
```

Символи можна порівнювати. Більшим вважається той символ, у якого код є більший, тобто символ, розташований у таблиці ASCII-кодів пізніше. Наприклад: `'a'<'h', 'A'<'a'`.

Оскільки символьний тип `char` вважається у C++ за цілий тип, змінні цього типу можна додавати й віднімати. Результатом додавання буде символ, код якого дорівнює сумі кодів символів-доданків, наприклад:

```
char c = 'A';
char c1 = c + 5;    // c1 = 'F'
char c2 = c + 32;   // c2 = 'a'
char c3 = c - 10;   // c2 = '7'
```

У C++ існує ціла низка спеціальних функцій для роботи з символьними даними. Деякі з цих функцій наведено у табл. 7.2.

Таблиця 7.2

Функції для роботи з символами

Функція	Призначення
`tolower()`	повертає символ у нижньому регістрі
`toupper()`	повертає символ у верхньому регістрі
Належність символу до множини перевіряють такі функції:	
`isalnum()`	латинських літер та цифр (`'A' – 'Z'`, `'a' – 'z'`, `'0' – '9'`)
`isalpha()`	латинських літер (`'A' – 'Z'`, `'a' – 'z'`)
`iscntrl()`	керувальних символів (з кодами 0...31 та 127)
`isdigit()`	цифр (`'0' – '9'`)
`isgraph()`	видимих символів, тобто не є відповідним до клавіш <Esc>, <Tab> тощо
`islower()`	латинських літер нижнього регістру (`'a' – 'z'`)
`isprint()`	друкованих символів (`isgraph()` + пробіл)
`isupper()`	літер верхнього регістру (`'A' – 'Z'`)
`ispunct()`	знаків пунктуації
`isspace()`	символів-роздільників

Розглянемо деякі *поширені алгоритми опрацювання символьних змінних*.

1) Щоб визначити **код символу**, треба значення цього символу присвоїти цілій змінній. І, навпаки, щоб дізнатися, який символ відповідає певному числу, слід це число присвоїти символові. C++ сам виконає потрібні перетворення.

Наприклад, після виконання команд

```
int x;  char c = 'n';
```

```
x = c;
```

змінна x набуде значення 110, яке відповідає коду символу 'n' в ASCII-таблиці.

2) Визначити, чи є символ c **цифрою**, можна двома способами: перевірити його належність до символьного проміжку від '0' до '9' за допомогою умови:

```
if(c>='0' && c<='9') . . .
```

або застосувати функцію isdigit() для перевірки належності символу до множини цифр:

```
if(isdigit (c)) . . .
```

3) Дізнатися, чи є символ c **великою латинською літерою**, можна теж двома способами: перевірити його належність до символьного проміжку від 'A' до 'Z' за допомогою умови:

```
if(c>='A' && c<='Z') . . .
```

або застосувати функцію isupper() для перевірки належності символу до множини великих латинських літер:

```
if(isupper (c)) . . .
```

4) Перевірку, чи є символ c **латинською літерою**, можна здійснити за допомогою умови

```
if(c>='A' && c<='Z' || c>='a' && c<='z') . . .
```

або за допомогою функції isalpha():

```
if(isalpha (c)) . . .
```

5) Для перевірки, чи є символ c **малою латинською літерою**, слід записати умову

```
if(c>='a' && c<='z') . . .
```

або використати функцію islower()для перевірки належності символу до множини малих латинських літер:

```
if(islower(c)) . . .
```

6) Для перетворення малої латинської літери c **на велику** (верхній регістр) можна використати функцію toupper():

```
c = toupper(c);
```

чи то відняти від значення літери різницю кодів між відповідними великими і малими літерами (вона становить 32):

```
char c = c - 32;
```

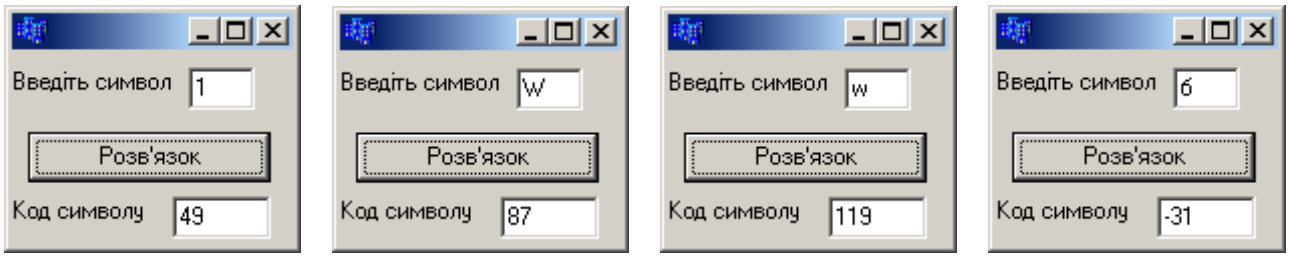
У аналогічний спосіб працює функція tolower(), яка збільшує код великих літер латиниці на 32 й, отже, здобуває **нижній регістр** літер латиниці.

Примітка. Вищезазначені функції перевірки й перетворення регістру працюють лише з латинськими літерами. Для виконання дій 1 – 5 з літерами кирилиці слід використовувати перевірку належності символу до відповідного символьного проміжку, а для перетворення регістра – зменшення або збільшення коду символу на різницю кодів між великими і малими літерами (для більшості символів вона теж становить 32).

Приклад 7.1 Увести один символ та визначити його код в таблиці ASCII.

Розв'язок. Для порівняння наведемо програмний код, реалізований у C++ Builder, та програму в консолі.

Вікно форми проекту з можливими результатами роботи програми:



Текст програми у Borland C++ Builder:

```
void __fastcall TForm1::Button1Click (TObject *Sender)
     { char c;              // Оголошення символьної змінної для зберігання символу
       int k;               // та цілої змінної для зберігання його коду
       c=Edit1->Text[1];    // Уведення значення символу
       k=c;                 // Обчислення коду
       Edit2->Text=IntToStr(k);
     }
```

Текст аналогічної програми у консолі:

```
#include <iostream.h>
     int main()
     { char c;   int k;
       cout<<"Введіть символ: ";
       cin>>c;              // Введення значення символу
       k=c;
       cout<<"Код введеного символу: "<<k<<endl;
       cin.get();
       return 0;
     }
```

7.2 Рядки

Символьні рядки можуть зберігати яку завгодно символьну інформацію, приміром: імена файлів, назви книг, імена працівників та інші символьні сполучення. C++ Builder підтримує кілька типів рядків – масиви символів, що перейшли від C (так звані C-рядки), клас `string` стандартної бібліотеки C++, класи `String` та `AnsiString` тощо. Останні є доволі зручні у застосуванні, однак на практиці нерідкі є ситуації, коли виникає потреба у користуванні вбудованим типом (C-рядками).

Майже всі різновиди рядків у C являють собою послідовність (масив) символів із завершальним нуль-символом. Нуль-символ (нуль-термінатор) – це символ з кодом 0, який записується у вигляді керувальної послідовності '\0'. За розташуванням нуль-символу визначається фактична довжина рядка.

Розпочнімо з розглядання символьних масивів.

7.2.1 Масиви символів

Відмінною рисою символьного масиву є те, що в ньому насправді може бути менше символів, аніж зазначено при оголошенні. Окрім того, з цими масивами можна виконувати певні специфічні дії, які не можна здійснювати з числовими масивами (наприклад перевіряти наявність у масиві літери чи послідовності літер, копіювати масив як одне ціле, порівнювати масиви за алфавітом, дописувати один масив наприкінці іншого тощо).

Пам'ять під розміщення рядків, як і для будь-яких масивів, може виділятися як компілятором, так і динамічно – при виконуванні програми. Довжина динамічного рядка може задаватися змінною з визначеним заздалегідь значенням, а довжина статичного рядка має задаватися лише константою.

Рядок може бути оголошеним в один з нижче наведених способів:

```
1) char *s;          // Оголошення вказівника на перший символ рядка;
                     // пам'ять під сам рядок не виділяється
2) char ss[15];      // Оголошення рядка ss з 14-ти символів;
                     // пам'ять виділяється компілятором
3) const int n = 10;
       char st[n];       // Оголошення рядка st з n-1 (тобто 9-ти) символів;
                         // пам'ять виділяється компілятором
4) int n = 10;
       char *str = new char[n];  // Оголошення рядка str з n-1 (тобто 9)
                                 // символів; пам'ять виділяється динамічно
```

При зазначенні довжини рядка слід враховувати завершальний нуль-символ. Наприклад, у вищенаведеному рядку `str` можна зберігати не 10, а лише 9 символів.

Зауважимо, що при оголошенні рядка першим способом пам'ять під рядок не виділяється і це може бути дуже небезпечним, оскільки до тієї самої ділянки пам'яті може бути розміщено інші змінні й рядок буде втрачено.

При оголошенні рядок можна ініціалізувати рядковою константою, при цьому нуль-символ формується автоматично після останнього символу:

```
char str[10] = "Vasia";
```

При цьому виділиться пам'ять під масив з 9-ти елементів та 10-й – нуль-символ (всього 10 байт) і перші 5 символів рядка записуються в перші 5 байт цієї пам'яті (`str[0]='V'`, `str[1]='a'`, `str[2]='s'`, `str[3]='i'`, `str[4]='a'`), а в шостий елемент `str[5]` записується нуль-символ. Якщо рядок при оголошенні ініціалізується, його розмірність можна опускати (компілятор сам виділить потрібну кількість байтів):

```
char str[] = "Vasia";      // Виділено й заповнено 6 байтів
```

Рядки у лапках завжди неявно містять нуль-символ, тому при ініціалізації прописувати його немає потреби. Окрім того, різні наведені способи введення символьних масивів автоматично долучають нуль-символ у кінець масиву.

При оголошенні й ініціалізації масиву слід бути впевненим, що розмір масиву є достатній, щоб умістити всі символи рядка з нуль-символом. Річ у тім,

що функції, які опрацьовують рядки, керуються позицією нуль-символу, а не розміром рядка. C++ не накладає жодних обмежень на довжину рядка.

Звернімо увагу на те, що рядкова константа (у подвійних лапках) і символьна константа (в одинарних лапках) не є взаємозамінними. Це константи різних типів. Символ у одинарних лапках, наприклад, `'s'` є *символьною* константою. Для зберігання такої константи компілятор C++ виділяє лише один байт пам'яті. Символ у подвійних лапках, наприклад, `"s"` є *рядковою* константою, що окрім символу `'s'` містить символ `'\0'`, який долучається компілятором. Більш того, `"s"` фактично являє собою адресу пам'яті, в якій зберігається рядок.

Як і числові масиви, символьні масиви опрацьовуються поелементно у циклі. Операція присвоювання одного рядка іншому є невизначена (оскільки рядок є масивом) і може виконуватися за допомогою циклу чи за допомогою функцій стандартної бібліотеки.

Для *введення* й *виведення* рядків у консолі використовуються функції **`scanf-printf`** і **`gets-puts`**, які вже розглядались у п. 2.2.1, наприклад:

```
const int n=10; char s[n];
gets(s);
puts(s);
```

Функція `gets(s)` зчитує символи з клавіатури до появи символу переведення рядка <Enter> і записує їх у рядок `s` (власне символ <Enter> до рядка не долучається, а замість нього записується нуль-символ). Функція `puts(s)` виводить рядок `s` на екран, замінюючи нуль-символ на <Enter>.

Окрім того, у консолі рядки можна вводити і виводити за допомогою **`cin`** і **`cout`**, як будь-які змінні, наприклад:

```
const int n=10;  char s[n];
cin>>s;
cout<<s;
```

Коли рядок виводиться за допомогою потоку `cout`, символи рядка виводяться по одному, допоки не зустрінеться завершальний символ `'\0'`.

При введенні рядків у консолі замість оператора >> більш оптимально використовувати метод **`getline`**`()`, оскільки потоковий оператор введення >> ігнорує пробіли. Окрім того, він може продовжувати введення елементів за межами масиву, якщо в пам'яті під рядок виділено менше місця, аніж вводиться символів. Функція `getline()` має два параметри: перший аргумент – рядок, який вводиться, а другий – кількість символів.

Наприклад:

```
char s[4];
cout<<"Введіть рядок: "<<endl;
cin.getline(s, 4);
```

Рядок `s` у цьому фрагменті програми може прийняти лише три значущих символи і буде завершений нуль-термінатором (символом `'\0'`). Решту введених символів рядка буде проігноровано.

Поширеним засобом доступу до символів рядка є вказівники типу `char*`.

У прикладі

```
char *st = "Комп'ютерна програма";
```

компілятор записує всі символи рядка до масиву і присвоює змінній `st` адресу першого елемента масиву.

Рядок може вважатися за порожній у двох випадках: якщо вказівник на рядок має нульове значення `NULL` (немає взагалі жодного рядка) чи коли вказівник вказує на масив, який складається з одного нульового символу (не містить жодного значущого символу).

```
char *pc1 = 0;          // pc1 не адресує жодного масиву символів,
const char *pc2 = "";   // pc2 адресує нульовий символ
```

Функції стандартної бібліотеки для роботи з рядками

C++ має багату колекцію функцій опрацювання рядків із завершальним нулем. Якщо в рядку відсутній нуль-термінатор, опрацювання рядка може тривати скільки завгодно, допоки в пам'яті не зустрінеться `'\0'`. У якості аргументів до функцій переважно передаються вказівники на рядки. Якщо при виконуванні функції здійснюється перенесення символів рядка з місця-джерела до місця-призначення, для рядка-призначення слід завчасно зарезервувати місце в пам'яті. Копіювання рядків з використанням просто вказівника, а не адреси початку завчасно оголошеного масиву – одна з найпоширеніших помилок програмування, навіть у досвідчених програмістів. При виділенні місця для рядка-призначення слід виділяти місце і для нуль-термінатора.

У табл. 7.3 наведено функції стандартної бібліотеки для роботи з С-рядками. Деякі з цих функцій, наприклад `strlen()`, `strcpy()`, опрацьовують рядки, оголошені в будь-який з чотирьох раніш розглянутих способів. Але більшість функцій потребує, щоб рядок був оголошений як вказівник типу `char*` (див. способи 1 і 4 на початку п. 7.2.1). У консолі для використання цих функцій слід залучити до програми бібліотеку `<string.h>`. Перелік усіх функцій цієї та інших стандартних бібліотек C++ наведено в додатку В.

Таблиця 7.3

Функції стандартної бібліотеки <string.h>

Функція	Призначення	Формат
`strlen()`	повертає довжину рядка (без урахування символу завершення рядка)	`size_t` **`strlen`**`(char *s);`
`strcat()`	долучає `s2` до `s1` і, як результат, повертає `s1`	`char *`**`strcat`**`(char *s1, char *s2);`
`strncat()`	долучає до рядка `s1` перші `n` символів з рядка `s2`	`char *`**`strncat`**`(char *s1, char *s2, size_t n);`
`strlwr()`	перетворює всі латинські літери до нижнього регістру	`char *`**`strlwr`**`(char *s);`
`strupr()`	перетворює всі латинські літери до верхнього регістру	`char *`**`strupr`**`(char *s);`

Продовження табл. 7.3

Функція	Призначення	Формат
strcmp()	порівнює рядки і повертає нульове значення, якщо рядки є однакові (s1=s2), від'ємне (s1<s2) чи додатне (s1>s2). Порівняння відбувається посимвольно і в якості результату повертається різниця кодів перших неоднакових символів	int ***strcmp**(char *s1, char *s2);
strncmp()	на відміну від попередньої функції, порівнює лише перші n символів рядка s1 з n символами рядка s2	int ***strncmp** (char *s1, char *s2, size_t n);
stricmp()	порівнює два рядки, не розрізнюючи прописні й малі літери латиниці	int **stricmp** (char*s1, char *s2);
strnicmp()	порівнює лише перші n символів двох рядків, не розрізняючи великі й малі літери латиниці	int **strnicmp** (char *s1, char *s2, size_t maxlen);
strcpy()	копіює до s1 рядок s2, повертає s1, при цьому попереднє значення s1 втрачається	char ***strcpy** (char *s1, char *s2);
strncpy()	замінює перші n символів рядка s1 на перші n символів рядка s2	char ***strncpy** (char *s1, char *s2, size_t n);
strchr()	відшукує перше входження символу ch в рядок s і повертає вказівник на цей символ, тобто частину рядка s, розпочинаючи з символу ch і до кінця рядка. Якщо символу ch в рядку s немає, результат – NULL	char ***strchr**(char *s, int ch);
strrchr()	відшукує останнє входження символу в рядку і повертає частину рядка s, розпочинаючи з останнього входження символу ch і до кінця рядка. Якщо символу ch в рядку s немає, результат – NULL	char ***strrchr** (char *s, int c);
strstr()	відшукує підрядок s2 в рядку s1, повертає частину рядка s1, розпочинаючи з першого спільного символу для обох рядків і до кінця s1	char ***strstr**(char *s1, char *s2);
strspn()	повертає довжину початкового сегмента рядка s1, символи якого є в рядку s2	size_t **strcspn** (char *s1, char *s2);
strcspn()	повертає індекс першого символу в рядку s1, який є спільним для обох рядків (нумерація індексів символів розпочинається з нуля)	size_t **strcspn** (char *s1, char *s2);
strset()	замінює усі символи рядка s на символ c	char ***strset**(char *s, char c);

Закінчення табл. 7.3

Функція	Призначення	Формат
strnset()	замінює лише перші n символів рядка s на символ c	char ***strnset** (char *s, int ch, size_t n);
strpbrk()	відшукує місце першого входження у рядок s1 будь-якого символу рядка s2 і повертає частину рядка s1, розпочинаючи з цього символу і до кінця рядка	char ***strpbrk** (char *s1, const char *s2);
strrev()	реверс рядка s	char ***strrev** (char *s);
strtok()	повертає частину (лексему) рядка s1 від поточної позиції вказівника до розділового символу, зазначеного у рядку s2; зазвичай використовується для перетворювання рядка на послідовність лексем	char ***strtok**(char *s1, const char *s2);

Розглянемо деякі з наведених у табл. 7.3 функцій детальніше на прикладах, для чого попередньо оголосимо вказівники на рядки s1, s2 та s3 і надамо їм початкові значення.

```
int k,n; char *s1="На Дерибасівській",*s2=" гарна погода",*s3;
n=strlen(s1);        // Обчислюється довжина рядка s1; n дорівнюватиме 17
k=strlen(s2);        // Обчислюється довжина рядка s2; k=13
k=strcmp(s1,s2);     // k=173; різниця між ASCII-кодами перших неоднакових
                     // символів рядків s1 і s2  дорівнює 173, а оскільки 173>0,
                     // можна стверджувати, що s1>s2
strcpy(s3,s2);       // s3=" гарна погода" (рядку s3 присвоюється рядок s2)
strncpy(s3,s1,2);    // Копіюються два перші символи з рядка s1 до  s3
s3[2]='\0';          // Третім символом після символів"На" задається
                     // нуль-символ щоби обрізати рядок
strcat(s1,s2);       // s1="На Дерибасівській гарна погода"; до рядка s1
                     // долучається рядок s2
s3=strchr(s1,' '); // Пошук пробілів у рядку s1;  тепер s3=" Дерибасівській
                     //гарна погода"  – це рядок від першого пробілу до кінця
                     // рядка s1, тобто без першого слова
s3=strrchr(s1,' ');// Пошук останнього пробілу в рядку s1;
                     // тепер s3=" погода"  – це останнє слово рядка s1
n=strchr(s1,' ')-s1+1;// n=3 – індекс першого пробілу в рядку s1, обчислений
                     // як різниця вказівників
k=strcspn(s1," ")+1;// k=3 – індекс першого пробілу в рядку s1; оскільки
                  // нумерація індексів символів розпочинається з 0, слід додати 1
s3=strstr(s1,s2);    // s3 = "гарна погода"; пошук рядка s2 у рядку s1
s3=strrev(s2);       // s3="адогоп анраг" – відбувається реверс рядка
s3=strupr("C++ Builder"); //  s3="C++ BUILDER" – перетворення усіх
                     // латинських літер рядка до верхнього регістру
```

```
s3=strlwr(s3);       // s3="c++ builder" – зворотне перетворення усіх
                     // латинських літер рядка до нижнього регістру
s3=strtok(s1," "); // s3="На" – перше слово (лексема) рядка s1 до розділового
                     // знака, який зазначено символом (пробілом) у другому
                     // аргументі функції. Почергове здобуття усіх лексем
                     // рядка можна організовувати у циклі (див. приклад 7.5)
```

Отже, функції `strcpy()` та `strncpy()` призначено для копіювання рядка чи то його частини до іншого рядка. Функції `strchr()`, `strrchr()` та `strstr()` повертають вказівник на віднайдений символ чи підрядок.

Функція `strtok()` використовується для перетворювання рядка на послідовність лексем. *Лексема* – це послідовність символів, відокремлених символами-розділювачами (зазвичай пробілами чи знаками пунктуації). Наприклад, у рядку тексту кожне слово може розглядатися як лексема, а пробіли, що відокремлюють слова одне від одного, можна розглядати як розділювачі. Для того щоб розбити рядки на лексеми, треба організувати кілька викликів функції `strtok()` (за умови, що рядок вміщує понад одну лексему). Більш докладно розглянемо ці функції у наведених далі прикладах програм.

Приклади програм з масивами символів

Приклад 7.2 Увести рядок до 10-ти символів і вивести через пробіл коди усіх уведених символів.

Розв'язок. У програмі `N` – максимальна кількість символів для оголошення масиву, `t` – кількість введених символів. Функція `Edit1->GetTextLen` дозволяє дізнатися про кількість уведених до `Edit` символів. Згідно до завдання, кількість символів має не перевищувати 10-ти. Тому, якщо буде введено понад 10 символів, для обмеження кількості опрацьовуваних символів у програмі прописано команду:

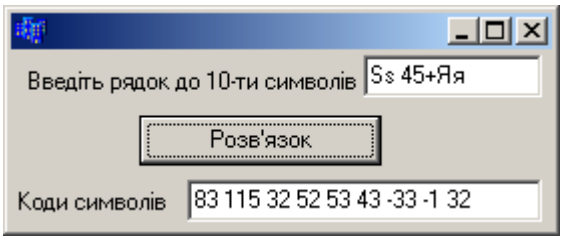
```
if(t>10) t=10;
```

Звернімо увагу на те, що нумерація елементів масиву розпочинається з 0, а нумерація символів в `Edit1->Text` – з 1. Тому при введенні індекс символу в `Edit1->Text` є на 1 більше за індекс елемента масиву.

Нагадаємо ще один важливий аспект. Як було зазначено у попередньому підрозділі (див. стор. 233), друга половина кодів таблиці ASCII (яка, зазвичай, включає кирилицю), з одиничним старшим бітом, інтерпретується як від'ємні значення, оскільки тип `char` за стандартом мови може мати знакову реалізацію.

Текст програми в Borland C++ Builder:

```
void __fastcall
TForm1::Button1Click(TObject
*Sender)
{ const int N=10;
  char c[N];
  int k, t=Edit1->GetTextLen();
  if (t>10) t=10;
  Edit2->Clear();
```

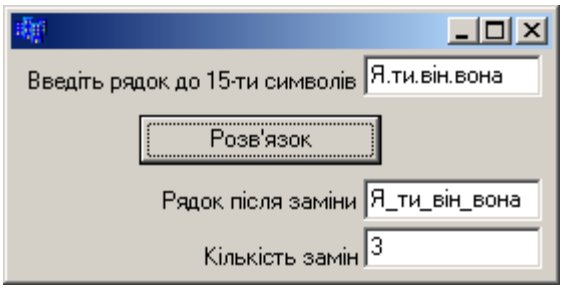


```
  for (int i=0; i<t; i++)
   { c[i]=Edit1->Text[i+1]; // Посимвольне введення рядка
     k=c[i];                // Обчислення коду символу та
     Edit2->Text=Edit2->Text+IntToStr(k)+" "; // його виведення
   }
}
```

Приклад 7.3 Увести рядок до 15 символів і замінити всі крапки на символ '_' та обчислити кількість замін.

Текст програми в Borland C++ Builder:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ const int N=15;   char c[N];
  int i,k=0,t=Edit1->GetTextLen();
  if (t>N) t=N;
  Edit2->Clear();
  for (i=0; i<t; i++)
   { c[i]=Edit1->Text[i+1];
      if(c[i]=='.')
        {c[i]='_'; k++;}
   }
  Edit3->Text=IntToStr(k);
  for (i=0; i<t; i++)  Edit2->Text=Edit2->Text+c[i];
}
```



Приклад 7.4 Створити рядок з усіх великих літер англійської абетки.

Розв'язок. Оскільки тип `char` є порядковим типом, то його можна застосовувати у якості типу параметра циклу. Після застосування операції інкремента (++) змінна типу `char` набуде значення коду наступного символу.

Текст програми в консолі:

```
#include <vcl.h>
#pragma hdrstop
#include <iostream.h>
//---------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{  char s[27];   // 26 латинських літер і '\0'
   char c;   int i=0;
   for(c = 'A'; c <= 'Z'; c++)
     s[i++] = c;
   s[i] = '\0'; // Записування символу '\0' до останнього символу рядка
   cout << "Англійська абетка: " << s;
   cin.get();
   return 0;
}
```

Результат виконання програми:

```
Англійська абетка: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Приклад 7.5 Увести рядок і вивести всі його слова окремо без розділових знаків.

Розв'язок. Наведемо три способи розв'язання цього завдання за допомогою функцій `strtok()` та `strncpy()`. Усі варіанти розв'язання є працездатними як у консолі, так і для C++ Builder.

Перший спосіб

Для розбивання рядка на слова (лексеми) застосовуватимемо функцію `strtok()`, яка має два аргументи: рядок, який треба розбити на лексеми, і рядок, який містить символи-розділювачі.

Перший виклик функції `strtok` у програмі

```
t = strtok(s, " .,;?!-");
```

присвоїть змінній `t` вказівник на перше слово рядка `s`. Другий аргумент наведеної функції містить можливі розділові символи. Функція `strtok()` відшукує перший символ у рядку `s`, який не є розділювачем. Це є початок першого слова. Потім функція віднаходить розділювач у рядку і замінює його на нуль-символ (`'\0'`). Цим завершується поточне слово. Функція `strtok()` зберігає вказівник на наступний символ рядка `s` за даним словом і повертає вказівник на поточну лексему (слово). У наступних викликах `strtok()` у програмі для почергового виокремлення слів з рядка `s` першим аргументом функції зазначається `NULL`, для того, щоб виокремлення наступного слова рядка `s` відбувалося з позиції, яку було збережено попереднім викликом `strtok()`. Якщо лексем при виклику `strtok()` більше немає, `strtok()` повертає `NULL` – і відбувається вихід із циклу.

Зверніть увагу, що `strtok()` модифікує вхідний рядок; тому, якщо рядок після виклику `strtok()` знову використовуватиметься у програмі, треба завчасно зробити копію рядка.

Цей алгоритм можна застосовувати для виокремлення чисел, але тоді у якості розділових знаків не слід застосовувати крапку, яка може інтерпретуватися як десяткова крапка.

Текст програми у консолі:

```
#include <iostream.h>
     #include <string.h>
     int main ()
     {char s[80], *t;
      puts(" Введіть рядок, який треба розбити на лексеми:");
      gets(s);
      puts("\n Лексеми:  ");
      t=strtok(s, " .,;?!-");
      while (t != NULL)
       { puts(t);
         t = strtok(NULL, " .,;?!-");
       }
      cin.get();   return 0;
     }
```

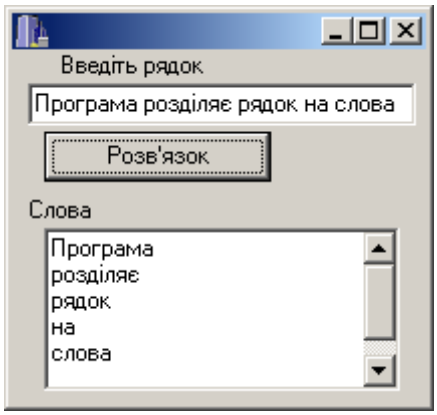
Результат виконання програми:

```
Введіть рядок, який треба розбити на лексеми:
    Hi, Tony! How are you? – I'm OK, thanks.
Лексеми:
    Hi
    Tony
    How
    are
    you
    I'm
    OK
    thanks
```

Другий спосіб

Для виокремлення слів рядка вважатимемо за ознаку завершення слова пробіл. Оскільки після останнього слова рядка здебільшого пробіл є відсутній, тому долучаємо його до рядка `s` у заздалегідь зарезервоване місце.

Рухаючись рядком, почергово виокремлюємо частину рядка `s` у змінну `slovo`. Виокремлення слова можна реалізувати за допомогою функції `strtok()` чи то, як у нижченаведеному тексті програми, функції `strncpy()`. Ця функція копіює частину рядка `s`, розпочинаючи з поточної позиції, а кількість літер копіювання визначається позицією пробілу. Після цього лишилося долучити до слова завершальний нуль і вивести слово окремим рядком до компонента `Memo`.



Після опрацювання усього рядка, щоб звільнити пам'ять, яку біло виділено за допомогою `new`, треба перемістити вказівник на початок рядка: `s-=n`, де `n` – кількість літер у рядку.

Текст програми в C++ Builder:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{int i, n=Edit1->Text.Length();
 char *s=new char[n+2], *slovo=new char[n+1];
 strcpy(s, Edit1->Text.c_str());
 strcat(s, " ");   n=strlen(s);
 Memo1->Clear();
 while(strchr(s,' '))                    // Допоки у рядку є пробіли,
 { for(i=0; i<n+2 && s[i]!=' '; i++); //відшукується позиція пробілу і
   strncpy(slovo, s, i); //копіюються символи до пробілу у змінну slovo
   slovo[i]='\0';
   Memo1->Lines->Add(AnsiString(slovo));
   s+=i+1;                          // Перехід до наступного слова
 }
 s-=n;                              // Повернення на початок рядка
 delete[]s;  delete[]slovo;
}
```

Третій спосіб

Іноді при виокремленні слів рядка виникає потреба у зберіганні цих слів в однотипному масиві для їхнього подальшого опрацювання. Для виділення пам'яті для масиву `arr` вказівників на слова спочатку слід визначити кількість слів (`kol`). Зверніть увагу, що в програмі ознака завершення слова визначається як: `i`-й символ не є пробілом `(!isspace(s[i]))`, а `(i+1)`-й символ – пробіл `(isspace(s[i+1]))`.

Текст програми в C++ Builder:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{int n=Edit1->Text.Length(), i, j=0, kol=0;
 char *s=new char[n+2];
 char *slovo=new char[n+1];
 strcpy(s, Edit1->Text.c_str());
 strcat(s," ");
 n=strlen(s);
 for(i=0;i<n-1;i++) if(!isspace(s[i])&&isspace(s[i+1])) kol++;
 char * *arr=new char *[kol];
 Memo1->Clear();
 while(strchr(s,' '))
  {for(i=0; i<n+2 && s[i]!=' '; i++);
   strncpy(slovo, s, i);
   slovo[i]='\0';
   arr[j]=new char[strlen(slovo)+1];
   strcpy(arr[j], slovo);
   j++;
   s+=i+1;
  }
 s-=n;
 for(i=0; i<kol; i++) Memo1->Lines->Add(AnsiString(arr[i]));
 for(i=0; i<kol; i++) delete []arr[i];
 delete[]s;
 delete[]slovo;
 delete[]arr;
}
```

Приклад 7.6 Вивести індекси всіх входжень літери `'a'` до рядка.

Розв'язок. Для виконання завдання доцільно використати функцію `strchr()`, яка відшукує перше входження відповідного символу до рядка і повертає вказівник на цей символ, тобто частину рядка `s`, розпочинаючи з цього символу і до кінця рядка. Якщо символу, зазначеного другим аргументом функції, в рядку немає, результатом функції є `NULL`.

У разі відсутності символу `'a'` у рядку `s` передбачено логічну змінну `F`, яка ще за першої перевірки щодо наявності символу сигналізує про його можливу відсутність, після чого виводиться відповідне повідомлення і відбувається вихід з програми.

Текст програми у консолі:

```
#include <iostream.h>
     #include <string.h>
     int main ()
     { char s[80], *p;
       bool F=0;
       puts("Введіть рядок:");
       gets(s);
       printf("Відшукуємо літеру 'a' у рядку \"%s\"...\n",s);
       p=strchr(s,'a');
       if(p==NULL)
        { puts("not found!");
          return 0;
        }
       while(p!=0)
        { printf ("found at %d\n",p-s+1);
          p=strchr(p+1,'a');
        }
       cin.get();
       return 0;
     }
```

Результат виконання програми:

```
Введіть рядок:
I am happy!
Відшукуємо літеру 'a' у рядку "I am happy! "...
found at 3
found at 7

Введіть рядок:
I'm hungry!
Відшукуємо літеру 'a' у рядку "I'm hungry! "...
not found!
```

Приклад 7.7 Увести рядок і перевірити, чи є він символьним поданням цілого числа.

Розв'язок. Перевірку, чи є символ цифрою, можна здійснити чи то за допомогою функції `isdigit()`, чи за перевірки належності його до діапазону цифр.

Текст програми у консолі:

```
#include <stdio.h>
void main()
{ char st[20];
  printf("Введіть ціле число ");
  scanf("%s",&st);
  int i, n=strlen(st);
```

```
  // Перший символ може бути цифрою чи знаком '-'
  if((st[0]<'0'||st[0]>'9')&&st[0]!='-')
   { printf("Рядок не є цілим числом");
     cin.get();
     return 0;
   }
  for(i=1; i<n; i++)
   if(st[i]<'0' || st[i]>'9') // Решта символів має бути цифрами
    { printf("Рядок не є цілим числом");
      cin.get();
      return 0;
    }
  printf("Рядок є цілим числом.\n");
  cin.get();
  return 0;
}
```

Результати виконання програми:

```
Введіть ціле число 23.5
Введений рядок не є цілим числом.
Введіть ціле число -56
Введений рядок є цілим числом.
```

Приклад 7.8 Увести рядок і перевірити, чи є він дійсним числом.

Текст програми у консолі:

```
#include <stdio.h>
#include <conio.h>
void main()
{ char st[20];
  int err=0; // Спочатку вважаємо, що рядок не є дійсним числом
  printf("Введіть дійсне число ");
  scanf ("%s",&st);
  int i, k=0, n=strlen(st);
  if(!isdigit(st[0])&&st[0]!='-')err=1;//Перший символ – цифра чи '-'?
  for(i=1; i<n-1; i++)
  {if(!isdigit(st[i]) && st[i]!='.')//Якщо символ не є цифрою чи
   {err=1; break;}//десятковою крапкою, переривається перевірка решти
   if(isdigit(st[i-1])&&st[i]=='.' && isdigit(st[i+1]))
   {k++;     //Якщо символ є десятковою крапкою, а попередній і наступний
    if(k>1) // символи – цифри, перевіряється можливість повторів таких
    {err=1; break;}//ситуацій і, якщо '.' вже була, перевірка припиняється
   }
  }
  if(k==0)err=1;     // Десяткових крапок усередині цифр не було?
  if(err==1) printf("Число не є дійсним!");
  else printf("Число є дійсним!");
  cin.get();   return 0;
}
```

Приклад 7.9 Створити програму для введення рядка і можливості вилучення із заданої позиції певної кількості символів рядка. Значення кількості символів для вилучення й номера позиції ввести з клавіатури.

Розв'язок. Цей алгоритм є поширеним у застосуванні і розв'язувати його можна по-різному, приміром за допомогою допоміжного рядка і вибирання до нього символів початкового рядка без тих символів, які треба "вилучити", після чого слід перезаписати створений рядок до початкового. Наведений нижче алгоритм вилучення частини рядка є більш зручним, оскільки не потребує використання допоміжного рядка. Розглянемо цей алгоритм на прикладі рядка `s`:

```
char *s="Погода дуже тепла";
```

У розглянутому нижче прикладі ціла змінна `pos` є номером символу, з позиції якого слід розпочати вилучення, `kol` – кількість символів, що їх буде вилучено, `&s[pos]` – частина рядка `s`, розпочинаючи з символу з індексом `pos`, а `pos+kol` – індекс першого символу, який не буде вилучено. Якщо `pos=7` і `kol=5`, команда

```
strcpy(&s[pos], &s[pos+kol]);
```

переставить (скопіює) слово "тепла" з нуль-символом на місце слова "дуже" унаслідок присвоювання вказівників. При цьому перша частина рядка залишиться на місці. Отже, слово "дуже" з пробілом зникне (його буде вилучено), а рядок стане коротшим.

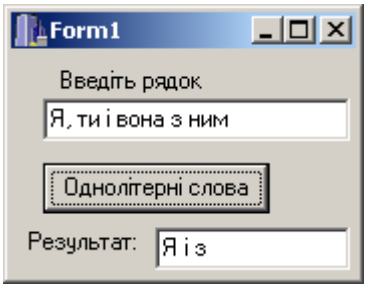
Текст програми у консолі:

```
#include <iostream.h>
void main()
{ int pos, kol;
  char *s=new char[80];
  gets(s);
  cout<<"pos = "; cin>>p;
  cout<<"kol= "; cin>>kol;
  pos --;  //Коригування номера, оскільки нумерація символів розпочинається з 0
  if(!(&s[pos+kol]==NULL || &s[pos]==NULL || pos<0 || kol<0))
     strcpy(&s[pos], &s[pos+kol]);
  puts(s);
  cin.get();
  return 0;
}
```

Приклад 7.10 Увести рядок і на його базі створити новий рядок з однолітерних слів.

Текст програми у C++ Builder:

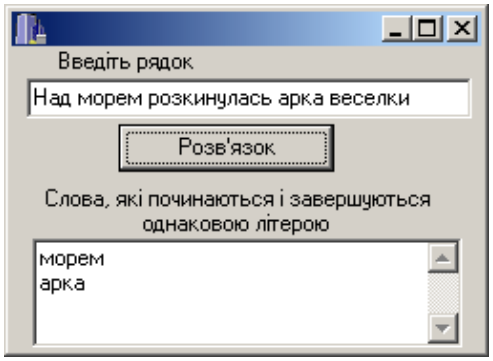
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{int N=Edit1->Text.Length();
 char *s=new char [N];
 char *s1=new char [N];
 AnsiString S=Edit1->Text;
 strcpy(s, S.c_str());
 strcpy(s1, "");
 char *D=" .,?!-";        // Розділювачі
 char *p=new char [N];
 p=strtok(s,D);           // Читання першого слова
 while(p!=0)              // Допоки не кінець рядка,
 {if(strlen(p)==1)        // перевіряється розмір слів і однолітерні слова
   {strcat(s1,p);         // долучаються до рядка s1
    strcat(s1," ");       // разом з пробілом
   }
  p=strtok(NULL,D);       // Виокремлення наступного слова
 }
 Edit2->Text=AnsiString(s1);
 delete []s;  delete []s1;  delete []p;
}
```



Приклад 7.11 Увести рядок і вивести всі слова, які розпочинаються і закінчуються на однакову літеру.

Розв'язок. Для виокремлення слів рядка вважатимемо ознакою кінця слова пробіл. Оскільки після останнього слова здебільшого пробіл є відсутній, тому долучаємо його до рядка `s` у заздалегідь зарезервоване місце.

Рухаючись рядком, почергово виокремлюємо слова і запам'ятовуємо у змінній `p` індекс першої літери (початку) поточного слова. Виокремлення слова можна реалізувати за допомогою функції `strtok()` або, як у наведеному нижче тексті програми, функції `strncpy()`. Ця функція копіює частину рядка `s` до рядка `slovo`, розпочинаючи з індексу `p`. Кількість літер, яку треба скопіювати, тобто довжину слова можна задати різницею індексів віднайденого пробілу і першого символу `i-p`. Після цього неодмінно слід долучити до слова завершальний нуль. Якщо у слові перша і остання літери збігаються, відбудеться виведення слова окремим рядком до компонента `Memo`.



Текст програми у C++ Builder:

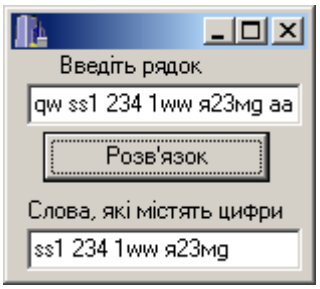
```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{int i, n=Edit1->Text.Length();
 char* s=new char[n+2];
 char* slovo=new char[n+1];
 strcpy(s, Edit1->Text.c_str());
```

```
 strcat(s," ");
 n=strlen(s);
 int p=0;
 Memo1->Clear();
 for (i=0; i<n+2; i++)
  {if(s[i]==' ')                    //Якщо віднайдено пробіл,
    { strncpy(slovo, &s[p], i-p);  // відбувається копіювання слова
      slovo[i-p]='\0';
      if(s[p]==s[i-1]) // Перша і остання літери слова збігаються?
        Memo1->Lines->Add(AnsiString(slovo));
      p=i+1;            // Запам'ятовування початку нового слова
  } }
 delete[]s;
 delete[]slovo;
}
```

Приклад 7.12 Увести рядок і на його базі створити новий рядок зі слів, які містять цифри.

Текст програми

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{int i, n=Edit1->Text.Length();
 char* s=new char[n+2];   // Виділення пам'яті для уведеного рядка s,
 char* sl=new char[n+1]; // слова sl
 char* rez=new char[n+2];// і результуючого рядка rez
 strcpy(s, Edit1->Text.c_str());
 strcat(s," ");
 n=strlen(s);
 strcpy(rez,"");
 int p=0;
 for(i=0; i<n+2; i++)
  {if (s[i]==' ')
    {strncpy(sl, &s[p], i-p);
     sl[i-p]='\0';
     for(int j=0; j<strlen(sl); j++)// Перебираються усі символи слова
      if(sl[j]>='0' && sl[j]<='9')    // і, якщо у ньому віднайдено цифру,
       { strcat(rez,sl);              // то це слово приєднається до rez
         strcat(rez," ");             // разом із пробілом
         break;           // Припинення подальшої перевірки цього слова
       }
     p=i+1;
    }
  }
 Edit2->Text=AnsiString(rez);
 delete[]s;
 delete[]sl;
 delete[]rez;
}
```



Створювання функцій опрацювання символьних масивів

Досить часто при опрацюванні рядків доводиться здійснювати набори подібних команд для реалізації однакових дій, наприклад, для введення і виведення рядків, вилучення частини рядка тощо. У таких ситуаціях, а також для спрощення програмного коду і полегшення сприйняття програми, ці набори команд організовують як функції. Передавання рядків до функції є подібним до передавання масиву в якості параметра, при цьому треба зазначити адресу початку масиву (див. п. 5.2.4). Зробити це можна трьома способами:

```
float fun (int *a);
float fun (int a[ ]);
float fun (int a[25]);
```

За першого способу передається вказівник як посилання на масив, явно визначений в основній програмі. Більш докладно роботу з вказівниками розглянуто в розд. 6. У другій синтаксичній формі константний вираз у квадратних дужках є відсутній. Така форма може бути використана лише тоді, коли масив ініціалізується чи оголошується як формальний параметр. За третього способу явно зазначено кількість елементів масиву, що накладає обмеження на довжину опрацьовуваного рядка і, як наслідок, обмежує застосування цієї функції.

В якості прикладу організуємо функцію `show_string()` для виведення рядка на екран, параметром якої є рядок для виведення:

```
#include <iostream.h>
     void show_string(char s[])
     { cout << s << endl;
     }
     int main()
     { show_string("Привіт, C++!");
       char *st="Я вчуся програмувати мовою C++");
       show_string(st);
       cin.get();
     }
```

Оскільки нуль-символ свідчить про кінець рядка, функція `show_string()` не потребує додаткового параметра, який би задавав кількість символів у рядку.

Наведемо ще один приклад функції, яка обчислюватиме довжину рядка. І хоча є подібна бібліотечна функція, для кращого розуміння організації рядків не зайвим буде створення власної функції для реалізації цього алгоритму. Нижченаведена функція `length()` для визначення кількості символів шукає символ `'\0'` у рядку. Тип функції, тобто її результату є `int`, а оператор `return` повертає у якості результату довжину рядка, яка обчислюється як номер останнього символу рядка `'\0'`.

```
#include <iostream.h>
int length(char *s)
{ int i;
  // У циклі перебираються усі символи, допоки не зустрінеться '\0'
  for(i=0; s[i] != '\0'; i++);
```

```
  return i; // Функція повертає довжину рядка, яка є номером нуль-символу
}
int main()
{ char s1[] = "Вчимося програмувати мовою С++";
  char *s2 = "Hello!";
  cout<<"Рядок "<<s1<<" містить "<<length(s1)<<" символів"<<endl;
  cout<<"Рядок "<<s2<<" містить "<<length(s2)<<" символів";
  cin.get();
}
```

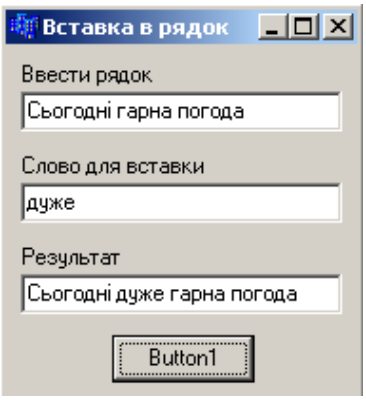
Приклад 7.13 Увести рядок і слово, яке слід вставити після першого слова рядка.

Розв'язок. Алгоритм, який уставляє до рядка `s` слово `slovo` на позицію `k`, реалізуємо у функції. Позицію для вставляння слова, яка є індексом символу наступного після першого пробілу рядка, задамо як різницю вказівників на пробіл і на початок рядка, збільшену на одиницю: `strchr(s,' ')-s+1`.

Функція `strcat()` долучає свій другий аргумент до першого аргументу, при цьому перший символ другого рядка записується замість нульового символу (`'\0'`), який завершував перший рядок. Програміст має бути впевненим, що перший рядок, є достатньо великий для того, щоб зберегти комбінацію першого рядка, другого рядка та нуль-символу, який копіюється з другого рядка.

Текст програми:

```
void str_insert(char *s, char *slovo, int k)
{ // Долучення до рядка slovo частини рядка s, розпочинаючи з позиції k
  strcat(slovo, &s[k]);
  // Копіювання в рядок s, розпочинаючи з позиції k, рядка slovo
  strcpy(&s[k], slovo);
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{int N=Edit1->GetTextLen();
 char *s = new char[2*N+1];
 strcpy(s,Edit1->Text.c_str());
 // Оголошення і введення слова
 char *slovo = new char[N+1];
 strcpy(slovo, Edit2->Text.c_str());
 // Долучення пробілу до слова
 strcat(slovo, " ");
 // Якщо в рядку s немає пробілу, він долучається
 if(!strchr(s,' '))strcat(s, " ");
 // Вставлення слова в рядок, розпочинаючи з індексу першого пробілу рядка
 str_insert(s, slovo, strchr(s,' ')-s+1);
 Edit3->Text=AnsiString(s);
}
```



Функції перетворювання рядка `char*` на число

Виконати перетворювання рядка на число можна багатьма способами. Перший – використання бібліотечних функцій `atoi()`, `atof()`, `atol()`. Ці функції входять до стандартної бібліотеки мови С і мають такий формат:

```
int atoi(const char* str);
long atol(const char* str);
double atof(const char* str);
```

Результатами перетворювання цих функцій є числа відповідного типу. Функції цілочисельного перетворювання `atoi()` і `atol()` сприймають такий формат перетворюваного рядка:

[*пробіли*][*знак*]*цифри*

а функція перетворювання рядка в дійсне число **`atof`**`()`, відповідно:

[*пробіли*][*знак*][*цифри*][.*цифри*][{**d** | **D** | **e** | **E** }[*знак*]*цифри*]

Тут пробіли – будь-який зі знаків пробілу, табуляції (`'\t'`), вертикальної табуляції (`'\v'`) – при перетворюванні ігноруються. Знак – символ `'+'` чи `'-'` (якщо його не зазначено, то вважається, що число є додатне). Цифри – символи від `'0'` до `'9'`. Для числа з рухомою крапкою, якщо немає цифр до символу `'.'`, має бути хоча б одна цифра після нього. Дійсне число може бути подано в експоненціальній формі з одним із символів-префіксів експоненти.

Основний недолік цих функцій полягає в тому, що вони ніяк не сигналізують про помилку, якщо така станеться в перебігу перетворювання рядка на число, наприклад через невідповідність його формату чи то з інших причин. Цю проблему розв'язують такі стандартні бібліотечні функції перетворювання рядків на числа:

```
long strtol(const char* str, char **endptr, int radix)
unsigned long strtoul(const char* str, char* *endptr, int radix)
double strtod(const char* str, char* *endptr)
```

Ці функції мають такі відмінності від попередньої групи:

✓ через параметр `endptr` вони повертають вказівник на перший символ, який не може бути інтерпретований як частина числа;

✓ відбувається контроль переповнення і, якщо це сталося, функції сигналізують встановленням змінної `errno` у значення `ERANGE`, а також повертають, відповідно, `LONG_MAX/LONG_MIN`, `ULONG_MAX/ULONG_MIN` та `+/-HUGE_VAL` залежно від знака числа у переданому рядку;

✓ функція `strtol()` використовує інформацію про поточні встановлені через `setlocale` регіональні налагодження, а, отже, може коректно інтерпретувати числа з символом `','` як розділювачем цілої та дробової частини;

✓ для функцій `strtol()` і `strtoul()` можна задавати параметром `radix` основу системи числення, при цьому, якщо `radix` дорівнює 0, основа визначається автоматично за першими символами числа.

Типове використання цих функцій може мати вигляд:

```
char* end_ptr;
long val = strtol(str, &end_ptr, 10);
```

```
if(*end_ptr)
 { . . . // Сигналізування про помилку в рядку
 }
if((val == LONG_MAX || val == LONG_MIN) && errno == ERANGE)
 { . . . // Сигналізування про переповнення
 }
// Продовження штатної роботи
```

Перетворення числа на рядок

Для перетворення цілого числа на рядок можна скористатися такими функціями:

```
char* itoa(int value, char* string, int radix);
char* ltoa(long value, char* string, int radix);
char* ultoa(unsigned long value, char* string, intradix);
```

Для цих функцій перший вхідний параметр `value` – ціле число, яке треба перетворити на рядок; другий параметр `string` є буфером, до якого буде записано результат перетворювання, а третій параметр `radix` – це основа системи числення, в якій буде подано число.

Приклад 7.14 Увести рядок – математичний вираз, що міститиме два числа і знак арифметичної дії поміж ними та обчислити значення цього виразу.

Текст програми в консолі:

```
#include <string.h>
#include <stdio.h>
#include <iostream.h>
int main()
{ char *s=new char [80];
 int n,i;
 double x,y,z;
 char c;
 gets(s);
 n=strlen(s);
 x=atof(s);
 for(i=1; i<n; i++)
  if(!isdigit(s[i])&&s[i]!='.')
    if(s[i]!='+' && s[i]!='-' && s[i]!='*' && s[i]!='/')
      { cout<<"Error"; cin.get();  return 0;}
    else { c=s[i];  y=atof(&s[i+1]);  break;}
 switch (c)
 {case '+': z=x+y; break;
  case '-': z=x-y; break;
  case '*': z=x*y; break;
  case '/': z=x/y; break;
 }
 cout << x << c << y << "=" << z << endl;
 cin.get();  return 0;
}
```

Результати роботи програми:

```
23,5+6
Error

23.5-7.6
23.5-7.6=15.9
```

7.2.2 Клас AnsiString (String)

AnsiString (String) – тип динамічного рядка, який може містити до $(2^{32}–1)$ символів. Цей тип рядків є зручним в опрацюванні, хоча і не є стандартним типом C++. Він з'явився у Borland C++ Builder з Delphi, а отже, його можна використовувати лише у C++ Builder. Особливістю цього типу є те, що два рядки можуть фізично займати одну й ту саму ділянку пам'яті. Рядок AnsiString містить лічильник посилань до нього і, коли цей лічильник набуває нульового значення, рядок автоматично знищується. На відміну від рядків типу char*, нумерація символів у рядку AnsiString розпочинається з 1.

Оголосити рядок AnsiString можна в такий спосіб:

```
AnsiString S;
```

Доволі зручним є введення та виведення рядків AnsiString за допомогою компонентів форми, оскільки відповідні текстові поля (властивості) компонентів мають цей тип. Наприклад, введення та виведення рядка S через компонент Edit1 можна подати в такий спосіб:

```
S=Edit1->Text;      // Введення рядка з Edit1
Edit2->Text=S;      // Виведення рядка до Edit2
```

Щоб визначити кількість символів рядка, використовується функція Length():

```
int N=S.Length();
```

Найбільш поширені у застосовуванні функції та методи опрацювання рядків AnsiString наведено у табл. 7.4.

Таблиця 7.4

Деякі функції та методи опрацювання рядків AnsiString

Назва	Призначення	Формат
Length()	повертає довжину рядка (без урахування символу завершення рядка)	`int` **`Length`**`();`
SetLength()	змінює довжину рядка на newLength, за потреби скорочуючи його	`AnsiString&` **`SetLength`** `(int newLength);`
Insert()	вставляє рядок str, розпочинаючи з індексу index	`AnsiString&` **`Insert`**`(const AnsiString &str, int index);`
Delete()	вилучає з рядка зазначену кількість символів count, розпочинаючи з індексу index	`AnsiString&` **`Delete`** `(int index, int count);`

Закінчення табл. 7.4

Назва	Призначення	Формат
`Pos()`	повертає номер індексу, з якого розпочинається підрядок `subStr`. Якщо рядок не містить `subStr`, функція повертає 0	`int Pos(const AnsiString &subStr);`
`Trim()`	вилучає початкові й кінцеві пробіли і повертає новий рядок без пробілів	`AnsiString Trim();`
`TrimLeft()`	вилучає початкові пробіли, розташовані ліворуч рядка	`AnsiString TrimLeft();`
`TrimRight()`	вилучає кінцеві пробіли, розташовані праворуч рядка	`AnsiString TrimRight();`
`SubString()`	повертає підрядок, який містить `count` символів, розпочинаючи з індексу `index`	`AnsiString SubString (int index, int count);`
`ToInt()`	перетворює рядок на ціле число	`int ToInt();`
`ToDouble()`	перетворює рядок на дійсне число	`double ToDouble();`
`LowerCase()`	перетворює символи рядка на малі, тобто на нижній регістр	`AnsiString LowerCase();`
`UpperCase()`	перетворює символи рядка на великі, тобто на верхній регістр	`AnsiString UpperCase() const;`
`c_str()`	перетворює рядок до типу `char*`	`char* c_str();`

За приклад використання деяких з наведених у таблиці функцій наведемо програмний код

```
AnsiString S = "Рядок символів";
int k=S.Pos(" ");       // Віднаходження позиції пробілу; k=6
if(k != 0)
{ S.Insert("чо",k-1);   // Тепер рядок має вигляд "Рядочок символів"
  S.Delete(k+4,4);      // Вилучення 4-х літер з позиції 10;
}                       // тепер рядок має вигляд "Рядочок слів"
S=S.UpperCase();        // S=" РЯДОЧОК СЛІВ"
S=S.SubString(2,1)+S.SubString(9,3);   // S="ЯСЛІ"
S.SetLength(3);         // S=" ЯСЛ"
char *s;
strcpy(s, S.c_str());   // Переведення рядка до типу char*
```

Рядки `AnsiString` можна переприсвоювати один одному. Також їх можна "склеювати" за допомогою операції конкатенації +:

```
AnsiString S1="Hello", S2="world", S3;
S3=S1+" "+S2;       // S3="Hello world"
S3=S1;              // S3="Hello"
```

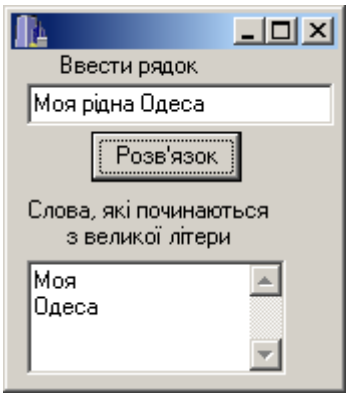
Рядки `AnsiString` можна порівнювати за допомогою операцій відношення (==, !=, <, <=, >, >=). Порівняння виконується згідно до ASCII-кодів символів рядка.

До символів рядка `AnsiString` може бути застосовано функції опрацювання символів, розглянуті в попередньому підрозд. 7.1. Наприклад, у поданому далі фрагменті програми рядок `S` зчитується з `Edit1`, а потім обчислюється кількість великих літер `kol` у цьому рядку:

```
AnsiString S=Edit1->Text;
int N=S.Length(), kol=0;
for(int i=1; i<=N; i++) if(isupper(S[i])) kol++;
```

Функції перетворення рядків `AnsiString` на числа розглянуто в підрозд. 3.6.

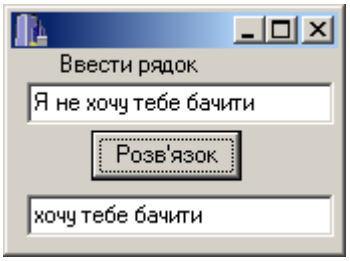
Приклади програм з рядками AnsiString



Приклад 7.15 Увести рядок і віднайти у ньому всі слова, які розпочинаються з великої літери.

Текст програми:

```
void __fastcall TForm2::Button1Click
                                (TObject *Sender)
{ Memo1->Clear();
 AnsiString S, S1;
 S=Edit1->Text;          // Введення рядка
 int p, n=S.Length(); // і обчислення його довжини
 // Долучення до рядка пробілу (ознакою завершення слова вважається пробіл)
 S=S+" ";
 p=1;       // p – позиція першого символу чергового слова, для першого слова  p=1
 for(int i=1; i<=n; i++)
  if(S[i]==' ') // Якщо символ є пробілом, це означає, що слово завершилось.
   { // Якщо перший символ цього слова є великою літерою латиниці чи кирилиці,
     if(isupper(S[p]) || (S[p]>='A' && S[p]<='Я'))
      { S1=S.SubString(p, i-p); // це слово копіюється до  S1
        Memo1->Lines->Add(S1); // і виводиться до Мето
      }
    // Тепер p – позиція початку нового слова (позиція пробілу плюс 1).
    p=i+1;
   }
}
```

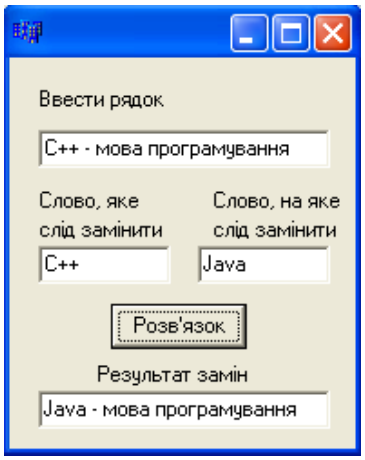


Приклад 7.16 Увести рядок і вилучити з нього всі слова, довжина яких є менше за три символи.

Текст програми

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString S=Edit1->Text; // Зчитування рядка
  S=S+" "; // Приєднання пробілу, щоб "не втратити" останнє слово рядка
  int p=1, i, dl; // p – перша літера слова
  // Оскільки довжина рядка буде змінюватися при вилучанні слів,
  // то не можна в умові виходу із циклу ставити її заздалегідь обчислене значення
  for(i=1; i<=S.Length(); i++)
   if(S[i]==' ') // Якщо символ є пробілом, це є ознакою завершення слова
    { dl=i-p;     // Обчислення довжини слова
```

```
        if(dl<3)     // Слова з довжиною меншою за 3
         { S.Delete(p, i-p+1);// вилучаються,
           i=p-1;   // а позиція переміщується на пробіл перед вилученим словом,
         }          // інакше (якщо слово не вилучено),
        else p=i+1;// p стає номером першої літери наступного слова
      }
  Edit2->Text=S;  // Виведення рядка після вилучення слів.
}
```



Приклад 7.17 Увести рядок та замінити у рядку всі входження одного введеного слова на інше.

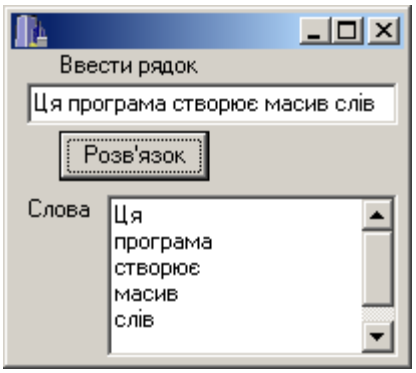
Текст програми

```
void __fastcall TForm1::Button1Click
                              (TObject *Sender)
{AnsiString S, sl1, sl2;
 S=Edit1->Text+" ";
 sl1=Edit2->Text;  sl2=Edit3->Text;
 int p=1, i=1, n=S.Length();
 while (S.Pos(sl1)>0)       // Допоки слово, яке треба замінити, є в рядку,
  {int k=S.Pos(sl1);              // визначається позиція цього слова,
   S.Delete(k, sl1.Length());     // воно вилучається і на його місце
   S.Insert(sl2,k);               // вставляється нове слово
  }
 Edit4->Text=S;                   // Виведення нового рядка
}
```

Приклад 7.18 Увести рядок і створити масив слів уведеного рядку.

Розв'язок. Подібне завдання для рядків `char*` наведено у третьому способі розв'язання прикладу 7.5, але корисним буде розглянути його розв'язання для рядків `AnsiString`.

Текст програми:




```
void __fastcall TForm1::Button1Click
                      (TObject *Sender)
{int i, j=0, kol=0, N, k;
 AnsiString S=Edit1->Text+" ";
 N=S.Length();
 Memo1->Clear();
 for(i=1; i<N; i++)
  if(!isspace(S[i])&&isspace(S[i+1]))
     kol++; // Обчислення кількості слів.
 AnsiString *arr=new AnsiString [kol];
 while(S.Pos(" ")>0)       // Допоки у рядку є слова,
  { S=S.TrimLeft();        // вилучаються зайві початкові пробіли,
    k=S.Pos(' ');          // обчислюється позиція пробілу,
    arr[j]=S.SubString(1, k-1); // копіюється слово в елемент масиву,
```

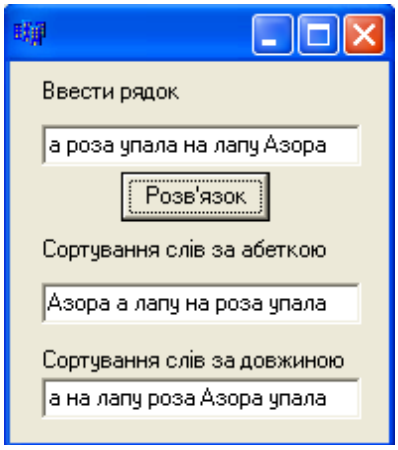
```
    j++;                    // збільшується індекс масиву
    S.Delete(1, k);         // і вилучається розглянуте слово з рядка.
   }
 for (i=0; i<kol; i++) Memo1->Lines->Add(arr[i]);
 delete []arr;  // Звільнення пам'яті, виділеної під масив вказівників на слова
}
```

Приклад 7.19 Увести рядок і відсортувати слова у рядку: а) за абеткою, б) за зростанням довжин слів.

Розв'язок. Оскільки завдання складається з двох явних частин, а також для більшої зрозумілості програмного коду організуємо розв'язання в окремих функціях.

Текст програми:

```
// Функція сортування масиву слів за абеткою
void sort_alf(AnsiString* arr, int kol)
{ bool f; int i;
  do
   {f=true;
    for(i=0; i<kol-1; i++)
     if(arr[i]>arr[i+1])
      { AnsiString temp=arr[i];
        arr[i]=arr[i+1];
        arr[i+1]=temp;
        f=false;
      }
   }while (f==false);
}
```

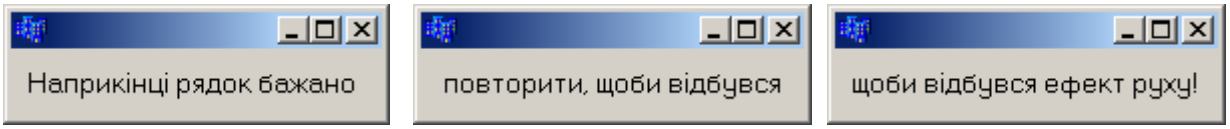



```
// Функція сортування масиву слів за зростанням довжин
void sort_len(AnsiString* arr, int kol)
{bool f; int i; AnsiString temp;
 do
  {f=true;
   for(i=0; i<kol-1; i++)
    if(arr[i].Length()>arr[i+1].Length())
     { temp=arr[i]; arr[i]=arr[i+1]; arr[i+1]=temp;
       f=false;
     }
  } while(f==false);
}

// Функція копіювання масиву слів у рядок
void copy_array(AnsiString &S, AnsiString *arr, int kol)
{ int i;   S="";
  for(i=0; i<kol; i++)
   { S=S+arr[i];
     if(i!=kol-1) S=S+" ";
}  }
```

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString S=Edit1->Text+" ";
 int i, j=0, kol=0, k, N=S.Length();
 for(i=1; i<N; i++)
  if(!isspace(S[i])&&isspace(S[i+1])) kol++;
 AnsiString *arr=new AnsiString [kol];
 while(S.Pos(" ")>0)
  { S=S.TrimLeft();        // Вилучення зайвих початкових пробілів
    k=S.Pos(' ');
    arr[j]=S.SubString(1, k-1);
    j++;
    S.Delete(1, k);
  }
 sort_alf(arr, kol);     // Виклик функції сортування масиву за абеткою
 copy_array(S,arr,kol); // Виклик функції копіювання масиву у рядок
 Edit2->Text=S;          // Виведення рядка після сортування за абеткою
 sort_len(arr,kol); // Виклик функції сортування слів за зростанням довжин
 copy_array(S,arr,kol);
 Edit3->Text=S;          // Виведення рядка після сортування
 delete []arr;
}
```

Приклад 7.20 Створити рядок, символи якого пересуваються, "біжать" екраном.

Розв'язок. Розмістимо на формі лише два компоненти: `Label1`, який показуватиме на формі певну змінювану частину рядка `GoString`, та `Timer1`, який задаватиме частоту змінювань символів. Ніякого попереднього встановлення значень властивостей цих компонентів наведена програма не потребує.



Текст програми:

```
int P=1;
// Функція виведення частини рядка викликається кожні 1/5 секунди
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{Timer1->Interval=200;     // Частота змінювань символів – 1/5 секунди
 // Рядок, який рухатиметься.
 AnsiString GoString="Наприкінці рядок бажано повторити,
            щоб відбувся ефект руху! Наприкінці рядок";
 int LengthGoString=25;                  // Ширина вікна виведення
 // Виведення частини рядка, розпочинаючи з позиції P.
 Label1->Caption=GoString.SubString(P, LengthGoString);
 P++;                                    // Перехід до наступного символу
 // Якщо решта рядка є коротша за ширину вікна виведення, перейти
 if(GoString.Length()-LengthGoString < P) P=1; // на початок рядка
}
```

7.2.3 Рядки string

Клас рядків `string` стандартної бібліотеки C++ дозволяє працювати з рядками як зі звичайними типами: копіювати, присвоювати та порівнювати рядки як базові типи. Щоб використовувати рядки класу `string`, треба залучити до програми відповідний заголовний файл:

```
#include <string>
```

а також написати після цього оператор (пояснення див. підрозд. 9.5):

```
using namespace std;
```

Оголошення рядків `string` може бути таким:

```
string s;
string st("рядок символів\n");     // Оголошення рядка з ініціалізацією
string s(st);              // Ініціалізування рядка s змінною st типу string
```

Більш докладно способи оголошення й ініціалізації рядків розглянуто далі у прикладі 7.23.

Порівнювання двох рядків виконується за допомогою операцій відношення: ==, !=, <, <=, >, >=. Присвоювання (копіювання) рядків виконується за допомогою звичайного присвоювання, а конкатенація (сполучання, зчеплення) рядків виконується за допомогою операції "+".

```
s = st;                  // Рядок st присвоюється рядкові s
string s2 = s + st;      // Долучення рядків
s = s + st;              // Долучення рядка st у кінець рядка s
```

Операція конкатенації рядків може сполучати рядки `string` не лише між собою, але й з С-рядками, наприклад:

```
string s1 ("hello");
const char *pc=", ";
string s2 ("world");
string s3 = s1 + pc + s2 + "\0";
```

Цей вираз працює завдяки тому, що компілятор "знає", в який спосіб можна автоматично перетворювати С-рядки та рядки `string`. Можливе й просте присвоювання С-рядка рядкові `string`:

```
string s1;               // Оголошення порожнього рядка
const char *pc = "символьний масив";
s1 = pc;
```

Зворотне перетворення не працює. Спроба зробити наступну ініціалізацію С-рядка спричинить помилку компіляції:

```
char *str = s1;          // Помилка компіляції
```

Щоб здійснити таке перетворення, слід застосувати функцію `c_str()`, яка повертає вказівник на константний масив:

```
const char *str = s1.c_str();
```

Для введення рядків `string` з `Edit1` треба використовувати функцію `c_str()`:

```
string str = Edit1->Text.c_str();
```

Для виведення рядків string до Edit2, також треба використовувати функцію c_str():

```
Edit2->Text= str.c_str();
```

До окремих символів рядків string можна звертатися за допомогою індексу. Нумерація символів розпочинається з 0.

Найбільш поширені у застосовуванні функції опрацювання string-рядків наведено у табл. 7.5.

Таблиця 7.5

Деякі функції і методи опрацювання рядків string

Назва	Призначення	Формат
append()	долучає рядок s у кінець рядка	string& **append** (string& s);
assign()	замінює значення рядка на значення іншого рядка s	string& **assign** (string& s);
copy()	копіює n символів рядка, починаючи з позиції pos, у символьний масив s	size_t **copy** (char *s, size_t n, size_t pos = 0);
empty()	повертає true, якщо рядок є порожній. Приклад перевірки, чи є рядок порожнім: if (s.empty()). . .	bool **empty**();
erase()	вилучає частину рядка довжиною n символів, розпочинаючи з позиції p. Існує декілька форматів запису цієї функції	string **erase** (size_t p, size_t n);
find()	відшукує і повертає позицію першого входження підрядка str в рядок. Функція може мати два параметри, тоді у якості першого зазначається підрядок str, у якості другого – позиція pos, розпочинаючи з якої слід здійснювати пошук	size_t **find** (string &str);
		size_t **find** (char* str, size_t pos);
insert()	вставляє слово str у рядок на позицію p	string **insert** (size_t p, const string& str);
length()	обчислює довжину рядка, наприклад: int k=s.length();	size_t **length**();
replace()	замінює len елементів рядка, розпочинаючи з індексу idx, на символи рядка str. Існує кілька форматів запису цього методу	**replace** (size_t idx, size_t len, string& str);
reserve()	змінює рядок у такий спосіб, що всі його символи записуються у зворотному порядку, наприклад: s.reserve();	void **reserve**();
resize()	змінює розмір рядка на n. Якщо новий розмір є менше за довжину рядка, рядок усікається	void **resize** (size_t n);
size()	обчислює довжину рядка, наприклад: int k=s.size();	size_t **size**();

Закінчення табл. 7.5

Назва	Призначення	Формат
substr()	повертає частину рядка розміром n символів (чи то менше, якщо символів не вистачає), яка розпочинається з індексу pos	string **substr** (size_t pos, size_t n);
swap()	переставляє значення рядка і аргумента s, наприклад: s1.swap(s2);	void **swap** (string &s);

Розглянемо приклади деяких найпоширеніших операцій з рядками.

У рядку str всі крапки на знаки підкреслювання змінює наступний фрагмент програми

```
string str("fa.disney.com");
int n = str.size();
for(int i=0; i<n; i++)
  if(str[i] == '.') str[i] = '_';
```

Заміну крапок на знаки підкреслювання у рядку можна записати однією командою за допомогою функції replace():

```
replace(str.begin(), str.end(),'.', '_');
```

де str.begin() та str.end() – вказівники відповідно на початок і кінець рядка str.

Функція substring() копіює частину рядка. Наприклад:

```
string s1 = str.substr(n);
```

копіює у рядок s1 з рядка str всі символи з позиції n до кінця, а команда

```
string s2 = str.substr(n,12);
```

копіює у рядок s2 з рядка str 12 символів з позиції n.

Функція resize(n) встановлює довжину рядка. Якщо n<=s.size(), то "зайві" символи відкидаються.

Приклади програм з рядками string

Приклад 7.21 Ініціалізувати string-рядки у різні способи.

Текст програми у консолі:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{ char *line="short line for testing";
  string s1="Program";    //Звичайне ініціалізування
  cout << "s1  = " << s1 << endl;
  string s2(s1);          // Ініціалізування string-рядка; s2="Program"
  cout << "s2  = " << s2 << endl;
  string s3(line);     // Ініціалізування С-рядком; s3="short line for testing"
  cout << "s3  = " << s3 << endl;
  // Ініціалізування частиною С-рядка (з початку рядка):
  // перший аргумент – С-рядок, другий – кількість символів;
```

```
  string s4(line,10);      // s4="short line"
  cout << "s4  = " << s4 << endl;
  // Ініціалізування частиною С-рядка (з середини рядка): перший аргумент –
  // С-рядок, другий – початкова позиція, третій – кількість символів.
  string s5(s3,6,4);       // Копіювання слова "line" з рядка s3;
  cout << "s5  = " << s5 << endl;   // s5 ="line"
  // Ініціалізування рядком зазначених символів:
  // перший аргумент – кількість символів, другий – символ для ініціалізування;
  string s6(15,'*');       // s6 ="***************"
  cout << "s6  = " << s6 << endl;
  // Ініціалізування частиною string-рядка (з середини рядка): перший аргумент –
  // вказівник на початок частини, другий – вказівник на кінець частини;
  string s7(s3.begin(),s3.end()-5); //s7 ="short line for te"
  cout << "s7  = " << s7 << endl;
  // Можна ініціалізувати рядок присвоюванням:
  string s8 = "Program";             //s8 ="short line for te"
  cout << "s8  = " << s8 << endl;
  cin.get();   return 0;
}
```

Результат роботи програми:

```
s1  = Program
s2  = Program
s3  = short line for testing
s4  = short line
s5  = line
s6  = ***************
s7  = short line for te
s8  = Program
```

Приклад 7.22 Демонстрація роботи функції `append()`.

Текст програми у консолі:

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{ string str = "Borland C++ Builder";
  string s   = "";             // Порожній рядок
  char *ch   = "abcdef";
  // У кінець рядка s долучаються перші (з позиції 0) шість символів рядка str;
  s.append(str,0,6);           // s="Borland"
  cout << "s = " << s << endl;
  // У кінець рядка s долучаються символи рядка str з позиції 6 до кінця;
  s.append(str.begin()+6, str.end());// s="Borland C++ Borland"
  cout << "s = " << s << endl;
  // У кінець рядка s долучаються три знаки оклику
  s.append(3,'!');             //s= "Borland C++ Borland!!!"
```

```
  cout << "s = " << s << endl;
  // У кінець рядка s долучаються перші три символи С-рядка ch
  s.append(ch,3);                  //s= "Borland C++ Borland!!!abc"
  cout << "s = " << s << endl;
  cin.get();  return 0;
}
```

Результати виконання програми:

```
s = Borland
s = Borland C++ Builder
s = Borland C++ Builder!!!
s = Borland C++ Builder!!!abc
```

Приклад 7.23 Демонстрація роботи функції `assign()`.

Текст програми у консолі:

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{ string s = "",  str = "Borland C++ Builder";
  char *ch = "Character array";
  s.assign(str);     // Рядку s присвоюється рядок str
  cout << "s = " << s << endl;
  // Рядку s присвоюються 7 символів рядка str, розпочинаючи з позиції 10;
  s.assign(str,10,7);     // s = "C++ Build".
  cout << "s = " << s << endl;
  // Рядку s присвоюється С-рядок ch;
  s.assign(ch);           // s = " Character array "
  cout << "s = " << s << endl;
  // Рядку s присвоюються перші 9 символів С-рядка ch;
  s.assign(ch,9);         // s = " Character"
  cout << "s = " << s << endl;
  // Рядку s присвоюється частина рядка str з позиції  str.begin() до str.end();
  s.assign(str.begin(),str.end());  // s = "Borland C++ Builder"
  cout << "s = " << s << endl;
  // Рядку s присвоюється рядок з 17-ти зірочок;
  s.assign(17,'*'); // s = "*****************"
  cout << "s = " << s << endl;
  cin.get();   return 0;
}
```

Результати виконання програми:

```
s is : Borland C++ Builder
s is : C++ Build
s is : Character array
s is : Character
s is : Borland C++ Builder
s is : *****************
```

Приклад 7.24 Демонстрація роботи функцій `find()`, `copy()`. Програма шукає символ `':'` і копіює у символьний масив початок рядка `str` до віднайденого символу включно.

Текст програми у консолі:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{ string str = "It is 8:30 now";  char ch[255];
  cout << "str = " << str << endl;
  int n = str.find(':');  // Визначення позиції символу ':' у рядку str
     // У символьний масив ch копіюється n+1 символ з початку рядка str
  str.copy(ch, n+1, 0);
  // У кінець символьного масиву долучається '\0'
  ch[n+1] = 0;
  cout << "ch = " << ch << endl;
  cin.get();   return 0;
}
```

Результат виконання програми:

```
str = It is 8:30 now
ch = It is 8:
```

Приклад 7.25 Демонстрація роботи функцій `empty()`, `erase()`. Рядок `str` складається з 7-ми зірочок. Допоки цей рядок є непорожній, з нього послідовно вилучається останній символ.

Текст програми у консолі:

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{ string str = "*******";
  while( ! str.empty() )
   { cout << str << endl;
     str.erase(str.end()-1); }
  cout << endl;
  cin.get();  return 0;
}
```

Результат виконання програми:

```
*******
******
*****
****
***
**
*
```

Приклад 7.26 Два рядки заповнюються символами англійської абетки. Демонструється робота функції `erase()`.

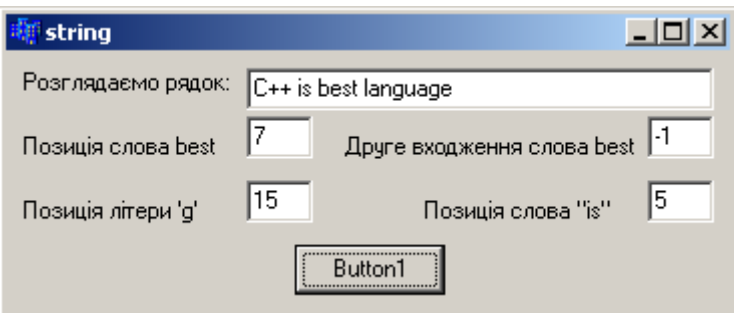
Текст програми у консолі:

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{ string str, s;
  for( char ch = 'a'; ch <= 'z'; ch++ )
    str.append(1,ch);
  s = str;
  cout << "str = " << str << endl;
  cout << "s   = " << str << endl<<endl;
  // Вилучається 13 символів від початку рядка str
  str.erase(0,13);
  cout << " str = " << str << endl;
  // Вилучається 13 символів рядка s, починаючи з 14-го
  str = s.erase(13,13);
  cout << " s   = " << str << endl<<endl;
  // Вилучається другий символ рядка s
  s.erase(s.begin()+1);
  cout << "s    = " << s << endl;
  // З рядка s вилучаються символи з першого до четвертого
  s.erase(s.begin(),s.begin()+4);
  cout << "s    = " << s << endl;
  cin.get();
  return 0;
}
```

Результати виконання програми

```
str = abcdefghijklmnopqrstuvwxyz
s   = abcdefghijklmnopqrstuvwxyz

str = nopqrstuvwxyz
s   = abcdefghijklm

s   = acdefghijklm
s   = fghijklm
```

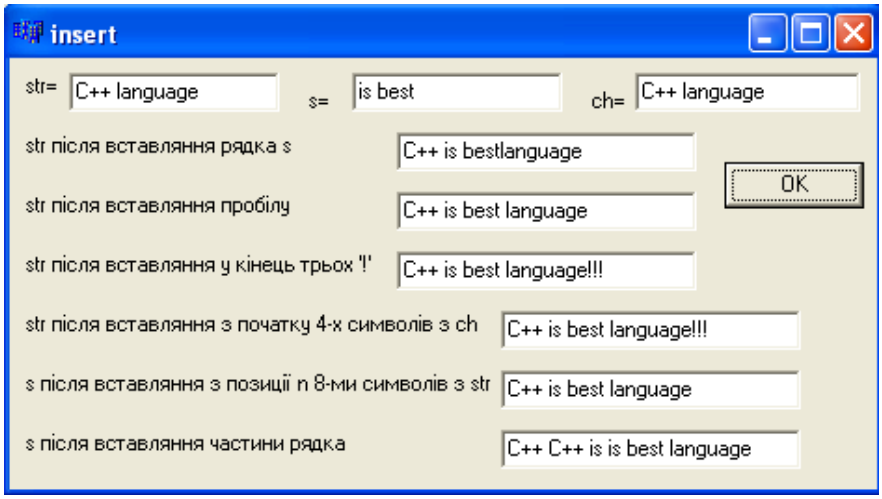
Приклад 7.27 Демонструється робота функції `find()`.



Текст програми:

```
#include <string>
using namespace std;
void __fastcall TForm1::Button1Click(TObject *Sender)
{ string  str("C++ is best language");
  Edit1->Text=str.c_str();
  int pos = str.find("best"); // Шукається позиція слова "best" у рядку str
  Edit2->Text= pos;
  // Шукається друге входження "best" у рядок, якщо не знайдено, повертається –1
  pos = str.find("best",pos+1);
  Edit3->Text=pos;
  pos = str.find('g');          // Шукається перше входження символу 'g'
  Edit4->Text= pos;
  string s = "is";
  pos = str.find(s);// Шукається перше входження слова "is" (string-рядок)
  Edit5->Text= pos+1;
}
```

Приклад 7.28 Демонструється робота функції `insert()`.



Текст програми:

```
#include <string>
using namespace std;
void __fastcall TForm1::Button1Click(TObject *Sender)
{ string str="C++ language";
  string s="is best";
  char ch[]="C++ language";
  Edit1->Text=str.c_str();
  Edit2->Text=s.c_str();
  Edit3->Text=AnsiString(ch);
  int pos=4;
  str.insert(pos,s); // Вставляння рядка s у рядок str на позицію pos
  Edit4->Text= str.c_str();
  int n = str.find('l');
```

```
  // Вставляння пробілу в рядок str на позицію str.begin() + n
  str.insert(str.begin() + n,' ');
  Edit5->Text= str.c_str();
  // Вставляння у кінець рядка str трьох знаків оклику
  str.insert(str.end(),3,'!');  Edit6->Text= str.c_str();
  // Вставляння на початок рядка s чотирьох символів з С-рядка ch
  s.insert(0,ch,4);             Edit7->Text= str.c_str();
  // Вставляння 8-ми символів з рядка str, починаючи з позиції n у рядок s на позицію X
  n = str.find('l');
  int X = s.length();
  s.insert(X,str,n,8);
  Edit8->Text= s.c_str();
  // Вставляння у рядок s частини рядку str з begin() до begin()+7
  // на позицію begin()+4
  s.insert(s.begin()+4,str.begin(),str.begin()+7);
  Edit9->Text= s.c_str();
}
```

7.3 Розширені символьні типи

7.3.1 Тип C++ wchar_t

Алфавіт більшості європейських мов може бути подано однобайтовими числами (тобто кодами в діапазоні від 0 по 255). Для більшості кодувань перші 127 кодів є однаковими і становлять набір ASCII-кодів: арабські цифри, великі й малі літери латиниці, спеціальні символи й розділові знаки (див. додаток А). Друга половина кодів – від 128 по 255 – відводиться під літери тієї чи іншої мови. Фактично друга половина кодової таблиці інтерпретується по-різному в різних кодуваннях для різних країн.

Проте для таких мов, як китайська, японська і деякі інші, одного байта вже недостатньо, щоб закодувати всі їхні символи, оскільки алфавіти цих мов нараховують понад 127 і навіть понад 255 символів. Тому для кодування символів стали відводити вже два байти, а таке кодування назвали **Unicode**. Перші 127 символів Unicode збігаються з ASCII-символами.

В мові C++ для опрацювання двобайтових символів існує спеціальний тип **`wchar_t`**. Цей тип іноді називають розширеним типом символів, і деталі його реалізації можуть варіюватися від компілятора до компілятора, зокрема може змінюватися і кількість байтів, яка відводиться під один символ; зазвичай вона відповідає типові `short` (2 байти). Константи типу `wchar_t` записуються з префіксом `L` у вигляді `L'ab'`:

```
wchar_t c=L'ab';
```

Рядкові константи типу `wchar_t*` записуються також з префіксом L, наприклад `L"Gates"`. Існує ціла низка спеціальних функцій для роботи з цими рядками. Майже всі функції для рядків `char*` мають аналоги для `wchar_t*`. Приміром, функція `wcslen()` визначає довжину рядка типу `wchar_t`, команди `wcin` та `wcout` вводять та виводять ці рядки, функції `swscanf()` та `swprintf()` вво-

дять і виводять їх у заданому форматі, функція `wcscat()` сполучає рядки, функції `wcschr()`, `wcspbrk()` та `wcsstr()` здійснюють пошук символів в таких рядках тощо.

За приклад опрацювання `wchar_t` рядків може слугувати програмний код

```
#include <iostream>
using namespace std;
int main()
{wchar_t *s1=L"Hello ",*s2=L"world";//Оголошення із ініціалізацією рядків
 int n=wcslen(s1);                  //Визначення довжини n рядка s1
 wcout << n <<" : " << s1 << endl;  //Виведення рядка s1
 n=wcslen(s2);
 wcout << n <<" : " << s2 << endl;
 wcscat(s1, s2);                    //Долучення s2 до s1
 n=wcslen(s1);
 wcout << n <<" : " << s1 << endl;
}
```

Результати виконання програми

```
6 : Hello
5 : world
11 : Hello world
```

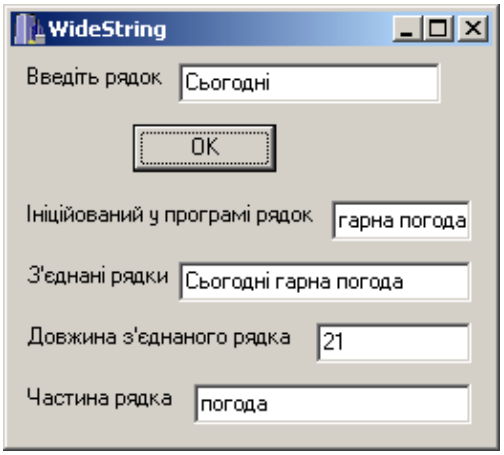
7.3.2 Тип C++ Builder WideString

Для опрацювання рядків розширених символів у C++ Builder існує клас **`WideString`**. Він є аналогом типу `WideString` у мові Object Pascal. Переважною відмінністю `WideString` від `AnsiString` є зберігання масиву розширених символів – wide characters типу `wchar_t*`. Тому здебільшого цей тип використовується в COM-додатках чи то при звертанні до OLE інтегрованих об'єктів.

Методи `WideString` є схожі до відповідних методів `AnsiString`. Найбільш оригінальним є метод `c_bstr()`, який повертає, за аналогією з `c_str()`, вказівник на масив `wchar_t*`.

У наведеному нижче прикладі програми проілюстровано основні особливості організації та опрацювання `WideString` рядків.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ WideString s, s1=Edit1->Text,
  s2="гарна погода";
  Edit1->Text=s1;
  Edit2->Text=s2;
  s=s1+" "+s2;
  Edit3->Text=s;
  int n=s.Length();
  Edit4->Text=n;
  s=s.SubString(16,6);
  Edit5->Text=s;
}
```



Питання та завдання для самоконтролю

1) Дайте означення терміну ASCII.
2) Яка є кількість ASCII-кодів? І чому саме?
3) Скільки байтів виділяється для змінної типу `char`?
4) Які операції можна виконувати над символами?
5) Назвіть вирази, які матимуть значення 1 (`true` – істина – "так"):
   а) `'0' < 'y'` г) `'w' > 'W'`
   б) `'5' > 'f'` д) `' ' > '0'`
   в) `'F' > 'f'` е) `'я' > 'ю'`
6) Як у програмі змінити регістр символів?
7) Чи можливо у програмі змінити регістр символів кирилиці? Якщо так, то в який спосіб?
8) Які типи рядків підтримуються в C++ Builder?
9) Наведіть відомі способи оголошення С-рядка.
10) Як ввести С-рядок із компонента `Edit`?
11) Чи можна присвоїти рядку `AnsiString` С-рядок? Якщо так, наведіть приклад.
12) Як присвоїти рядку `string` рядок `AnsiString`? Якщо так, наведіть приклад.
13) В який спосіб можна звернутися до окремого символу рядка?
14) Як порівняти два С-рядки? Наведіть приклад.
15) Чи можна присвоїти С-рядку частину іншого? Якщо так, наведіть приклад.
16) Яка функція C++ Builder перетворює ціле число на рядок?
17) Чи можна з'єднувати рядки, якщо так, то в який спосіб?
18) Наведіть способи введення С-рядка у консолі.
19) Поясніть, що виведеться на екран після виконання фрагменту програми:

```
char s1[]="слово";
char *s2=new char[5];
strncpy(s2, s1, 3);
puts(s2);
```

20) Як виправити логічну помилку, припущену у завданні 19?
21) Наведіть фрагменти коду для заміни всіх малих літер `'a'` на коми:
   а) у С-рядку;
   б) у рядку `string`;
   в) у рядку `AnsiString`.
22) Наведіть і поясніть фрагмент коду для видалення з С-рядка трьох символів, розпочинаючи з п'ятої позиції.
23) У рядку символів визначити кількість повторювань кожного слова та видалити дублікати слів. Слова відокремлюються пробілами.
24) У рядку символів довільного розміру видалити усі слова, довжина яких перевищує *N* літер.
25) Скопіювати до нового рядка частину рядка від початку до:
   а) першого пробілу; б) другого пробілу; в) останнього пробілу.

Розділ 8

Функції

8.1 Правила організації функцій

Коли програма стає завеликою за обсягом і складною для сприйняття, є сенс розділити її за змістом на невеликі логічні частини, називані *функціями*, кожна з яких виконуватиме певне завдання. Унаслідок цього програма стане більш легкою і для розуміння при створюванні, і для процесу налагодження. Окрім того, створення функції позбавляє потреби створювання двох, а то й і більшої кількості, майже однакових фрагментів програмного коду для розв'язання однакових завдань за різних вхідних даних. Розподіл програми на функції є базовим принципом структурного програмування.

Функція – це незалежна іменована частина програми, яка може багаторазово викликатися з інших частин програми, маніпулювати даними та повертати результати. Кожна функція має власне ім'я, за яким здійснюють її виклик. Розрізнюють два основні різновиди функцій: 1) стандартні убудовані функції, які є складовою частиною мови програмування, наприклад: `sin()`, `pow()` тощо і 2) функції, які створюються безпосередньо користувачем для власних потреб.

Створювана у програмі функція повинна бути *оголошеною* і *визначеною*. Оголошення функції має бути написаним у програмі раніш за її використання. Визначення може перебувати у будь-якому місці програми, за винятком тіла (середини) інших функцій. *Оголошення* функції складається з *прототипу* (чи *заголовка*) і має форму

[*клас*] <*тип результату*> <*ім'я функції*>
(<*перелік формальних аргументів із зазначенням типу*>);

Наприклад оголошення функції обчислення середньоарифметичного двох цілих чисел може мати вигляд

```
float seredne (int a, int b);
```

де `seredne` – ім'я функції;

`a` і `b` – формальні вхідні аргументи (параметри), які за виклику набувають значень фактичних параметрів;

`float` – тип функції, який безпосередньо є типом результату виконання операторів функції. Результат повертається оператором `return`.

Прототип функції може бути розташовано як безпосередньо у програмі, так і в заголовному файлі (див. далі п. 9.2).

Окрім оголошення, кожна функція повинна мати визначення (реалізацію). *Визначення* функції, окрім заголовку, містить тіло функції (команди, які виконує функція) і команду повертання результату `return`:

<*тип функції*> <*ім'я функції*> (<*перелік аргументів*>)
{ <*тіло функції*>
`return` <*результат обчислень*>

```
}
```

Наприклад, визначення функції `seredne` може бути таким:

```
float seredne (int a, int b)
{ float sr;
  sr=(a+b)/2.0;
  return sr;
}
```

Наведена функція обчислює `sr` – середньоарифметичне двох цілих чисел `a` та `b` і повертає його значення у точку виклику.

У разі, коли визначення функції розміщено у програмі раніш за точку виклику, писати окремо оголошення і окремо реалізацію цієї функції немає потреби і оголошення функції можна уникнути.

Якщо функції виконують певні обчислення й дії, які не потребують повертання результатів, за їхній тип вказують тип `void` (тобто порожній, без типу). У таких функціях оператор `return` може бути відсутнім чи записуватись без значення, яке повертається. Яскравим прикладом таких функцій є відгуки подій у C++ Builder, наприклад:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
}
```

У поданому нижче прикладі створювання власної консольної функції `max3()` всі обчислення для пошуку максимального з трьох дійсних чисел `x`, `y` та `z`, а також дії стосовно виведення результатів обчислень відбуваються у функції і немає потреби повертати їх через `return`:

```
void max3(float x, float y, float z)
{ if (x>y && x>z) cout<<x;
  else
    if (y>x && y>z) cout<<y;
    else cout<<z;
}
```

Приклади створювання функцій з типом `void` вже розглядались у розд. 5 при опрацюванні числових масивів у функціях.

Виконання функції розпочинається, коли у тексті програми зустрічається оператор виклику, тобто ім’я функції з фактичними параметрами у дужках. При цьому формальні параметри послідовно набувають значень фактичних: значення першого фактичного аргументу копіюється у перший формальний аргумент, другий – у другий і т. д. А тому дуже важливими чинниками є кількість, порядок записування і відповідність типів всіх аргументів, інакше це може призвести до серії помилок.

Якщо функцію оголошено з будь-яким типом, окрім типу `void`, тобто вона повертає певний результат, при викликанні такої функції її варто присвоїти змінній того ж самого типу, що й сама функція, чи то вивести на екран за допомогою оператора виведення. Наприклад, функцію `seredne()` може бути викликано в один зі способів:

```
int a=5, x=2, y=-3, z, s1, s2;
1) z = seredne(5, 4);          // z = 4.5
2) s1 = seredne(a, 11);        // s1 = 8
3) s2 = seredne(x, y);         // s2 = –0.5
```

У першому наведеному виклику числа 5 та 4 є фактичними параметрами, які підставляються у дужки замість формальних параметрів `a` та `b`:

```
float seredne(int a, int b)
                  ↑      ↑
    z = seredne( 5,      4);
```

У другому виклику першим фактичним параметром є змінна `a`, а другим – число 10. У третьому – обидва фактичних параметри є змінними.

Вищенаведена функція `max3()` має тип `void`, тому її виклик може мати вигляд

```
max3(5, 1, 14);
```

У результаті цієї команди здійсниться виклик функції `max3()`, яка з трьох чисел: 5, 1, 14 віднайде максимальне (14) і його буде виведено на екран.

Функція може бути оголошена *убудованою* (`inline`):

```
inline int modul(int x)
{ if(x<0) return –i;
  else return i;
}
```

У такому разі компілятор підставляє у точку виклику тіло цієї функції.

Правила організації функцій

1) Фактичними параметрами можуть бути константи, змінні чи вирази. В останньому разі спочатку обчислюється значення виразу, а потім результат передається у функцію.

2) Імена змінних, які є фактичними параметрами, можуть не збігатися з іменами формальних параметрів.

3) Типи змінних, які є фактичними параметрами, мають збігатися з типами відповідних формальних параметрів чи то мати можливість бути перетвореними до типів формальних параметрів.

4) Якщо кількість чи типи фактичних параметрів не збігаються з формальними параметрами, це призведе до помилки з відповідним повідомленням.

5) Послідовність змінних, які є фактичними параметрами, має збігатися з послідовністю формальних параметрів.

6) У прототипі функції необов’язково задавати імена формальних параметрів, достатньо їх оголосити, наприклад у такий спосіб:

```
float seredne(int, int);
```

7) Якщо функція не отримує жодного параметра, слід у заголовку функції ставити порожні дужки (опустити дужки не можна) чи записати у дужках `void`:

```
double func();
double func(void);
```

8) Тип значення, яке повертає функція, може бути яким-завгодно, але не може бути масивом чи функцією. Наприклад, таке оголошення є помилкове:

```
int [10] func();   // Помилка!
```

Якщо функція має повернути масив, то можна оголосити її тип як вказівник на перший елемент масиву (кількість елементів цього масиву має бути відомою):

```
int * func();
```

9) C++ дозволяє існування у програмі функцій з однаковим ім'ям, але різними параметрами.

Функції використовують пам'ять зі стека програми. Певна область стека надається функції й залишається пов'язаною з нею до завершення її роботи. Після завершення роботи надана для функції пам'ять звільнюється й може бути зайнята іншою функцією.

Усі величини, оголошені всередині функції, а також її параметри, є *локальними*. Областю їхньої дії є функція, тобто для решти функцій програми ці змінні та їхні значення залишаються невідомими. Глобальні змінні є відомими всім функціям програми, кожна з функцій може змінювати значення таких змінних. Використовування глобальних змінних часто призводить до помилок, які важко відшукати. Тому слід уникати використовування глобальних змінних, якщо це не є нагальною потребою.

При викликанні функції, як і при вході до кожного блока, у стеку виділяється пам'ять під локальні автоматичні змінні. При виході з функції пам'ять, яку було виділено під локальні змінні, звільнюється. За спільної роботи функції обмінюються даними. Це можна здійснити за допомогою *глобальних* змінних, через *параметри* й через *значення*, які повертає функція.

У розглянутому нижче прикладі змінні з ім'ям `x` визначені одразу у двох функціях – в `main()` і в `Sum()`, що не заважає компіляторові розрізнювати їх поміж собою:

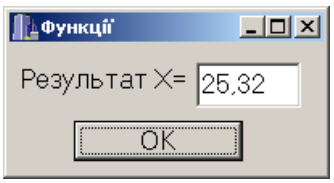
```
int Sum(int A, int B)
{ // Локальну змінну x і параметри A та B видно лише у тілі функції Sum()
 int x = A + B;
 return x;
}

void main()
{ int x = 2, y = 4;     // x, y – локальні змінні для main()
  cout << Sum(x, y);
}
```

Глобальні змінні є видимими у всіх функціях, де не описані локальні змінні з тими ж іменами, тому використовувати їх для передаванні даних між функціями дуже легко. Якщо імена глобальної й локальної змінної збігаються, перевага надається локальній змінній. Проте використовувати змінні з однаковими іменами не рекомендовано, оскільки це ускладнює налагодження програми. Слід прагнути до того, щоб функції були максимально незалежними, а їхній інтерфейс повністю визначався би прототипом функції.

Приклади програм з функціями

Приклад 8.1 Обчислити значення виразу $x=\frac{\sqrt{6}+6}{2}+\frac{\sqrt{13}+13}{2}+\frac{\sqrt{21}+21}{2}$.

Розв'язок. Усі три доданки заданої формули є схожі один з одним. Тому кожен з них можна записати за допомогою спільної формули: $\frac{\sqrt{a}+a}{2}$, де a – число, яке у першому доданку дорівнює 6, у другому – 13, а у третьому – 21. Слушно є створити функцію, параметром якої буде дійсне число a і яка обчислюватиме значення цієї формули. У програмі функцію буде викликано тричі для кожного з доданків: 6, 13, 21.

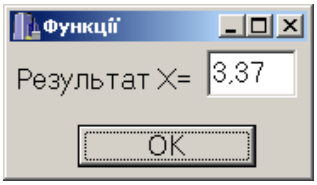


Текст функції та її виклику в основній програмі:

```
double f(double a)
{ return (sqrt(a)+a)/2; }

void __fastcall TForm1::Button1Click(TObject *Sender)
{ double x=f(6)+f(13)+f(21);
  Edit1->Text=FormatFloat("0.00", x);
}
```

Приклад 8.2 Обчислити значення виразу $x=\frac{13+\sqrt{7}}{7+\sqrt{13}}+\frac{15+\sqrt{12}}{12+\sqrt{15}}+\frac{21+\sqrt{32}}{32+\sqrt{21}}$.

Розв'язок. Формула цього прикладу схожа на формулу попереднього завдання і відрізнюється лише тим, що функція тут матиме два параметри.



Текст функції та основної програми:

```
double f(double a, double b)
{ return (a+sqrt(b))/(b+sqrt(a)); }
void __fastcall
Form1::Button1Click(TObject *Sender)
{ double x=f(13,7)+f(15,12)+f(21,32);
  Edit1->Text=FormatFloat("0.00", x);
}
```

Приклад 8.3 Обчислити значення виразу $z=\frac{2\cdot 5!+3\cdot 8!}{6!+4!}$.

Розв'язок. У формулі чотири рази зустрічається факторіал. Тому доречно обчислити значення факторіалу у функції і викликати його відповідно чотири рази для різних параметрів.

Текст функції та основної програми:

```
long fact(int n)
{ long c=1;
  for(int i=1; i<=n; i++) c*=i;
  return c; }
```

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ double z=(2.0*fact(5)+3*fact(8))/(fact(6)+fact(4));
  Edit1->Text=FloatToStr(z);
}
```

Приклад 8.4 Віднайти периметр трикутника, заданого координатами вершин.

Розв'язок. Для обчислення довжини сторони трикутника доцільно створити функцію і викликати її тричі для кожної зі сторін. Ця функція обчислюватиме довжини відрізку, заданого координатами точок, які він з'єднує, за формулою

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} .$$

Функція `segment()` має чотири параметри – координати обох кінців відрізка. Результат функції – дійсне число (довжина відрізка).

Тексти функції та її виклику в основній консольній програмі:

```
double segment(double x1,double y1,double x2,double y2)
{ return sqrt(pow(x2-x1,2)+pow(y2-y1,2));
}
void main()
{ double x1, y1, x2, y2, x3, y3, P;
  cout<<"Перша вершина:\n";
  cout<<"Уведіть координату x: "; cin>>x1;
  cout<<"Уведіть координату y: "; cin>>y1;
  cout<<"Друга вершина:\n";
  cout<<"Уведіть координату x: "; cin>>x2;
  cout<<"УВведіть координату y: "; cin>>y2;
  cout<<"Третя вершина:\n";
  cout<<"Уведіть координату x: "; cin>>x3;
  cout<<"Уведіть координату y: "; cin>>y3;
  P=segment(x1,y1,x2,y2)+segment(x2,y2,x3,y3)+
                                    segment(x1,y1,x3,y3);
  cout<<"Периметр дорівнює: "<<P<<endl;
}
```

Результати виконання програми:

```
Перша вершина:
Уведіть координату x: 3
Уведіть координату y: 5
Друга вершина:
Уведіть координату x: 1
Уведіть координату y: 7
Третя вершина:
Уведіть координату x: 4
Уведіть координату y: 2
Периметр дорівнює: 11.8217
```

8.2 Способи передавання параметрів до функцій

Значення автоматичних локальних змінних між викликами однієї й тієї самої функції не зберігаються. Якщо значення певних локальних змінних треба зберігати, при їхньому оголошуванні слід використовувати специфікатор **static**:

```
void f(int a)
{ int m=0;
  cout<<"n m p\n";
  while (a--)
  { static int n=0;
    int p=0;
    cout<<n++<<' '<<m++<<' '<<p++<<'\n';
} }
int main()
{ f(3);  f(2);
  return 0;
}
```

Статична змінна `n` оголошується й ініціалізується *одноразово* за першого виконання оператора, який містить її визначення. *Автоматична* змінна `m` оголошується й ініціалізується за *кожного* входження до функції. Автоматична змінна `p` оголошується й ініціалізується за *кожного* входження до блока циклу. Програма виведе на екран:

```
n m p
0 0 0
1 1 0
2 2 0
n m p
3 0 0
4 1 0
```

При викликанні функції першою чергою обчислюються вирази, які стоять на місці фактичних параметрів; потім у стеку виділяється пам’ять під формальні параметри функції відповідно до їхнього типу і кожному з них присвоюється значення відповідного фактичного параметра. При цьому перевіряється відповідність типів і, за потреби, виконуються їхні перетворювання. За невідповідності типів видається діагностичне повідомлення.

Існує два способи передавання параметрів до функції: за значенням і за адресою.

За передавання *за значенням* до стека заносяться *копії* фактичних параметрів і оператори функції працюють з цими копіями. Доступу до вихідних значень параметрів у функції немає, а, отже немає й можливості їх змінити.

За передавання *за адресою* до стека заносяться *копії адрес* параметрів, а функція здійснює доступ до комірок пам’яті за цими адресами і може змінювати вихідні значення параметрів. Передавати параметри за адресою можна у два способи: за допомогою *вказівника* і *посилання*.

За обох способів до функції передається адреса параметра, який зазначено при викликанні. При передаванні вказівника всередині функції виникає потреба використовувати явно операції розіменування чи то отримання адреси, що робить менш зрозумілим текст коду функції і більш складним налагодження функції. За передавання за посиланням до функції передається посилання, тобто синонім імені, внаслідок чого всередині функції всі звертання до параметра неявно розіменовуються.

Отже, використовування посилань замість вказівників зоптимізовує читабельність програми, позбавляючи потреби застосовувати операції отримання адреси й розіменування. Використовування посилань замість передавання за значенням є більш ефективне, оскільки не потребує копіювання параметрів, а це є надто важливо при передаванні структур даних великого обсягу.

Розглянемо приклад функції з передаванням параметрів за посиланням:

```
void duplicate(int& a, int& b, int& c)
{ a*=2;  b*=2;  c*=2;
}
int main ()
{ int x=1, y=3, z=7;
  duplicate(x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

Результат:

```
x=2, y=6, z=14
```

У наведеному фрагменті функція `duplicate()` отримає три цілих параметри, кожний з яких збільшується вдвічі у тілі функції. Для того щоб ці зміни були відомі у точці виклику (в функції `main()`), всі параметри передаються за посиланнями. Фактичними параметрами є самі змінні `x`, `y` та `z`, а не копії їхніх значень. Функція підставляє змінну `x` замість формального параметра `a`, змінну `y` – замість формального параметра `b`, змінну `z` – замість формального параметра `c`:

```
void duplicate(int& a, int& b, int& c)
               ↕       ↕       ↕
     duplicate( x,      y,      z);
```

Розглянемо детальніше різницю поміж передаванням параметра за значенням, за посиланням і за вказівником.

У поданому нижче фрагменті програми функція `f` має три параметри, один з яких передається за значенням, другий – за вказівником, а третій – за посиланням.

Тексти функції та її виклику в основній програмі:

```
void f(int i, int* j, int& k);
int main()
{ int i=1, j = 2, k = 3;
  cout << "i j k\n";
```

```
  cout <<i<<' '<<j<<' '<<k<<'\n';
  f(i,&j,k);
  cout<<i<<' '<<j<<' '<<k;
}
void  f(int i, int* j, int& k)
{ i++; (*j)++; k++;
}
```

Результат роботи програми:

```
i j k
1 2 3
1 3 4
```

Перший параметр `i` передається до функції за значенням. Його змінення у функції не впливає на вихідне значення. Другий параметр `j` передається за адресою за допомогою вказівника, при цьому для передавання до функції адреси фактичного параметра використовується операція отримання адреси `&j`, а для одержання його значення у функції потрібна операція розіменування (розадресації) `*j`. Третій параметр `k` передається за адресою за допомогою посилання. Це дозволяє передавати фактичний параметр `k` без операції отримання адреси і використовувати його у функції без розадресації.

Якщо виникає потреба уникнути змінювання параметра всередині функції, використовується специфікатор `const`:

```
int f(const char*);
char *t(char *a, const int *b);
```

На місце параметра типу `const&` може передаватися константа, а для змінної за потреби – виконуватися перетворення типу. Вихідні дані, які не повинні змінюватися у функції, більш оптимально є передавати їй за допомогою константних посилань.

За замовчуванням параметри будь-якого типу, окрім масиву й функції, передаються до функції за значенням.

Розглянемо ще один приклад функцій з параметрами, які передаються за посиланням і за вказівником:

```
void swap_values1(float *a, float *b)
{ float temp = *a;
  *a = *b;
  *b = temp;
}
void swap_values2(float &a, float &b)
{ float temp = a;
  a = b;
  b = temp;
}
```

До функції `swap_values1()` параметри передаються за вказівником. Виклик цієї функції може мати вигляд

```
a=3; b=5;
swap_values1(&a, &b);
```

Звертаємо увагу, що фактичними параметрами при передаванні за посиланням і вказівником можуть бути лише змінні.

До функції `swap_values2()` параметри передаються за посиланням. Виклик цієї функції матиме більш природній вигляд:

```
a=3; b=5;
swap_values2(a, b);
```

Якщо функція має повернути понад одне значення, можна передати їй за посиланням додаткові параметри, а у тілі функції їм буде присвоєно значення, які треба повернути. Наступна функція `prevnext()` отримає ціле число `x` і обчислюватиме попереднє `prev` та наступне `next` значення відносно введеного `x`:

```
void prevnext (int x, int& prev, int& next)
{ prev = x-1;  next = x+1; }
```

При використовуванні *масиву* як параметра функції передається вказівник на його перший елемент, тобто масив завжди передається за адресою (див. п. 5.2.4). При цьому інформація про кількість елементів втрачається і слід передавати його розмірність через окремий параметр. Якщо розмірність масиву є константою, проблем не виникає, оскільки можна зазначити її при оголошенні формального параметра і як верхню межу циклів при опрацюванні масиву всередині функції. У разі передавання до функції масиву символів, тобто рядка, його фактичну довжину можна визначити за розташуванням нуль-символу і передавати кількість елементів немає потреби.

У поданому нижче фрагменті програми оголошується функція `sum()`, параметрами якої є масив і кількість його елементів. Ця функція обчислює суму елементів масиву.

```
int sum (int* mas, int n)
{// Варіанти: int sum(int mas[], int n) чи int sum(int mas[n]) (якщо n – константа).
  for(int i = 0, s = 0; i < n; i++) s += mas[i];
  return s;
}
void main()
{ int marks[] = {3, 4, 5, 4, 4};
  cout << "Сума елементів масиву: "<< sum(marks, 5);
}
```

При передаванні до функції *багатовимірних масивів* усі розмірності, якщо вони є невідомі на етапі компіляції, мають передаватися як параметри (див. п. 5.3.4). Приміром, заголовок функції, яка обчислює суму елементів динамічного двовимірного масиву, може мати вигляд

```
int sum(const int **a, const int nr, const int nc);
```

Виклик цієї функції у програмі може бути таким:

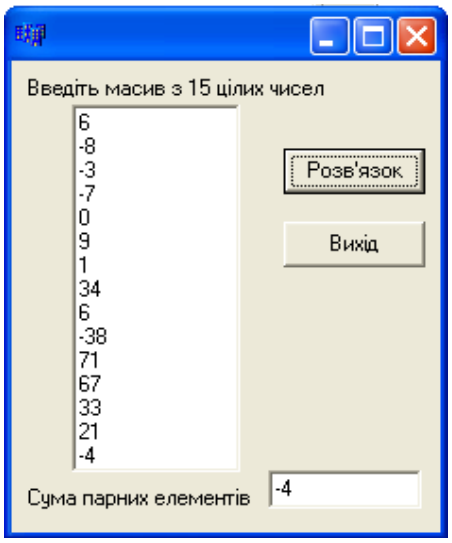
```
cout << "Сума елементів a: " << sum((const int**)a,nstr,nstb);
```

Для звичайного статичного двовимірного масиву, коли обидві розмірності є відомі й є константами, заголовок функції матиме вигляд

```
int sum(int a[4][6]);
```

Приклад 8.5 Увести масив з 15 цілих чисел та обчислити суму його парних елементів.

Розв'язок. Функція `sum_par()` має один параметр – масив `A`. Оскільки кількість елементів задано константою, вона є відома і передавати її до функції не треба. Результат функції має цілий тип `int`.



Тексти функції та основної програми:

```
int sum_par(int A[10])
{ for(int i=0, sum=0; i<10; i++)
    if(A[i]%2==0) sum+=A[i];
  return sum;
}
void __fastcall TForm1::Button1Click
                (TObject *Sender)
{ int i, a[10];
  if(Memo1->Lines->Count<10)
    {ShowMessage("Введіть ще числа");
     return;
    }
  for(i=0; i<10; i++) a[i]=StrToInt(Memo1->Lines->Strings[i]);
  Edit1->Text=IntToStr(sum_par(a));
}
```

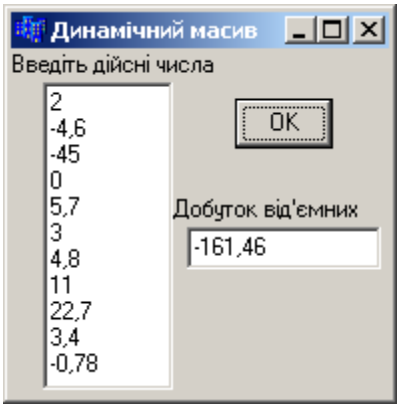
Приклад 8.6 Увести масив і обчислити за допомогою функції добуток від'ємних елементів одновимірного масиву.

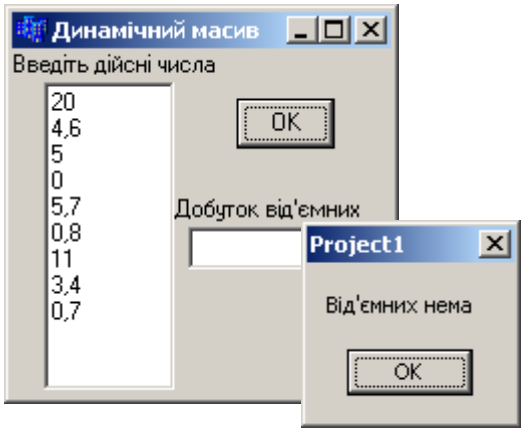
Розв'язок. Оскільки в завданні нічого не сказано про кількість елементів, вважатимемо її довільною. Введемо числа у `Memo1` і створимо з них масив. Цей масив буде динамічним, оскільки кількість елементів є заздалегідь невідома.

Функція повертатиме дійсне значення і має два параметри: вказівник на початок масиву і кількість елементів. Якщо від'ємних елементів нема, буде повернуто значення добутка 0.

Тексти функції таі її виклику в основній програмі:

```
double dob(double *a, int n)
{ int i, k=0;  double p=1;
  for(i=0; i<n; i++)
    if(a[i]<0) { k++;  p *= a[i];   }
  if(k>0) return p;  else return 0; }
```



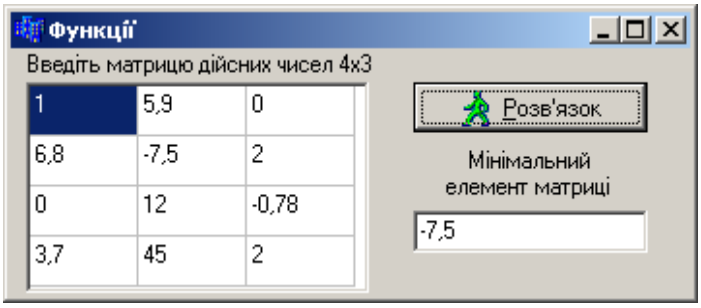


```
void __fastcall TForm1::Button1Click (TObject *Sender)
{ double P;    int i, n = Memo1->Lines->Count;
  double *a = new double [n];
  for(i=0; i<n; i++) a[i]=StrToFloat(Memo1->Lines->Strings[i]);
  P = dob(a, n);
  if(P!=0) Edit1->Text=FloatToStr(P);
  else ShowMessage("Від'ємних нема");
}
```

Приклад 8.7 Увести матрицю дійсних чисел розмірності 4×3 й обчислити за допомогою функції мінімальний елемент матриці.

Тексти функції та її виклику в основній програмі:

```
 double matrix_min(double a[4][3])
{ double min=a[0][0];
  for(int i=0; i<4; i++)
  for(int j=0; j<3; j++)
   if(a[i][j]<min)
     min=a[i][j];
  return min;
}
```




```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{ double a[4][3], m;
  for(int i=0; i<4; i++)
  for(int j=0; j<3; j++)
    a[i][j]=StrToFloat(StringGrid1->Cells[j][i]);
  m=matrix_min(a);
  Edit1->Text=FloatToStr(m);
}
```

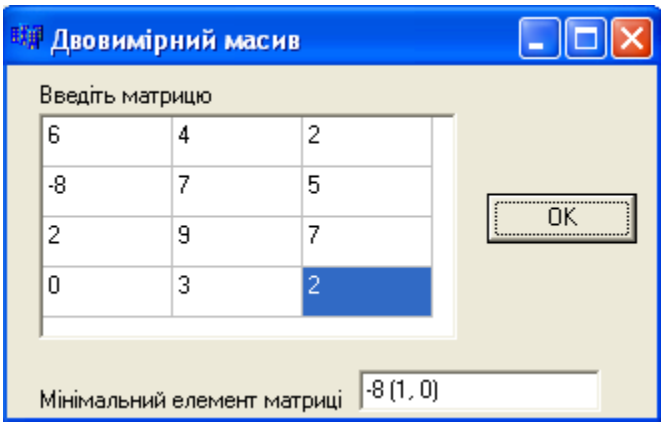
Приклад 8.8 Увести матрицю дійсних чисел розмірності 4×3 й обчислити за допомогою функції мінімальний елемент матриці та його індекси.

Розв'язок. Реалізацію подібного завдання без створення функції вже було розглянуто в підрозд. 5.3. До того ж цей приклад є продовженням попереднього. Змінимо функцію в такий спосіб, щоб вона обчислювала і повертала три значення. У цьому разі слід передати до функції два додаткових параметри – `ind_i` та `ind_j`, – в які буде записано обидва індекси мінімального елемента. Ці параметри мають передаватися за посиланням (чи за вказівником), щоб у тілі функції можна було їх змінити і повернути ці зміни у точку виклику.

Тексти функції та її виклику в основній програмі:

```
double matrix_min(double a[4][3], int& ind_i, int& ind_j)
{ double min=a[0][0];  ind_i=0; ind_j=0;
  for(int i=0; i<4; i++)
  for(int j=0; j<3; j++)
   if(a[i][j]<min) { min=a[i][j]; ind_i=i;  ind_j=j;  }
  return min;   }
```

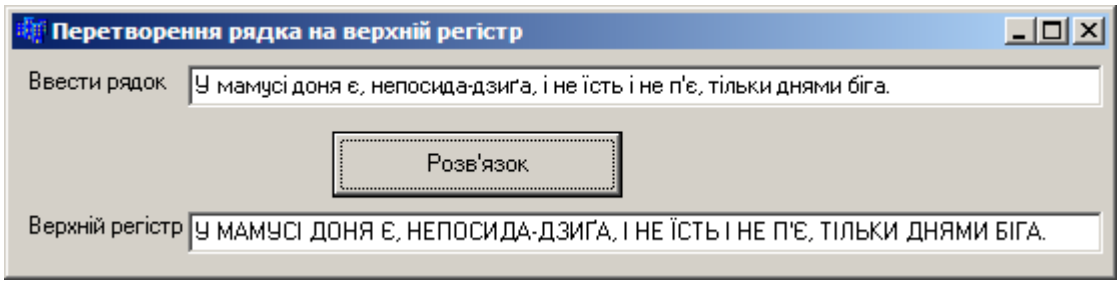
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ double a[4][3], m;    int i, j, ind_i=0, ind_j=0;
  for(i=0;i<4;i++)
  for(j=0;j<3;j++) a[i][j]=StrToInt(StringGrid1->Cells[j][i]);
  m = matrix_min(a, ind_i, ind_j);
  Edit1->Text = FloatToStr(m) + "(" + IntToStr(ind_i) + ", " +
                IntToStr(ind_j) + ")";
}
```



Приклад 8.9 Увести рядок і перетворити його на верхній регістр.

Розв'язок. Створимо функцію для перетворення будь-яких літер (у тому числі й українських) на верхній регістр. З попереднього розділу відомо, що функції `isupper()` та `islower()` не працюють з літерами кирилиці. Коди більшості малих російських літер є більші, аніж коди відповідних великих літер, на 32. Але в українській абетці є кілька специфічних літер, місцеположення яких у таблиці не підпадає під цю залежність при обчислюванні великої літери, а тому їх слід перевіряти окремо.

Оскільки, на відміну від звичайних масивів, кінець рядка можна визначити за розміщенням нуль-символу, немає потреби передавати до функції довжину рядка, тобто рядок можна передавати як вказівник на його перший символ. При такому способі передаванні будь-які змінення рядка всередині функції автоматично змінюють вихідний рядок.



Тексти функції та її виклику в основній програмі:

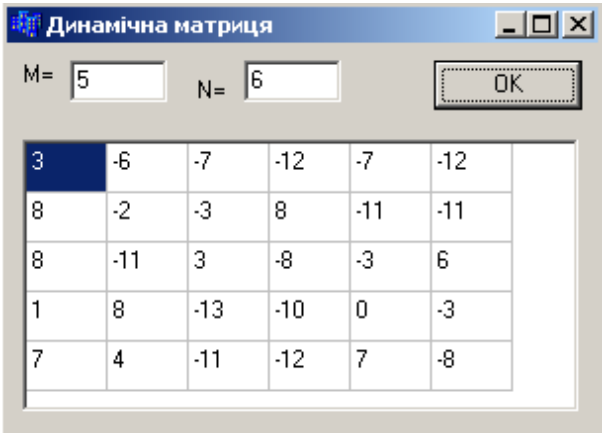
```
void toUp(char *c)
{ int n=strlen(c);
  for(int i=0; i<n; i++)
  { if(isalpha(c[i])) c[i] = toupper(c[i]);
    if(c[i] >= 'а' && c[i] <= 'я') c[i] -= 32;
```

```
      if(c[i] == 'ї' || c[i] == 'є') c[i] = c[i] - 16;
      if(c[i] == 'ґ') c[i] = 'Ґ';
      if(c[i] == 'і') c[i] = 'І';
    }
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ char s[50];
  strcpy(s, Edit1->Text.c_str());
  toUp(s);
  Edit2->Text=AnsiString(s);
}
```

Приклад 8.10 Заповнити матрицю розмірності M×N (значення M та N увести) випадковими числами з проміжку [–15, 10) за допомогою функції.

Тексти функції та її виклику в основній програмі:

```
void fill_random(int** a, int M, int N)
{ randomize();
  for(int i=0; i<M; i++)
  for(int j=0; j<N; j++)  a[i][j]=random(25)-15;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int M, N, i, j;
  if(Edit1->Text=="" || Edit2->Text=="")
   { ShowMessage("Введіть M та N"); return; }
  M=StrToInt(Edit1->Text);
  N=StrToInt(Edit2->Text);
  StringGrid1->RowCount=M;
  StringGrid1->ColCount=N;
  int **a = new int* [M];
  for(i=0; i<M; i++)
    a[i]=new int [N];
  fill_random(a, M, N);
  for(i=0; i<M; i++)
  for(j=0; j<N; j++)
   StringGrid1->Cells[j][i]=
           IntToStr(a[i][j]);
  for(i=0; i<M; i++) delete []a[i];
  delete []a;
}
```



8.3 Параметри зі значеннями за замовчуванням

Щоб спростити виклик функції, в її заголовку можна задати значення параметрів за замовчуванням. Ці параметри мають бути останніми у списку й можуть опускатися при виклику функції. Якщо при виклику параметр є опущений, має бути опущено й усі параметри, що стоять за ним. Як значення параметрів за замовчуванням можуть використовуватися константи, глобальні змінні й

вирази. Розглянемо приклади прототипів функцій з параметрами за замовчуванням.

```
int f(int a, int b=0);     // Параметр b має значення за замовчуванням 0.
void f1(int, int=100, char*  = 0);// Зверніть увагу на пробіл між * й =,
                          //без нього б вийшла операція складного присвоювання *=
void err(int errValue = errno);    //errno – глобальна змінна.
```

Варіанти виклику цих функцій:

```
f(100); // Виклик функції з першим параметром 100, другий за замовчуванням – 0.
f(A, 1); // Виклик функції з першим параметром – значенням змінної A, другим – 1.
f1(A);  // Виклик функції з другим та третім параметрами за замовчуванням.
f1(a, 10); // Виклик функції зі значенням третього параметра за замовчуванням.
f1(a, 10, "Hello");   // Виклик без значень параметрів за замовчуванням.
```

Приклад 8.11 Написати функцію, яка виконує ділення одного цілого числа на друге. Якщо друге число не зазначено, то виконується ділення на 2.

Тексти функції та її виклику в основній програмі:

```
double divide(int a, int b=2)
{ return (double)a/b; }
void main()
{ cout << divide(12);     //12:2
  cout << endl;
  cout << divide(20,4);   //20:4
  getch();
}
```

Результати виконання програми:

```
6
5
```

8.4 Функції зі змінною кількістю параметрів

Якщо список формальних параметрів функції завершується трьома крапками, це означає, що при її виклику на цьому місці можна зазначити ще кілька параметрів. Перевірка відповідності типів для цих параметрів не виконується, `char` та `short` передаються як `int`, а `float` – як `double`. За приклад стандартної функції зі змінним числом параметрів можна навести функцію `printf()`, прототип якої має вигляд

```
int printf (const char*, ...)
```

Це означає, що виклик функції має містити принаймні один параметр типу `char*` й може чи то містити, чи не містити інші параметри:

```
printf("Уведіть вихідні дані");      // Один параметр
printf("Сума: %5.2f рублів", sum);   // Два параметри
printf("%d %d %d %d", a, b, c, d);   // П'ять параметрів
```

Для роботи з функціями зі змінною кількістю аргументів використовуються величини va_list, va_start, va_arg і va_end, описані в заголовному файлі <stdarg.h>.

Тип va_list призначено для зберігання вказівника на черговий аргумент.

Для роботи з функціями зі змінною кількістю аргументів використовуються декілька стандартних макросів (дещо подібних до функцій наборів команд).

Макрос va_start ініціює вказівник на черговий елемент, він встановлює аргумент arg_ptr на початок списку необов'язкових параметрів і має вигляд функції з двома параметрами:

```
void va_start(arg_ptr, prav_param);
```

де параметр prav_param має бути останнім обов'язковим параметром викликуваної функції, а вказівник arg_ptr має бути оголошено у списку змінних типу va_list у вигляді

```
va_list arg_pt;
```

Макрос va_start має бути використано до першого використання макросу va_arg. Макрос va_arg повертає значення чергового аргументу, кожен його виклик призводить до просування вказівника, який зберігається в va_list. Макрокоманда va_arg забезпечує доступ до поточного параметра викликуваної функції й теж має вигляд функції із двома параметрами:

```
type_arg va_arg (arg_ptr, type);
```

Ця макрокоманда бере значення типу type за адресою, заданою вказівником arg_ptr, збільшує значення вказівника arg_ptr на довжину використаного параметра (довжина type) і в такий спосіб параметр arg_ptr вказуватиме на наступний параметр викликуваної функції. Макрокоманда va_arg використається стільки разів, скільки треба для отримання всіх параметрів викликуваної функції.

Після перебирання аргументів, але до виходу з функції зі змінним числом аргументів, необхідно звернутися до макросу va_end. Він встановлює вказівник списку необов'язкових параметрів на NULL.

Приклад 8.12 Увести послідовність цілих чисел і обчислити за допомогою функції зі змінною кількістю параметрів середнє значення цієї послідовності. За завершення послідовності слід вважати значення, яке дорівнює –1.

Розв'язок. Оскільки у списку має бути принаймні один елемент, у функції буде один обов'язковий параметр x, після якого слід поставити три крапки.

Тексти функції та її виклику в основній програмі:

```
float sred_znach (int x, ...)
{ int i=0,  j=0,  sum=0;
  va_list uk_arg;
  va_start(uk_arg, x);     // Встановлення вказівника uk_arg на перший
                           // необов'язковий параметр.
  if(x!=-1) sum=x;         // Перевірка наявності чисел у списку.
  else return 0;
```

```
  j++;
  while((i=va_arg(uk_arg, int))!=-1) // Вибірка чергового параметра
     {sum+=i; j++;}          // й перевірка завершення списку
  va_end(uk_arg);            //Закриття списку параметрів
  return (float)sum/j;
}
void main()
{ float sr=sred_znach(2,3,4,-1);      // Виклик з чотирма параметрами
  cout << "sr=" << sr << endl;
  sr=sred_znach(5,6,7,8,9,-1);        // Виклик з шістьма параметрами
  cout << "sr=" << sr;
  getch();
}
```

8.5 Рекурсивні функції

Рекурсивною називається функція, яка викликає сама себе. Така рекурсія називається прямою. Існує ще непряма рекурсія, коли дві чи більше функцій викликають одна одну.

Несерйозні приклади, які добре ілюструють принцип рекурсії, можна знайти в дитячих лічилках, наприклад:

У попа був собака, він її любив
Вона з'їла шматок м'яса, він її убив
У землю закопав,
Напис написав:
“У попа був собака, він її любив
Вона з'їла шматок м'яса, він її убив
У землю закопав,
Напис написав:
“У попа був собака, він її любив
Вона з'їла шматок м'яса, він її убив
У землю закопав,
Напис написав:
... (лапки цитат ніколи не закриються)

Загальновідомий приклад рекурсивного зображення – предмет між двома дзеркалами – у кожному з них видно нескінченний ряд відображень.

Окрім того, поняття рекурсії добре ілюструється матрьошками. Це означає, що рекурсія дозволяє звести складне завдання до розв’язку його зменшених копій. Послідовно зменшуючи розмір цих завдань, процес рекурсії в решті решт приведе до відомого розв’язку, використовуючи який можна легко здобути розв’язок початкового завдання.

Більш серйозні приклади рекурсії можна віднайти у математиці: рекурентні співвідношення визначають певний елемент послідовності через один чи кілька попередніх. Наприклад, числа Фібоначчі :

`F(n) = F(n-1) + F(n-2)`, де `F(0)=1`, `F(1)=1`.

Якщо розглядати цю послідовність, розпочинаючи від молодших членів до старших, спосіб її побудови задається циклічним алгоритмом, а якщо, навпаки, – від заданого $n=n_0$, то спосіб визначення цього елемента через попередні буде рекурсивним.

Вочевидь, що рекурсія не може бути безумовною, бо у такому разі вона стає нескінченною. Про це свідчить принаймні наведена вище лічилка. **Рекурсія повинна мати всередині себе умову завершення**, за якою наступний крок рекурсії вже не виконуватиметься.

Рекурсивні функції лише на перший погляд схожі на звичайні фрагменти програм. Щоб відчути специфіку рекурсивної функції, варто простежити за текстом програми перебіг її виконування. У звичайній програмі будемо йти ланцюжком викликів функцій, але жодного разу повторно не увійдемо до того самого фрагменту, допоки з нього не вийшли. Можна стверджувати, що перебіг виконування програми "лягає" однозначно на текст програми. Інша річ – рекурсія. Якщо спробувати відстежити за текстом програми перебіг її виконування, то дістанемось такої ситуації: увійшовши до рекурсивної функції, "рухаємося" її текстом доти, допоки не зустрінемо її виклик (можливо, з іншими параметрами), після чого знову розпочнемо виконувати ту ж саму функцію спочатку. При цьому слід зазначити найважливішу властивість рекурсивної функції: її перший виклик ще не завершився. Чисто зовні складається враження, що текст функції відтворюється (копіюється) щоразу, коли функція сама себе викликає. Насправді цей ефект відтворюється в комп'ютері. Однак копіюється при цьому не весь текст функції (не вся функція), а лише її частини, пов'язані з локальними даними (формальні, фактичні параметри, локальні змінні й точка повертання). Алгоритмічна частина (оператори, вирази) рекурсивної функції й глобальні змінні не змінюються, тому вони присутні в пам'яті комп'ютера в єдиному екземплярі.

Кожний рекурсивний виклик породжує новий "екземпляр" формальних параметрів і локальних змінних, причому старий "екземпляр" не знищується, а зберігається у стеку на засаді вкладеності як "матрьошки". Тут має місце єдиний випадок, коли одному імені змінної в перебігу роботи програми відповідають кілька її екземплярів. Відбувається це в такій послідовності:

- ✓ у стеку резервується місце для формальних параметрів, куди записуються значення фактичних параметрів. Зазвичай це виконується в порядку, зворотному до їхнього місця у списку;
- ✓ при виклику функції до стека записується точка повертання – адреса тієї частини програми, де міститься виклик функції;
- ✓ на початку тіла функції у стеку резервується місце для локальних (автоматичних) змінних.

Зазначені змінні створюють групу (фрейм стека). Стек "пам'ятає історію" рекурсивних викликів у вигляді послідовності (ланцюжка) таких фреймів. Програма у кожний конкретний момент працює з останнім викликом і з останнім фреймом. По завершенні рекурсії програма повертається до попередньої версії рекурсивної функції й до попереднього фрейму у стеку.

Послідовність рекурсивних викликів можна прокоментувати приблизно такою фразою: "Функція `F` виконує ... і викликає `F`, яка виконує ... і викликає `F`...".

Класичним прикладом рекурсивної функції є обчислення факторіала (це не означає, що факторіал треба обчислювати лише в такий спосіб). Для того щоб визначити факторіал числа `n`, слід помножити `n` на факторіал числа (`n-1`):

```
n! = n*(n-1)! // Це так зване рекурентне співвідношення.
```

Відомо також, що `0!=1` й `1!=1`, тобто умовою зупинки рекурсії буде: якщо `n=0` чи `n=1`, то факторіал дорівнює `1`.

Дамо повне рекурсивне визначення факторіала:

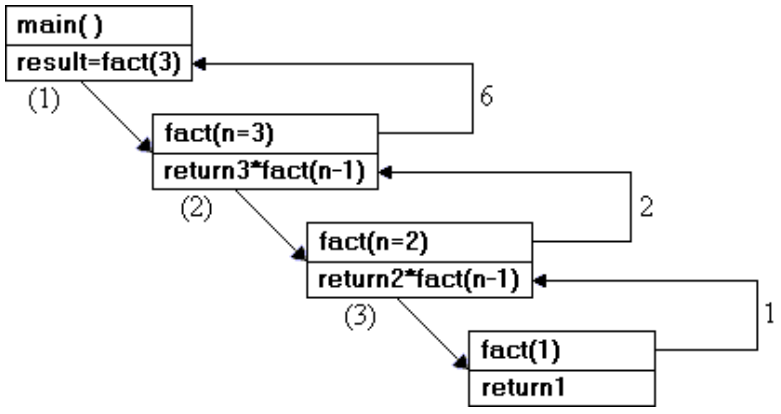
$$n! = \begin{cases} (n-1)! \cdot n, & n > 0; \\ 1, & n = 0. \end{cases}$$

Отже, рекурсивна функція обчислення факторіала числа `n` виглядатиме так:

```
long fact (long n)
{ if (n==0 || n==1) return 1;
  return (n*fact(n-1));     }
```

Якщо у функції `main()` зустрінеться виклик функції `fact(3)`, при виконуванні цієї функції буде викликано функцію `fact(2)`, яка своєю чергою викличе `fact(1)`. Для останньої спрацює умова зупинки рекурсії (`n==1`) і у функцію `fact(2)` буде повернуто значення `1` та обчислено значення `2` (`2!=2`), яке своєю чергою буде повернуто до функції `fact(3)` і буде використано для обчислення `3!`.

Отже, програма має 3 рекурсивні виклики. Занурившись на три рівні рекурсії, програма дісталася "глухого кута" – граничної точки, після чого вона зворотним ланцюжком має піднятися на ті самі три рівні. В результаті кожного рекурсивного виклику функція зберігає своє значення змінної `n`, оскільки до третього виклику у неї буде вже три окремих змінних з ім'ям `n`, при цьому кожна має власне, відмінне від інших, значення, однак тіло функції буде присутнім в пам'яті в єдиному екземплярі. Цей ланцюжок рекурсивних викликів можна зобразити в такий спосіб:



Породження усіх нових копій рекурсивної функції до виходу на граничну умову називається *рекурсивним спуском*. Максимальна кількість копій рекурсивної функції, яке водночас може міститися у пам'яті комп'ютера, називається *глибиною рекурсії*. У разі відсутності граничної умови необмежене зростання кількості таких копій призведе до аварійного завершення програми за рахунок переповнення стека. Завершення роботи рекурсивних функцій, аж до самої першої, яка ініціювала рекурсивні виклики, називається *рекурсивним підйомом*.

При створюванні рекурсивної функції завжди слід добре поміркувати, чи ефективно є застосовувати рекурсивний алгоритм, оскільки рекурсія з надто великою кількістю вкладених викликів може вичерпати ресурси оперативної пам'яті, що спричинить збій у роботі операційної системи.

Розглянемо алгоритм нераціонального використання рекурсії на прикладі обчислення чисел Фібоначчі – числового ряду, в якому кожен наступний член є сумою двох попередніх: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 і т.д. Італійський математик середньовіччя Леонардо Фібоначчі відкрив цю числову послідовність, вивчаючи піраміду Хеопса у Гізі. До речі, ці числа пов'язані поміж собою цікавими співвідношеннями. Наприклад, кожне число є приблизно в 1.618 разів більше попереднього (цю пропорцію називають золотою), у 2.618 рази більше того, яке розташоване через одне число від нього і в 4.236 разів більше числа, розміщеного двома числами раніш, а кожне попереднє число складає приблизно 0.618 від наступного. Поширеність співвідношень Фібоначчі у житті просто вражає. Принцип золотого перетину – найвищий прояв структурної й функціональної довершеності цілого та його частин у мистецтві, науці, техніці й природі. Закономірності "золотої" симетрії проявляються в енергетичних переходах елементарних часток, у будові деяких хімічних сполук, у планетарних і космічних системах, у генних структурах живих організмів, у принципах формотворення рослин тощо. Ці закономірності, виявлено як у будові окремих органів людини, так і тіла у цілому, а також вони проявляються у біоритмах і функціонуванні головного мозку і зорового сприйняття.

Ітераційна функція обчислення чисел Фібоначчі має вигляд

```
long fib1(int n)
{ long a=0, b=1, c=n;
  for(int i=2; i<=n; i++)
   { c=a+b;   a=b;     b=c;   }
  return c;
}
```

Рекурсивна функція матиме вигляд

```
long fib2(int n)
{ if(n==0 || n==1) return n;
  else return fib2(n-1)+fib2(n-2); }
```

або

```
long fib3(int n)
{ return (n>1) ? fib3(n-1)+fib3(n-2) : n; }
```

Така рекурсивна функція є придатна для використання лише для невеликих значень n, оскільки кожне звертання призводить до двох подальших викликів, причому викликів з одним і тим самим значенням параметра, тобто кількість викликів зростає експоненціально. Отже, у цьому завданні доцільніше застосовувати ітераційну функцію **fib1**.

Рекурсивні функції найчастіше застосовують для компактної реалізації рекурсивних алгоритмів, а також для роботи зі структурами даних, описаними рекурсивно, наприклад з бінарними деревами. Кожну рекурсивну функцію можна реалізувати без застосування рекурсії. Достоїнством рекурсії є компактний

запис, а недоліками – витрата часу й пам'яті на повторні виклики функції й передавання їй копій параметрів і, головне, небезпека переповнення стека.

Приклад 8.13 Увести ціле число і вивести всі цифри, які зображують це число.

Текст консольної програми:

```
void cnum (int n)
{ int a=10;
  if (n==0) return;
  else  { cnum(n/a);   cout << n%a<<endl;  }
}
void main()
{ int n;
  cout << "Увести ціле число: ";   cin >> n;
  cnum(n);
  getch(); return 0;
}
```

Результати виконання програми:

```
Увести ціле число: 1234
1
2
3
4
```

Приклад 8.14 Створити рекурсивну функцію піднесення числа до степеня. Аргументами цієї функції мають бути певне дійсне число, яке буде підноситись до степеня, і, безпосередньо, показник степеня цілого типу.

Тексти функції та її виклику в основній програмі:

```
#include <iostream.h>
#include <conio.h>
double power(double x, int st)
{ if(st<=0) return 1;
  else  return(x*power(x,st-1));
}
void main()
{ double x; int st;
  cout<<"Увести число x= ";  cin>>x;
  cout<<"Увести показник степеня st= "; cin>>st;
  cout<<"x ^ st= " << power(x, st);
  getch();  return 0;
}
```

Результати виконання програми:

```
Увести число x= 2.5
Увести показник степеня st= 2
x ^ st= 6.25
```

Приклад 8.15 Написати рекурсивну функцію, яка обчислює мінімальний елемент масиву.

Розв'язок. Створимо рекурсивну функцію `minimum()`, яка повертатиме індекс мінімального елемента. Вона матиме три параметри: 1) вказівник на початок масиву `mas`, 2) розмір масиву `n` і 3) індекс елемента, з якого розпочинається підмасив. Цей підмасив рекурсивно зменшується, допоки його розмір не становитиме 1. Коли у ньому залишиться один елемент (`k==n-1`), – повертається значення `k`. На наступному кроці рекурсивного повертання порівнюються `mas[k]` (останній елемент) та `mas[a]` (передостанній елемент) та повертається індекс меншого з них. Продовжуючи, на кожному кроці порівнюються мінімум, здобутий на попередніх кроках, і `mas[a]` (поточний елемент).

Тексти функції та її виклику в основній програмі:

```
int minimum(int *mas, int n, int k)
{ if(k == n-1) return k;
  int a = minimum(mas,n,k+1);// Виклик для підмасиву розміром на 1 менше
  if(mas[a]<mas[k]) return a;   else return k;
}
void main()
{ int i, n, min;
  cout<<"Уведіть розмір масиву: ";   cin>>n;
  int *mas = new int[n];
  for (i=0;i<n;i++)
   { cout<<"Уведіть mas["<<i<<"]: ";   cin>>mas[i];  }
  cout<<endl;
  min = minimum(mas,n,0); // Виклик для всього масиву (k=0).
  cout<<"Мінімум: mas["<<min<<"] = "<<mas[min]<<endl;
  getch();
  return 0;
}
```

Приклад 8.16 Написати рекурсивну функцію, яка обчислює суму квадратних коренів натуральних чисел від `1` до `n`.

Тексти функції та її виклику в основній програмі:

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
double sumkor(int a)
{ if (a==1) return 1;
  else  return (sqrt(a) + sumkor(a-1));
}
void main()
{ int n;
  cout<<"n= ";   cin>>n;
  cout<<sumkor(n);
  getch(); return 0;
}
```

Приклад 8.17 Увести масив з 10-ти дійсних чисел і створити рекурсивну функцію, яка записує масив у зворотній послідовності .

Розв'язок. Функція `invertItem()`змінює місцями елементи з індексами `i` та `j`. Спочатку це перший та останній елементи масиву, потім індекс `i` збільшується, а `j` – зменшується – і функція викликає себе для обміну другого й передостаннього елементів. Ланцюжок викликів триває, допоки `i < j`.

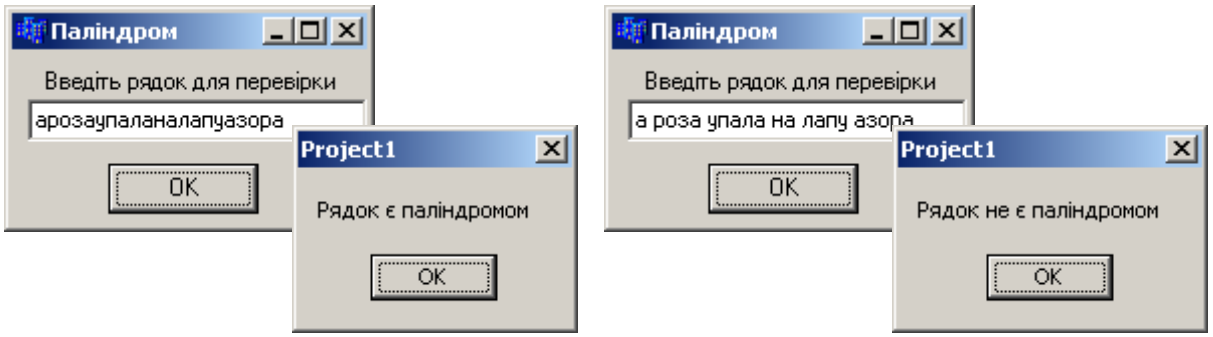
Тексти функції та її виклику в основній програмі:

```
void invertItem(double *a, int i, int j)
{ double t=a[i];  a[i]=a[j];  a[j]=t;
  i++; j--;
  if (i<j) invertItem(a, i, j);
}
void main()
{ double a[10];  int i;
  cout<<"Увести 10 дійсних чисел:"<<endl;
  for (i=0; i<10; i++)  cin>>a[i];
  invertItem(a, 0, 9);
  for (i=0; i<10; i++)  cout<<a[i]<<" ";
  getch(); return 0;
}
```

Результати виконання програми:

```
Увести 10 дійсних чисел:
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

Приклад 8.18 Визначити, чи є введений рядок паліндромом, тобто таким, що читається однаково зліва направо і справа наліво.



Тексти функції та її виклику в основній програмі:

```
int pal(char *s)
{int len;  char *s1=new char[strlen(s)+1];
 if (strlen(s)<=1) return 1;
 else
  { len=(s[0]==s[strlen(s)-1]);
    strncpy(s1, s+1, strlen(s)-2);
    s1[strlen(s)-2]='\0';
    return len && pal(s1);
  }
```

```
delete []s1;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString ss = Edit1->Text;
  char *s = new char [ss.Length()];
  strcpy(s,ss.c_str());
  int p = pal(s);
  if(p)ShowMessage("Рядок є паліндромом");
  else ShowMessage("Рядок не є паліндромом");
 }
```

Зверніть увагу на команду:

```
l=(s[0]==s[strlen(s)-1]);
```

У ній спочатку виконується порівняння `s[0]==s[strlen(s)-1]`, а потім його результат (`true` чи `false`) перетворюється на ціле число (відповідно 1 чи 0) і присвоюється змінній `l`.

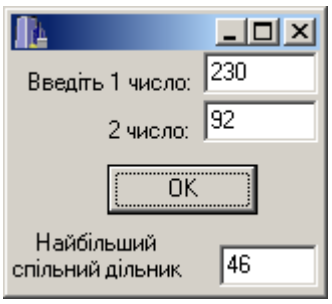
Запис

```
return l && pal(s1);
```

означає, що після обчислення **pal**`(s1)` обчислюється результат операції логічного множення `&&` і цей результат повертається у точку виклику функції.

Приклад 8.19 Увести два числа й обчислити найбільший спільний дільник (НСД) двох цілих чисел, застосовуючи рекурсивний алгоритм Евкліда.

Розв'язок. Нерекурсивна реалізація алгоритму Евкліда обчислення НДС двох цілих чисел наведена у п. 4.4.4.



Тексти функції та основної програми:

```
int nod(int a, int b)
{ int  c;
  // За умови b>a параметри змінюються місцями
  if(b > a) c = nod(b, a);
  else
    if(b == 0) c = a;
    else   c = nod(b, a%b); // Тут a%b – залишок від ділення a на b
  return c;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int a, b;
  a = StrToInt(Edit1->Text);
  b = StrToInt(Edit2->Text);
  Edit3->Text = IntToStr(nod(a, b));
}
```

Приклад 8.20 Увести масив з 12-ти цілих чисел і упорядкувати їх за зростанням методом швидкого сортування за допомогою рекурсивної функції.

Розв'язок. Швидке сортування (англ. quicksort), часто зване qsort на ім'я реалізації в стандартній бібліотеці мови С – широко відомий алгоритм сорту-

вання, розроблений англійським інформатиком Чарльзом Хоаром. Це один з швидких відомих універсальних алгоритмів сортування масивів.

Коротко описуючи цей алгоритм, треба обрати опорний елемент, порівняти решту елементів з опорним, на підставі порівняння розбити множину елементів масиву на три – "менші опорного", "рівні" і "більші", розташувати їх у порядку менші-рівні-більші, після чого повторити рекурсивно ці дії для "менших" і "більших".

Зазвичай опорним елементом вибирають середній елемент, хоча з точки зору коректності алгоритму це не є суттєвим. Операція реорганізації масиву полягає в тому, щоб усі елементи, які є менші чи рівні опорному елементові, опинилися ліворуч від нього, а усі елементи, більші за опорний, – праворуч від нього. Для цього у функції `QuickSort()` організовано таку послідовність дій:

1) два індекси – `L` та `R`, прирівнюються до мінімального та максимального індексів розділеного масиву відповідно: `i=L; j=R;`

2) обчислюється індекс опорного елемента `n=(L+R)/2;`

3) індекс `i` послідовно збільшується допоки `i`-й елемент не перевищить опорний: `while (a[i] < x) i++;`

4) індекс `j` послідовно зменшується допоки `j`-й елемент не виявиться менше чи рівний опорному: `while (x < a[j]) j--;`

5) якщо `i<j` – знайдену пару елементів треба поміняти місцями й продовжити операцію поділу з тих значень `i` та `j`, які були досягнені;

6) чергуючи зменшення `j` та збільшення `i`, порівняння та необхідні обміни, віднаходимо середину масиву, коли `i=j`, тобто операцію поділу завершено, обидва індекси вказують на опорний елемент. У результаті елементи масиву виявляються розділеними на дві частини так, що всі елементи ліворуч є менші за опорний елемент, а всі елементи праворуч – більше за нього;

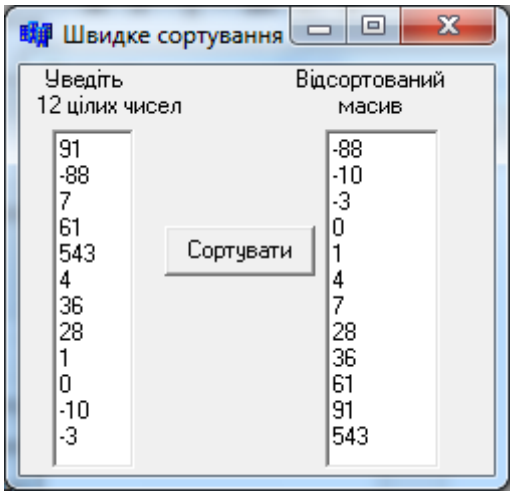
7) рекурсивно впорядковуємо підмасиви, що лежать ліворуч і праворуч від опорного елемента, застосовуючи розглянутий алгоритм. Оскільки на кожній ітерації (на кожному наступному рівні рекурсії) довжина оброблюваного відрізка масиву зменшується, щонайменше, на одиницю, термінальна гілка рекурсії буде досягнена завжди й опрацювання гарантовано завершиться.

Цікаво, що Хоар розробив цей метод для машинного перекладу: справа в тім, що в той час словник зберігався на магнітній стрічці, і якщо впорядкувати всі слова в тексті, їх переклади можна дістати за один прогін стрічки.

Тексти функції та її виклику в основній програмі:

```
void QuickSort(int a[], int L, int R)
                                        // L – ліва та R – права межі сортування
{ int i=L, j=R, x, y;
   int n=(L+R)/2;
    x = a[n];
  do
  { while (a[i] < x) i++;
    while (x < a[j]) j--;
```

```
    if ( i <= j )
    { y=a[i]; a[i]=a[j]; a[j]=y;
      i++; j--;
  } }  while (i <= j);
  if (L < j)  QuickSort(a, L, j);
  if (i < R)  QuickSort(a, i, R);
  return;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i,  a[12];
  for (i=0; i<12; i++) a[i]=StrToInt(Memo1->Lines->Strings[i]);
  QuickSort(a, 0, 11 ); // На вході ліва і права межа сортування
  Memo2->Clear();
  for (i=0; i<12; i++) Memo2->Lines->Add(IntToStr(a[i]));
}
```



8.6 Перевантаження функцій

Часто буває зручно, щоб функції, які реалізовують однаковий алгоритм для різних типів даних, мали однакове ім'я. Якщо це ім'я несе певну інформацію, це робить програму більш зрозумілою, оскільки для кожної дії треба пам'ятати лише одне ім'я. Використовування кількох функцій з однаковим ім'ям, але з різними типами параметрів називається *перевантаженням* функцій.

Компілятор визначає, яку саме функцію слід викликати, за типом фактичних параметрів. Цей процес називається *дозволом перевантаження* (переклад з англійського слова "resolution"). Тип значення, яке повертає функція, у дозволі не бере участі. Механізм дозволу ґрунтується на доволі складному наборі правил, зміст яких зводиться до того, щоб використати функцію з найбільш придатними аргументами й видати повідомлення, якщо така не віднайдеться. Мета перевантаження – зробити так, щоб функції з однаковим ім'ям виконувалися по-різному (і за можливість повертали різні значення) при звертанні до них з різними за типами й кількістю параметрами. За приклад наведемо чотири варіанти функції для визначення максимального значення:

```
int max(int, int); // Повертає найбільше з двох цілих
char* max(char*, char*);//Повертає рядок максимальної довжини
```

```
// Повертає найбільше значення з першого параметра і довжини рядка, переданого
// до функції в якості другого параметра
int max(int, char*);
// Повертає найбільше значення з першого параметра і довжини рядка, переданого
// до функції в якості першого параметра
int max(char*, int);
void f(int a, int b, char* c, char* d)
{  cout << max(a,b) << max(c,d) << max(a,c) << max(c,b); }
```

Усі чотири функції мають однакове ім'я (`max`) й відрізняються типами параметрів (тому вважаються за різні функції). За викликання функції `max()` компілятор обирає відповідно до типу фактичних параметрів варіант функції (у наведеному прикладі буде послідовно викликано всі чотири варіанти функції). Якщо точної відповідності не віднайдено, виконуються перетворювання порядкових типів відповідно до загальних правил, наприклад `bool` й `char` до `int`, `float` до `double` тощо. Далі виконуються стандартні перетворювання типів, наприклад `int` до `double`, вказівників до `void*` тощо. Наступним кроком є виконання перетворювань типів, заданих користувачем, а також пошук відповідностей за рахунок змінної кількості аргументів функції. Якщо відповідність на тому самому етапі може бути набута у понад один спосіб, виклик вважається за неоднозначний і видається повідомлення про помилку.

Неоднозначність може виникнути за:

- ✓ перетворення типу;
- ✓ використання параметрів-посилань;
- ✓ використання аргументів за замовчуванням.

Приклад неоднозначності за перетворення типу:

```
#include <iostream.h>
int f(int a){return a;}
int f(int a, int b=1){return a*b;}
void main ()
{ cout << f(10,2);   // Викликається f(int, int)
  cout << f(10);   } // Неоднозначність – що викликати: f(int, int) чи f(int)?
```

Слід відрізняти перевантаження функцій і редекларацію. Якщо функції з однаковим ім'ям мають однакові параметри і типи результату, вважається, що друге визначення розглядатиметься як редекларація (повторне визначення). Якщо параметри функцій з однаковим ім'ям збігаються і функцій відрізняються лише типом значення, яке повертається, це призведе до помилки.

Правила опису перевантажених функцій:

- ✓ перевантажені функції мають перебувати в одній області видимості, інакше станеться приховування аналогічне до однакових імен змінних у вкладених блоках;
- ✓ перевантажені функції можуть мати параметри за замовчуванням, при цьому значення одного й того самого параметра у різних функціях мають збігатися. У різних варіантах перевантажених функцій може бути різна кількість параметрів за замовчуванням.

Функції не можуть бути перевантажені, якщо опис їхніх параметрів відрізняється лише специфікатором `const` чи використанням посилання, наприклад `int` та `const int` чи `int` та `int&`.

Приклад 8.21 Написати дві функції для обчислення суми сімох елементів цілого й дійсного масивів відповідно. Використати ці функції для обчислення сум двох масивів.

Тексти функцій та їх викликів в основній програмі:

```
int adder(int iarray[7])
{ int isum = 0;
  for(int i=0; i<7; i++)
    isum += iarray[i];
  return(isum);
}
float adder (float farray[7])
{ float fsum = 0;
  for(int i= 0; i < 7; i++)
    fsum += farray[i];
  return (fsum) ;
}
void main() {
  int iarray[7] = {5,1,6,20,15,0,12};
  float farray[7] = {3.3,5.2,0.05,1.49,3.12345,31.0,2.007};
  int isum; float fsum;
  isum = adder (iarray); // Виклик для масиву цілих чисел
  fsum = adder (farray); // Виклик для масиву дійсних чисел
  cout << "Сума масиву цілих чисел дорівнює " << isum << "\n";
  cout << "Сума масиву дійсних чисел дорівнює "<< fsum << "\n";
  getch ();
}
```

Результати виконання програми:

```
Сума масиву цілих чисел дорівнює 59
Сума масиву дійсних чисел дорівнює 46.1705
```

Приклад 8.22 Написати три функції для обчислення периметра трикутника, чотирикутника й п’ятикутника відповідно.

Розв’язок. Завдання відрізняється від попереднього тим, що тип параметрів є однаковий, а кількість параметрів – різна. При викликанні функції обиратиметься той варіант, для якого збігається кількість фактичних і формальних параметрів.

Тексти функції та її виклику в основній програмі:

```
// Визначення всіх потрібних версій функції
// Версія функції для трикутника
int Perimetr(int Len1, int Len2, int Len3)
{ return Len1 + Len2 + Len3;  }
```

```
// Версія функції для чотирикутника
int Perimetr(int Len1, int Len2, int Len3, int Len4)
{ return Len1 + Len2 + Len3 + Len4; }
// Версія функції для п'ятикутника
int Perimetr(int Len1, int Len2, int Len3, int Len4, int Len5)
{ return Len1 + Len2 + Len3 + Len4 + Len5; }
int main()
{ int AB = 5, BC = 3,  CA = 7;
  int CD = 8, DA = 11, DE = 2, EA = 9;
  cout << "Perimetr ABC = " << Perimetr(AB, BC, CA) << "\n";
  cout << "Perimetr ABCD = " << Perimetr(AB,BC,CD,DA) << "\n";
  cout << "Perimetr ABCDE = " << Perimetr(AB, BC, CD, DE, EA);
  getch();
}
```

Результати виконання програми:

```
Perimetr ABC = 15
Perimetr ABCD = 27
Perimetr ABCDE = 27
```

Питання та завдання для самоконтролю

1) У чому полягає різниця поміж оголошенням прототипу й визначенням функції?

2) Які дії виконує у функції оператор `return`?

3) Для яких функцій при їхньому визначенні не використовується ключове слово `return`?

4) Який тип має функція, яка не повертає значення?

5) Яку інформацію про функцію надає такий її заголовок:

```
int fun(double x)
```

6) В який спосіб можна передавати з функції декілька результатів?

7) Які параметри функції називають формальними, а які – фактичними?

8) Чи повинні збігатися імена відповідних формальних і фактичних параметрів функції?

9) Що таке локальна змінна?

10) Поясніть різницю поміж локальними та глобальними змінними.

11) В яких випадках слід використовувати глобальні змінні?

12) Що буде виведено на екран після виконання такої програми?

```
#include <iostream.h>
void swap(int i, int j)
{ int temp = i;  i = j;  j = temp;}
int main()
{ int a = 10, b = 3;
  swap (a, b);
  cout << "a = " << a << " and b = " << b;
  return 0;  }
```

13) Що таке область видимості змінної?

14) Чи можна у C++ оголошувати вкладені функції?

15) Які є способи передавання параметрів до функції?

16) Що буде виведено на екран після виконання програми у завданні 12, якщо замінити функцію `swap()` на таку:

```
void swap(int &i, int &j)
{ int temp = i;  i = j;  j = temp; }
```

17) Що таке перевантаження функцій?

18) Яка проблема виникне з такими перевантаженими функціями?

```
void inc(int &i)
{  i = i + 1; }
void inc(int &i, int diff = 1)
{  i += diff; }
```

19) Знайдіть помилку у функції

```
double volume(double length, double width = 1, double height)
{  return length * width * height;   }
```

20) Поясніть способ передавання параметрів до функції

```
void inc (int &i, int diff = 1)
{  i += diff;  }
```

21) Яка помилку припущено у такій програмі? Як її виправити?

```
# include <iostream.h>
int main()
{ double x = 5.2;
  cout  << x << " ^ 2 = " << sqr(x);
  return 0;
}
double sqr( double x)
{ return x * x; }
```

22) Що таке рекурсія?

23) Чи є наступна рекурсивна функція правильною? Поясніть.

```
double F(double x)
{ return F(x-1); }
```

24) Напишіть програму із застосуванням перевантаження функцій обчислення суми двох чисел цілого і дійсного типів відповідно.

Розділ 9

Багатофайлові програми

9.1 Міжфайлова взаємодія

Проект C++ Builder складається з багатьох файлів, але власне програма дотепер містилася в єдиному файлі `Unit1.cpp`. Існує багато ситуацій у програмуванні, коли мати лише один файл програми незручно.

Якщо певний проект розробляє група програмістів, кожний з яких виконує окреме завдання, має сенс розв'язок кожного завдання розташовувати в окремому файлі, а потім долучати всі такі файли до спільного проекту. Такий підхід дозволяє більш оптимально організувати роботу команди програмістів і більш чітко визначити взаємодію частин програми. Відповідно до засад технології програмування, великі програми розбиваються на частини за функціональною направленістю. Один файл може містити програмний код підтримки графічного виведення, другий – математичні розрахунки, третій – код введення-виведення даних файла. При написанні великих програм тримати всі ці частини в одному файлі-програмі складно чи взагалі неможливо.

Інколи виникає потреба у використовуванні в різних програмах однакових типів і функцій. В таких ситуаціях є сенс винести оголошення цих типів і функцій до окремого файла й долучати його за потреби до різних програм. Має сенс створювати окремі бібліотеки класів і розміщувати їх в окремих файлах.

У багатофайлових програмах елементи програми (змінні, константи, функції тощо), які зберігаються у різних файлах, повинні взаємодіяти один з одним. Розглянемо спочатку окремі `cpp`-файли. Зазначимо, що *область видимості* – це частина програми, всередині якої до певної змінної чи іншого елемента програми є доступ (детальніше див. підрозд. 9.5). Приміром, елементи, оголошені всередині функції, мають *локальну область видимості*, тобто доступ до них існує лише з даної функції. Компоненти класу є видимі лише всередині класу (докладніше про класи див. розд. 14), окрім певних окремих випадків.

Елементи програми, оголошені зовні усіх функцій та класів, мають *глобальну область видимості*, тобто доступ до них можливий з кожного місця коду програми. Окрім того, глобальні елементи є видимі і з інших файлів проекту.

Уточнимо різницю поміж *оголошенням* та *визначенням* змінних у багатофайлових проектах. При *оголошенні* задаються тип та ім'я змінної. Це не означає, що для неї одразу ж зарезервується місце у пам'яті. Оголошення лише повідомляє компіляторові, що десь у програмі може зустрітися змінна такого типу з таким ім'ям. Змінна *визначається*, коли для неї зарезервовується місце у пам'яті, яке здатне містити її значення. Більшість оголошень є визначеннями. Єдиним оголошенням простої змінної, яке не є визначенням, є оголошення з використанням зарезервованого слова `extern` (без ініціалізації):

```
int a;            // Оголошення і визначення
extern int b;     // Лише оголошення
```

Глобальну змінну може бути визначено у програмі лише одноразово, незалежно від того, скільки файлів міститься у програмі. Локальні змінні з однаковими іменами може бути визначено у різних областях видимості. Але використовувати змінні з однаковими іменами, навіть у різних блоках програми, не рекомендовано, оскільки це може призвести до помилок.

```
// Файл A
int X; // Визначення змінної X у файлі A
// Файл B
int X; // Так робити не можна: є визначення такої самої змінної X у файлі A
```

Але й наступний запис буде неправильним, оскільки у файлі B змінна X є невідома:

```
// Файл A
int X; // Визначення змінної X у файлі A
// Файл B
X=3;     // Так робити не можна, оскільки змінна X у файлі B є невідома
```

Щоб глобальна змінна, визначена в одному файлі, була видимою у всіх файлах програми, слід оголосити її із зарезервованим словом extern:

```
// Файл A
int X;            // Визначення змінної X у файлі A
// Файл B
extern int X; // Визначення змінної X у файлі B
X=3;
```

У цьому прикладі оголошення змінної X у файлі A зробило її видимою у файлі B. Зарезервоване слово extern означає, що оголошення в такому разі – це лише оголошення, й нічого більше. Воно спонукає компілятор, який кожного моменту часу бачить лише один з файлів, не звертати уваги на те, що змінна X не є визначена у файлі B.

Слід пам’ятати про одне обмеження: змінну не можна ініціалізувати при оголошенні з extern. Вираз

```
extern int X=27;
```

примусить компілятор вважати, що це не є оголошення, а визначення. Тобто він проігнорує слово extern і сприйме оголошення за визначення. Якщо цю змінну в іншому файлі вже визначено, буде видано помилку повторного визначення.

Якщо треба мати дві глобальні змінні з однаковими іменами у двох різних файлах, їх слід визначати із зарезервованим словом static. У такий спосіб область видимості змінної звужується до файла, в якому її визначено. Решта змінних з таким ім’ям можуть в інших файлах використовуватися без обмежень.

```
// Файл A
static int X;       // Визначення: змінна X є видима лише у файлі A
// Файл B
static int X;       // Визначення: змінна X є видима лише у файлі B
```

Хоча тут визначено дві змінні з однаковими іменами, конфлікту це не спричинює. Код, до якого входить звертання до X, звертатиметься лише до змінної, ви-

значеної у даному файлі. У такому разі вважають, що статична змінна має *внутрішнє зв'язування*. Нестатичні глобальні змінні мають *зовнішнє зв'язування*.

У багатофайлових програмах рекомендовано глобальні змінні робити статичними, якщо за логікою роботи доступ до них вимагається не більше ніж з одного файла. Це допоможе запобігти виникненню помилки, пов'язаної з випадковим задаванням того ж самого імені в іншому файлі. Окрім того, текст програми стає більш прозорим – не треба піклуватися про збіг імен у різних файлах.

Звернімо увагу на те, що зарезервоване слово `static` має декілька значень залежно від контексту. У контексті глобальних змінних слово `static` звужує область видимості змінної до одного файла.

Змінна-константа, яка визначається за допомогою `const`, у загальному випадку є невидимою за межами одного файла. В цьому вона є подібна до `static`. Але її можна зробити видимою з кожного файла програми і для визначення, і для оголошення за допомогою слова `extern`:

```
// Файл A
extern const int X=99;  // Визначення X
// Файл B
extern const int X;     // Оголошення X
```

Тут файл `B` матиме доступ до константи `X`, визначеної в `A`. Компілятор розрізнює оголошення й визначення константи за наявністю чи то відсутністю ініціалізації.

Оголошення функції задає її ім'я, тип значення, яке повертає функція, і типи всіх аргументів. *Визначення функції* – це оголошення плюс реалізація функції (код, який міститься всередині фігурних дужок).

Коли компілятор генерує виклик функції, йому не треба знати, в який спосіб вона працює. Все, що потрібно знати, – це її ім'я, тип результату і типи параметрів (аргументів). Усі дані стосовно цього містяться в оголошенні функції. Тому можна легко визначити функцію в одному файлі, а викликати її з іншого. Жодних додаткових дозволів (як-от слова `extern`) не вимагається. Слід лише оголосити функцію в тому файлі, звідки ця функція викликатиметься.

```
// Файл A
int add(int a, int b)        // Визначення функції
{ return a+b; }              // з тілом функції
// Файл B
int add(int, int);           // Оголошення функції (без тіла)
...
int answer=add(2, 3);        // Виклик функції
```

Зарезервоване слово `extern` з функціями не використовується, оскільки компілятор здатен відрізнити оголошення від визначення за наявності чи відсутності тіла функції.

Можна оголосити (не визначити!) функцію чи інший елемент програми кількаразово. Якщо оголошення не суперечать одне одному, компілятор не видасть помилку:

```
// Файл A
int add(int, int);          // Оголошення функції (без тіла);
int add(int, int);          // друге оголошення – помилки немає
```

Подібно до змінних, функції можна створити невидимими для інших файлів. Для цього при їхньому оголошенні використовується одне й те саме слово `static`.

```
// Файл A
static int add(int a, int b)      // Визначення функції
{ return a+b; }
// Файл B
static int add(int a, int b)      // Інша функція
{ return a+b; }
```

Цей код створює дві функції, жодна з них не є видима поза файлом, у якому її визначено.

9.2 Заголовні файли

Заголовні файли мають розширення `h` чи `hpp` і містять заголовки (прототипи) функцій, а також оголошення типів, констант та змінних з ключовим словом `extern`. Перевага заголовних файлів полягає у тому, що, коли створено заголовний файл для одного модуля (одного чи декількох `cpp`-файлів) програми, можна переносити всі зроблені оголошення, які були зроблені, до іншої програми C++. Для цього слід просто залучити заголовний файл за допомогою директиви `#include`. Зазвичай для кожного `cpp`-файла створюється власний заголовний файл з таким самим ім'ям і розширенням `h`. І, навпаки, здебільшого, для кожного заголовного файла створюється `cpp`-файл з реалізацією функцій, оголошених у `h`-файлі.

При залученні до програми заголовного файла його текст (а з ним і текст відповідного `cpp`-файла) автоматично вставляється до програми замість рядка з відповідною директивою `#include`. При компіляції програми всі залучені заголовні файли з файлами їхньої реалізації перекомпільовуються, тому залучення надмірної кількості заголовних файлів уповільнює компіляцію програми.

Заголовні файли можна розділити на стандартні й створювані програмістом. Стандартні заголовні файли зберігаються у спеціальній теці INCLUDE, яка є підтекою теки, в яку було встановлено Borland C++. Імена стандартних заголовних файлів пишуться у кутових дужках "<" і ">", наприклад:

```
#include <math.h>      /* Залучення заголовного файла
                          з математичними функціями */
```

Окрім того, для таких файлів можна не зазначати розширення `.h`. Якщо стандартний заголовний файл розміщено в іншій теці, слід зазначити його місцезнаходження, наприклад, щоб залучити до програми заголовний файл `types.h`, який міститься у теці `INCLUDE\SYS`, слід написати:

```
#include <include\sys\types.h>
```

Якщо cpp-файл і h-файл мають однакові імена, редактор коду Borland C++ Builder дозволяє перемикатися поміж цими файлами. Для цього слід клацнути правою кнопкою миші на імені вкладки, в якій відкрито один з файлів, і в контекстному меню обрати **Open Source / Header File** (Відкрити файл коду / Заголовний файл).

Заголовні файли, створені програмістом, зазвичай розташовують у теці проекту. Імена цих файлів у директиві `#include` пишуться у подвійних лапках і завжди з розширенням `h` або `hpp`, наприклад, `#include "Myfile.h"`.

Для створення нового заголовного файла слід виконати команду **File / New / Unit**. Буде створено новий модуль, який складатиметься з двох файлів: заголовного файла, наприклад з ім'ям `Unit2.h` (рис. 9.1), і відповідного до нього файла реалізації `Unit2.cpp` (рис. 9.2). При зберіганні буде запитано ім'я тільки cpp-файла. Ім'я h-файла буде створено автоматично.

Для використання заголовного файла у проекті його слід додати до проекту командою **Project / Add To Project**. У діалоговому вікні слід обрати ім'я заголовного файла. Окрім того, для використання в інших модулях проекту функцій, типів, констант, оголошених у заголовному файлі, слід залучити заголовний файл до модуля за допомогою директиви `#include` чи то виконати команду **File / Include Unit Hdr** і обрати відповідний файл.

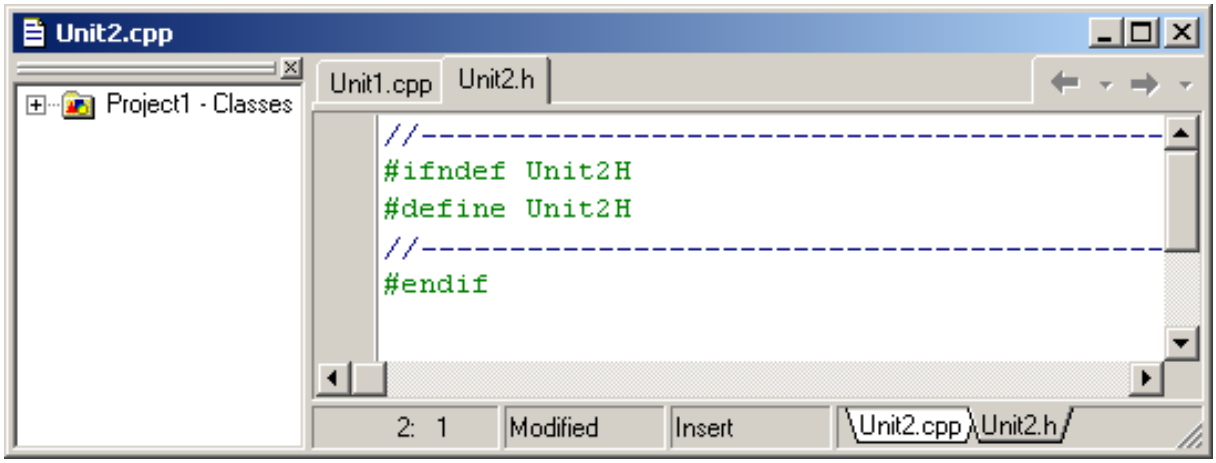


Рис. 9.1. Вікно заголовного файла

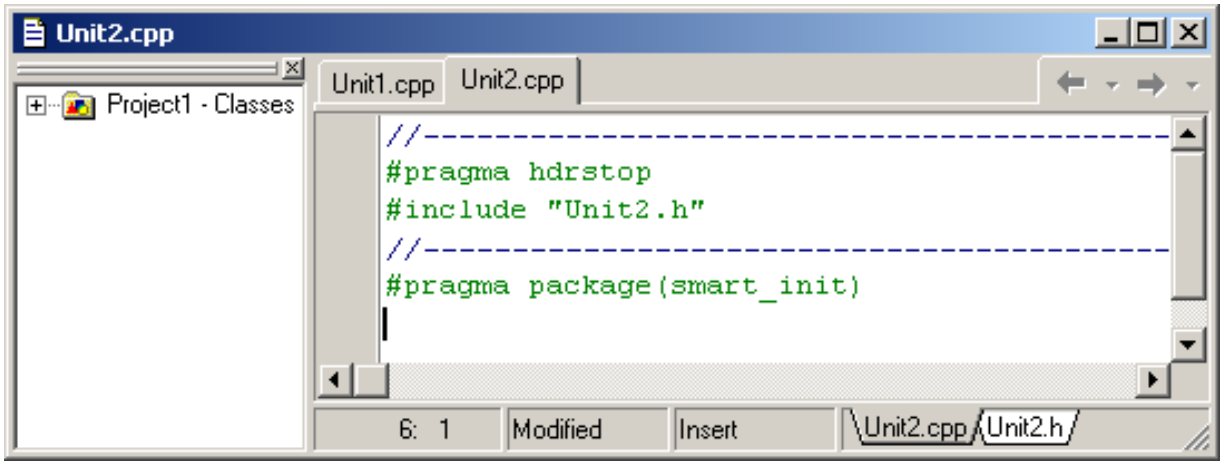


Рис. 9.2. Вікно файла реалізації

Заголовний файл може містити:

- ✓ оголошення типів
```
typedef double arr [14];
```
- ✓ шаблони типів
```
template <class T>
class V { /* ... */ }
```
- ✓ оголошення (прототипи) функцій
```
extern int strlen(const char*);
```
- ✓ визначення функцій-підстановок
```
inline char get()
   { return *p++; }
```
- ✓ оголошення даних
```
extern int a;
```
- ✓ визначення констант
```
const float pi = 3.141593;
```
- ✓ перерахування
```
enum bool { false, true };
```
- ✓ оголошення імен
```
class Matrix;
```
- ✓ команди долучення файлів
```
#include <signal.h>
```
- ✓ макровизначення
```
#define Case break;case
```
- ✓ коментарі
```
/* Перевірка на кінець файла */
```

Перелік того, що саме слід розміщувати в заголовному файлі, не є вимогою мови C++, це є лише порада розумного використовування долучення файлів. З іншого боку, в заголовному файлі ніколи не повинно бути:

- ✓ визначення функцій
```
char get() { return *p++; }
```
- ✓ визначення даних
```
int a;
```
- ✓ визначення складених констант
```
const tb[i] = { /* ... */ };
```

Як вже було зазначено, не можна визначити функцію чи то змінну в заголовному файлі, який використовуватиметься кількома файлами програми. Це призводе до помилок повторних визначень. Проблема виникає й тоді, коли помилково залучають один і той самий заголовний файл двічі, наприклад:

```
// Файл app.cpp з помилкою
#include "headone.h"
#include "headone.h"
```

Припустімо, що є файл з кодом програми `app.cpp` і два заголовних файли `headone.h` та `headtwo.h`. До того ж `headone.h` долучає до себе `headtwo.h`. Якщо є потреба залучити обидва заголовні файли до `app.cpp`, слід уважно стежити за тим, щоб не залучити один і той самий файл двічі. Наприклад:

```
// Файл headtwo.h
extern int X;
// Файл headone.h
#include "headtwo.h"
// Файл app.cpp
#include "headone.h"
#include "headtwo.h"   // Повторне залучення заголовного файла
```

Оскільки директивою `#include` заголовні файли долучаються до програми з усім їхнім умістом, файл `app.cpp` буде таким:

```
// Файл app.cpp
 . . .
int X;        // З headtwo.h через headone.h
 . . .
int X;        // Безпосередньо з headtwo.h
```

Як наслідок, компілятор виведе повідомлення, що змінну X оголошено двічі.

Для попередження помилок повторних включень визначення у заголовному файлі слід розпочинати з директиви препроцесора:

```
#ifndef HEADCOM
```

чи директиви

```
#if!defined(HEADCOM)
```

На місці HEADCOM може бути який завгодно ідентифікатор. Цей вираз говорить про те, що якщо HEADCOM ще не було визначено, то весь текст звідси і до директиви #endif просто вставлятиметься до файла реалізації. В іншому разі (якщо HEADCOM вже було визначено раніш, в чому можна впевнитися за допомогою директиви #define HEADCOM) наступний за #if текст не буде долучено до початкового коду. Оскільки змінну HEADCOM не було визначено до того як ця директива зустрілася вперше, але одразу ж після #if!defined() вона стала визначеною, увесь текст, розміщений поміж #if і #endif, буде долучено один раз, але це буде перший і останній раз. Наприклад:

```
#if!defined (HEADCOM)      // Якщо змінну  HEADCOM ще не визначено,
#define HEADCOM            // визначити її,
extern int X;              // визначити змінну X,
int func(int a, int b);    // визначити функцію  func().
#endif                     // Директива, яка закриває умову.
```

Цей підхід слід використовувати завжди, коли існує можливість випадково долучити заголовний файл до початкового понад одного разу. Раніше використовувалась директива #ifndef, це є те ж саме, що й #if!defined.

Отже, мова C++ є зручна для створення програм, в яких застосовується кілька файлів водночас. Ефективною організаційною стратегією у цьому відношенні є використання заголовного файла для визначення типів користувача і прототипів для функцій, які опрацьовують типи користувача. Визначення функцій слід виносити до окремого файла. В обох файлах – у h-файлі й у cpp-файлі визначається і реалізується тип даних користувача, а також те, в який спосіб його може бути використано. Код, який викликає ці функції, слід розміщувати в іншому cpp-файлі.

Приклад 9.1 Створити заголовний файл, який містить прототипи таких функцій:

1) обчислення елементів матриці a розмірністю 8×8 за формулою

$$a_{i,j}=\begin{cases}\sqrt[3]{\left(\dfrac{i+j+2}{7i+1}+j\right)^4}-3.7*(i+j), & \text{за} \quad \sin(i+j)>0\\ \pi\sin^3(i-j), & \text{за} \quad \sin(i+j)\le 0\end{cases}$$

2) обчислення вектора (одновимірного масиву), елементи якого є сумами елементів відповідних рядків матриці a;

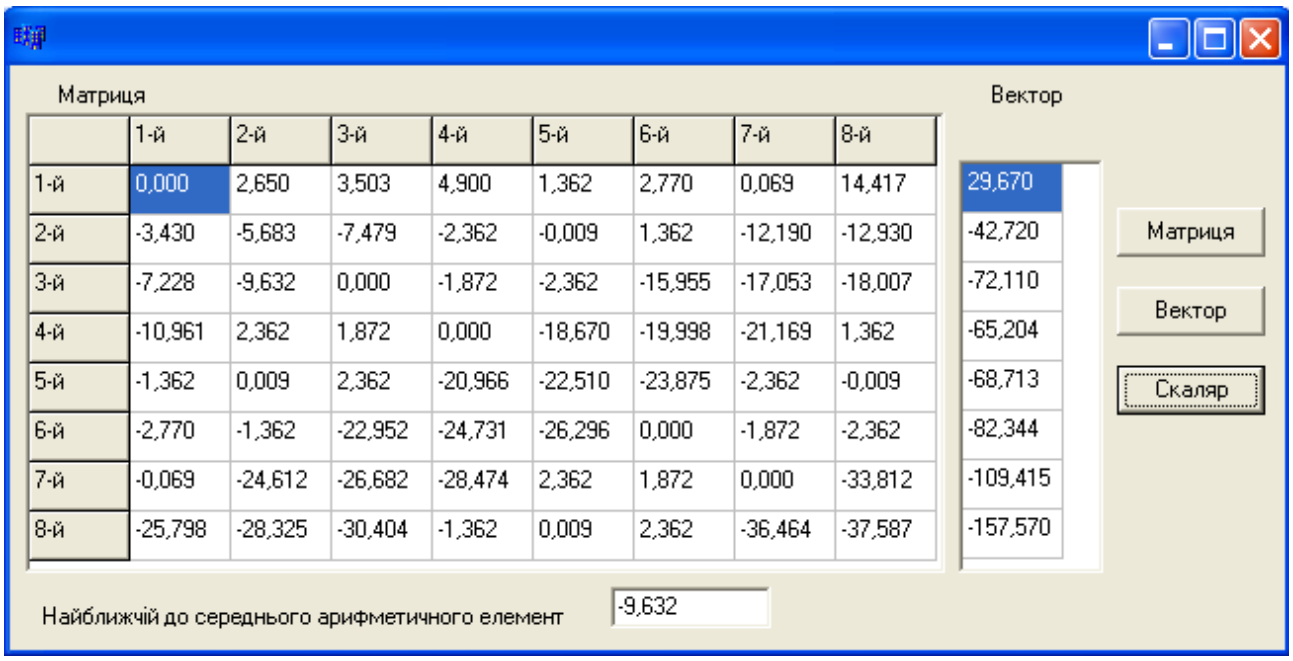
3) пошук елемента матриці a, найбільш наближеного до значення середнього арифметичного елементів матриці a.

Розв'язок. У файлі MyLib.h зробимо потрібні оголошення й заголовки функцій. Використаємо директиву #define, щоб задати константі k значення 8.

```
#ifndef MyLibH
#define MyLibH
#define k 8
// Оголошення типів matr ("матриця") і vekt ("вектор")
typedef double matr[k][k],  vekt[k];
// Оголошення прототипів функцій
void elem_Matr(matr c);
void elem_Vect(matr c, vekt v);
double G(matr c);
#endif
```

У файлі MyLib.cpp пропишемо визначення функцій, прототипи яких розміщено у файлі MyLib.h.

```
#pragma hdrstop
#include <math.h>   // Долучення математичної бібліотеки
#include "MyLib.h"
#pragma package(smart_init)
 // Визначення функції розрахунку елементів матриці
 void elem_Matr(matr c)
{ for(int i=0; i<k; i++)
  for(int j=0; j<k; j++)
    if(sin(i+j)>0)
      c[i][j] = pow((i+j+2.)/(7*i+1)+j,4./3)-3.7*(i+j);
    else
      c[i][j] = M_PI*pow( sin(i-j),3);
}
// Визначення функції  розрахунку елементів вектора
void elem_Vect(matr c, vekt v)
 { for(int i=0; i<k; i++)
   { v[i]=0;
     for(int j=0; j<k; j++) v[i]+=c[i][j];
   }
 }
// Пошук найближчого елемента до середнього арифметичного
double G(matr c)
 { int ni=0, nj=0, i, j;
   double sr=0;
   for(i=0; i<k; i++)
   for(j=0; j<k; j++)
     sr += c[i][j]/pow(k,2);
   for(i=0; i<k; i++)
   for(j=0; j<k; j++)
     if(fabs(sr-c[i][j])<fabs(sr-c[ni][nj]))
      {  ni=i;     nj=j; }
   return c[ni][nj] ;
 }
```



Головний файл проекту Unit1.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include "MyLib.h"    // Долучення власної бібліотеки
#include "Unit1.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//------------ Глобальне оголошення матриці та вектора --------------
matr a;    vekt x;
void __fastcall TForm1::FormCreate(TObject *Sender)
{ // Компоненти Button2 та Button3 стануть невидимими
  Button2->Visible = 0;  Button3->Visible = 0;
  // Заповнення фіксованих комірок компонентів StringGrid
  for(int i=1;i<=k;i++)
   { SG1->Cells[0][i] = IntToStr(i) + "-й";
     SG1->Cells[i][0] = IntToStr(i) + "-й";;
}  }
//------------ Глобальне оголошення матриці та вектора --------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ elem_Matr( a); // Виклик функції обчислення елементів матриці
  for(int i=0; i<k; i++)
  for(int j=0; j<k; j++)
    SG1->Cells[j+1][i+1] = FormatFloat("0.000",a[i][j]);
  Button2->Visible = 1;  Button3->Visible = 1;
}
```

```
//--------------- Виведення елементів вектора ------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{ elem_Vect(a, x); // Виклик підпрограми обчислення елементів вектора
  for(int i=0;i<k;i++)
    SG2->Cells[0][i] = FormatFloat("0.000",x[i]);
}
//--------------- Виведення значення скаляра --------------------
// Виведення значення скаляра
void __fastcall TForm1::Button3Click(TObject *Sender)
{ Edit1->Text = FormatFloat("0.000",G(a));
}
```

Приклад 9.2 Створити заголовний файл, який містить функцію обчислення визначника квадратної матриці.

Розв'язок. Матриця називається *верхньотрикутною*, якщо значення всіх її елементів, розташованих під головною діагоналлю, дорівнюють нулю. Визначник трикутної матриці дорівнює добуткові всіх її діагональних елементів. Отже, для обчислення визначника матриці її слід звести до верхньотрикутної. Для цього треба обчислити коефіцієнти за формулою $f_i = -a_{ik} / a_{ii}$ і додати до елементів *i*-го рядка відповідні елементи *k*-го рядка, помножені на f_i. Процес триває *n*-1 кроків. Отже, значення елементів трикутної матриці визначаються за формулою $a_{ij}^{'} = a_{ij} - \frac{a_{ik} \cdot a_{kj}}{a_{kk}}$. Тут a_{kk} – опорний елемент (він лежить на головній діагоналі); i, j, k – індекси рядків та стовпчиків матриці, n – вимірність матриці. Слід врахувати, що на головній діагоналі матриці не повинно бути нульових елементів, оскільки вони є дільниками у формулі. Тому, якщо опорний елемент є нульовим, слід переставити нижні рядки матриці в такий спосіб, щоб елемент головної діагоналі став ненульовим. При перестановці рядків матриці знак визначника зміниться на протилежний.

Створимо заголовний файл `Det.h`, який міститиме перейменування типу матриця `matr` і дві функції: зведення матриці до верхньотрикутної `triangular()` і обчислення визначника `determinant()`.

У заголовному файлі `Det.h` уведемо потрібні оголошення й заголовки функцій:

```
#ifndef DetH
#define DetH
// Описання типу матриці (вказівник на двовимірний масив)
typedef float** matr;
// Прототипи функції, які можуть бути викликані в програмі
void triangular (matr a, int, int);    /*Зведення матриці
                                          до верхньотрикутної */
float determinant (matr, int);    //Обчислення визначника
#endif
```

Для поставленого завдання щодо обчислення визначника, можна в заголовному файлі оголосити прототип лише функції `determinant()`, яка безпосе-

редньо обчислює визначник. Але функцію `triangular()` може бути використано для розв'язання якихось інших завдань (приміром, при розв'язуванні системи лінійних рівнянь), тому є сенс розмістити її також у заголовному файлі (для підвищення його універсальності) і у разі потреби долучити його до іншої програми.

Файл `Det.cpp` містить визначення функцій, прототипи яких розміщено у файлі `Det.h`. Обидві функції й тип може бути використано у кожній програмі, до якої буде залучено заголовний файл `Det.h`. Окрім того, файл `Det.cpp` містить визначення функції `change()`, яка викликатиметься у функції `determinant()` і буде недоступною в інших файлах, і змінну `znak`, яка буде глобальною для файла `Det.cpp`, але не буде приступною в інших файлах.

Функція `triangular()` має три параметри: матриця `a`, розмір матриці `n` і індекс опорного елемента `k` (індекс обчислюваного рядка). Оскільки матриця завжди передається з доступом для змінювання, функція `triangular()` обчислює нові значення елементів матриці за формулою

$$a_{ij}' = a_{ij} - \frac{a_{ik} \cdot a_{kj}}{a_{kk}},$$

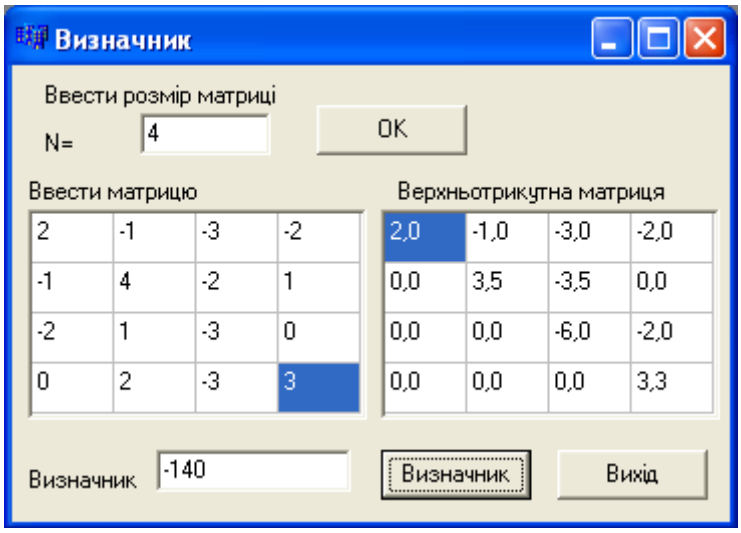
де k – індекс опорного елемента, який є параметром функції.

Функція `change()` має також три параметри: матриця `a`, розмір матриці `n` та індекс рядка, який міняється місцем з будь-яким рядком `k`. Виконується пошук рядка з ненульовим елементом у стовпчику `k`. Якщо такий елемент знайдено, виконується переставляння рядка з індексом `k` і рядка, який містить знайдений ненульовий елемент, і змінюється знак визначника. Після цього виконування функції можна припинити, оскільки подальший пошук не має сенсу.

Функція `determinant()` має два параметри – матрицю `a` та її розмір `n`.

```
#pragma hdrstop
#include "Det.h"
#pragma package(smart_init)
// Глобальне оголошення змінної znak, яка буде зберігати знак визначника
int znak;
// Визначення функції заміни рядків матриці
void change(matr a, int n, int k)
{ for(int j=0;j<n-k;j++) // Переглядання рядків, розміщених нижче рядка
                         // з індексом k.
   if(a[k+j][k]!=0)      // Якщо у стовпчику k знайдено ненульовий елемент,
   { znak=-znak;         // змінити знак визначника на протилежний
     for(int i=0; i<n; i++) // і переставити рядки з індексами k та k+j.
      { float z = a[k][i];
        a[k][i] = a[k+j][i];
        a[k+j][i] = z;
      }                    // Якщо один раз рядки переставлено,
      return;              // продовжувати пошук немає сенсу.
   }
}
```

```
void triangular(matr a, int n, int k)  // Визначення функції triangular()
{ for(int i=k+1; i<n; i++)
  { float koef = a[i][k]/a[k][k];// Обчислення коефіцієнта
    for(int j=0; j<n; j++)       // Обчислення елементів в обраному рядку
      a[i][j] = a[i][j] - a[k][j]*koef;
  }
}
//  Визначення функції обчислення визначника матриці
float determinant(matr a, int n)
{ for(int i=0; i<n; i++)
  { if(a[i][i]==0)      // Якщо опорний елемент дорівнює нулю,
      change(a, n, i); // викликати функцію переставляння рядків матриці
    triangular(a, n, i);// Обчислення стовпчика трикутної матриці
  }
  float det=1;
  for(int i=0; i<n; i++)
    det *= a[i][i];       // Перемноження діагональних елементів
  if(znak<0) det=-det;    // Змінення знака визначника
  return det;
}
```



Головний файл проекту Unit1.cpp.

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#include "Det.h"     // Долучення створеного заголовного файла Det.h
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------
```

```
int N;      // Глобальна змінна для розмірності матриці
// Введення розмірності матриці і встановлення розмірів компонентів StringGrid
void __fastcall TForm1::Button1Click(TObject *Sender)
{ if(Edit1->Text=="")
   { ShowMessage("Введіть розмірність матриці"); return; }
  N=StrToInt(Edit1->Text);
  SG1->RowCount=N;    SG1->ColCount=N;
  SG2->RowCount=N;    SG2->ColCount=N;
}
//---------------------------------------------------------
// Обчислення визначника
void __fastcall TForm1::Button2Click(TObject *Sender)
{ if(N==0){ ShowMessage("Введіть розмірність матриці");return;}
  matr a=new float*[N];   // Виділення пам'яті під матрицю
  for(int i=0; i<N; i++)  a[i]= new float[N];
  randomize();
  for(int i=0; i<N; i++)
  for(int j=0; j<N; j++)
   { a[i][j]=random(10)-5;   //Заповнення матриці випадковими числами
     SG1->Cells[j][i]=FloatToStr(a[i][j]);
   }
  Edit2->Text=determinant(a, N);// Виклик функції обчислення визначення
  for(int i=0; i<N; i++)        // Виведення верхньотрикутної матриці
  for(int j=0; j<N; j++)
    SG2->Cells[j][i]=FormatFloat("0.0", a[i][j]);
}
```

Приклад 9.3 Увести масив координат і нарисувати за числами цього масиву ламану. Створити проект, який містить три форми. На основній формі вводиться масив координат. Друга форма повідомляє про умову прикладу. Третя форма – полотно для рисування ламаної.

Розв'язок. На першій формі розмістимо `StringGrid1` (для спрощення перейменуємо його на `SG1`) та кнопки для автоматичного заповнення масиву випадковими числами чи то довільними числами користувачем. Окрім того, перша форма містить кнопки "Про програму", при клацанні на яку з'являється друга форма з умовою завдання, і "Нарисувати ламану", при клацанні на яку відкривається третя форма з нарисованою ламаною.

Перша форма (`Form1`) відповідає `Unit1.cpp`, друга ("Про програму" – `About`) – `Unit2.cpp`, третя (`Form3`) – `Unit3.cpp`.

Для створення форми `AboutBox` слід в меню **File / New / Other** (Файл / Новий / Інше) обрати вкладку `Forms` (Форми) і клацнути двічі на `AboutBox`.

Для створення третьої форми слід в меню **File / New** (Файл / Новий) обрати `Form` (Форма).

Після створення трьох форм проект слід зберегти (**File / Save All**). Якщо проект було збережено раніше, треба долучити нові форми до проекту: **Project / Add to project** і обрати імена `Unit2` і `Unit3`.

Для виклику у початковій програмі другої й третьої форм слід долучити відповідні заголовні файли Unit2.h і Unit3.h до Unit1.h. Для цього можна виконати команду **File / Include Unit Hdr** і обрати потрібні файли або власноруч написати на початку файла Unit1.cpp:

```
#include "Unit2.h"
#include "Unit3.h"
```

Глобальна змінна a оголошена зі словом extern у Unit3.h. Оскільки цей файл долучено до Unit1.cpp, то змінна a в ньому є відома (навпаки зробити не можна). Визначення цієї змінної може бути розміщено як в Unit1.cpp, так і в Unit3.cpp. В даному прикладі визначення масиву a розміщено в Unit1.cpp.

Слід звернути увагу, що для виклику форм використовуються функції Show() та ShowModal(). Різниця поміж ними полягає в тому, що перша просто відкриває ще одну форму, при цьому завжди є можливість перемкнутися до першої форми. Друга відкриває форму, і перемкнутися до інших форм, допоки відкрито цю форму, є неможливо.

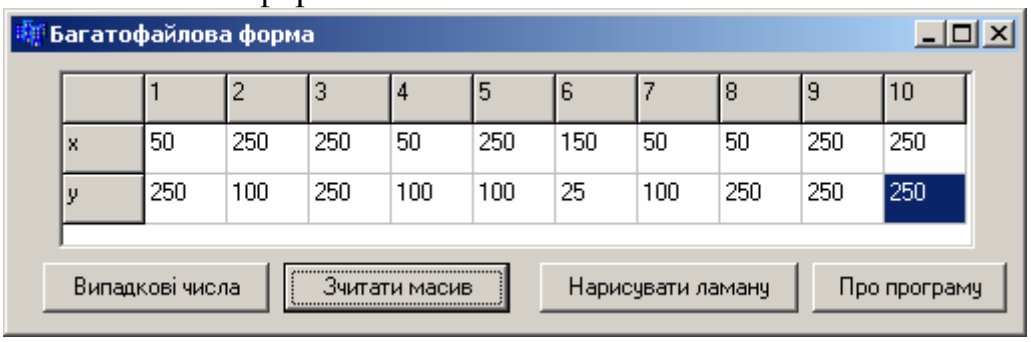
Масив a заповнюється значеннями в Unit1, а використовується – в Unit3.

Файл Unit1.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#include "Unit2.h"   // Долучити файл з другою формою
#include "Unit3.h"   // Долучити файл з третьою формою
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{ SG1->Cells[0][1]="x";
  SG1->Cells[0][2]="y";
  for(int i=1; i<SG1->ColCount; i++)
    SG1->Cells[i][0]=IntToStr(i);
}
//---------------------------------------------------------
int a[2][10];        // Визначення масиву координат (глобальна змінна)
//------------------- Випадкові числа ---------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ randomize();
  for(int i=0; i<10; i++)
   { a[0][i]=random(500)+5;
     a[1][i]=random(400)+5;
     for(int j=0; j<2; j++)
        SG1->Cells[i+1][j+1]=a[j][i];
   }
}
```

```
//------------------ Зчитати масив ----------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{ for(int j=0; j<10; j++)
  for(int i=0; i<2; i++)a[i][j]=StrToInt(SG1->Cells[j+1][i+1]);
}
//------------------- Нарисувати ламану ------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{ Form3->Show();          }      // Показати третю форму
//-------------------- Про програму ----------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{ AboutBox->ShowModal();  }
```

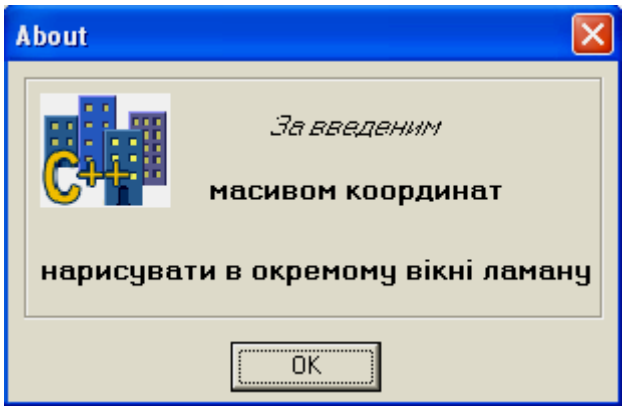
Вікно основної форми:



Файл `Unit2.cpp`:

```
#include <vcl.h>
#pragma hdrstop
#include "Unit2.h"
#pragma resource "*.dfm"
TAboutBox *AboutBox;
//---------------------------------------------------------------
__fastcall TAboutBox::TAboutBox(TComponent* AOwner)
    : TForm(AOwner)
{
}
//---------------------------------------------------------------
void __fastcall TAboutBox::OKButtonClick(TObject *Sender)
{  Close();
}
```

Вікно другої форми “Про програму”:



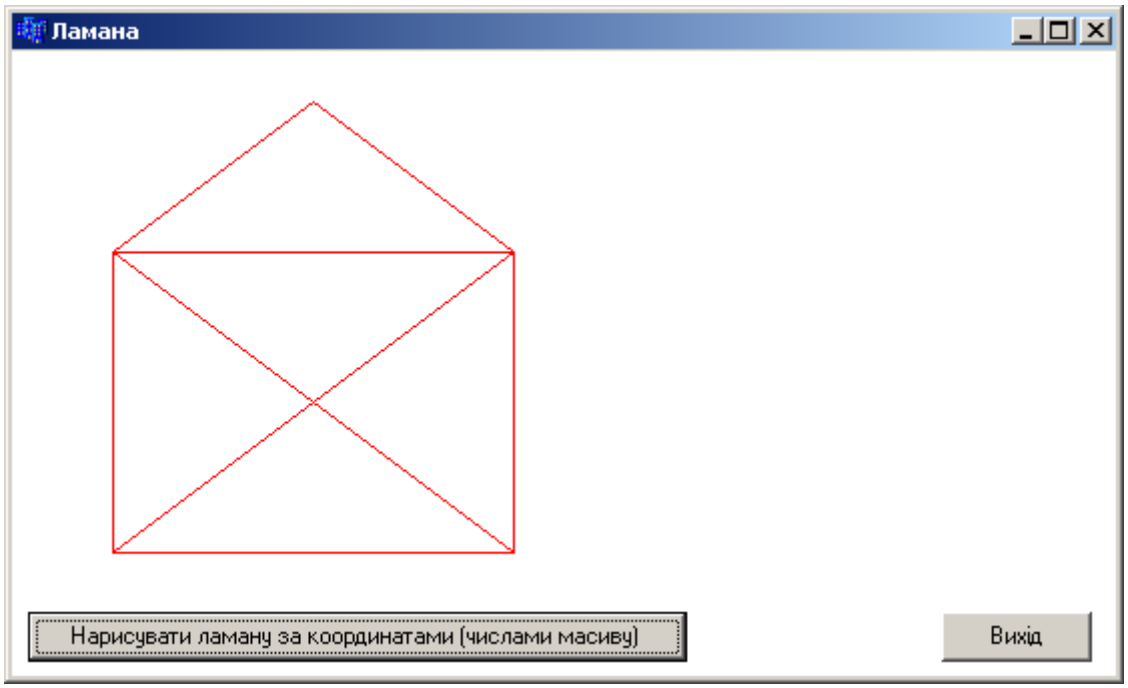
Файл Unit3.h:

```
#ifndef Unit3H
#define Unit3H
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
class TForm3 : public TForm
{ __published: // IDE-managed Components
        TButton *Button1;
        TButton *Button2;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
private:  // User declarations
public:   // User declarations
        __fastcall TForm3(TComponent* Owner);
};
//---------------------------------------------------------------
extern PACKAGE TForm3 *Form3;
//---------------------------------------------------------------
extern int a[2][10];      //Оголошення (не визначення!) масиву координат
#endif
```

Файл Unit3.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include "Unit3.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm3 *Form3;
//---------------------------------------------------------------
__fastcall TForm3::TForm3(TComponent* Owner)
        : TForm(Owner)
{
}
//---------- Нарисувати ламану за координатами (числами масиву) -------
void __fastcall TForm3::Button1Click(TObject *Sender)
{ Canvas->MoveTo(a[0][0],a[1][0]);   // Перейти у першу точку
  Canvas->Pen->Color=clRed;          // і встановити червоний колір.
  for(int i=1; i<10; i++)            // Послідовно у циклі
    Canvas->LineTo(a[0][i],a[1][i]); // з'єднати лініями всі точки.
}
//-------------------- Вихід ---------------------------------
void __fastcall TForm3::Button2Click(TObject *Sender)
{ Close();
}
```

Вікно третьої форми з нарисованою ламаною:



Приклад 9.4 Створити багатофайловий проект для опрацювання раціональних дробів і обчислення найбільшого спільного дільника і найменшого спільного кратного. Надати такі можливості: скорочення дробів, обчислення спільного знаменника двох дробів, порівняння на рівність/нерівність та більше/менше, додавання, множення й ділення дробів.

Розв'язок. Раціональний дріб може бути подано двома цілими числами: чисельником і знаменником. Більшість операцій з раціональними дробами (порівняння, додавання, множення тощо) виконуються за іншими, більш складними, правилами, аніж відповідні операції з цілими числами та десятковими дробами. Тому доречно створити заголовний файл з прототипами функцій, які виконують деякі з зазначених операцій, а потім, за потреби їхнього використання, залучати цей файл до різних проектів.

При опрацюванні раціональних дробів багато операцій потребують обчислення спільного знаменника, який є найменшим спільним кратним знаменників двох дробів. Окрім того, інколи виникає потреба визначити найбільший спільний дільник цілих чисел. Тому буде корисним створити функції, які обчислюють НСД (найбільший спільний дільник) та НСК (найменше спільне кратне). Однак, оскільки ці функції не пов'язані безпосередньо з опрацюванням дробів, є сенс створити окремий заголовний файл, який міститиме оголошення цих двох функцій. Назвемо цей файл `mat.h`. Його вміст буде таким:

```
#ifndef matH
#define matH
int nsd(int, int);          // Найбільший спільник дільник
int nsk(int, int);          // Найменше спільне кратне
#endif
```

Функції **nsd**(int, int) та **nsk**(int, int) далі слід визначити. Для цього переходимо до файла mat.cpp і створюємо текст реалізації оголошених у mat.h функцій:

```
#pragma hdrstop
#include "mat.h"
#include <math.h>
#pragma package(smart_init)
// Рекурсивна функція обчислення найменшого спільного кратного
int nsd( int a, int b)
{ int c;
  if(abs(b) > abs(a))
    c = nsd(b, a);            // За b>a параметри обмінюються місцями
  else
    if(b == 0) c = a;
    else  c = nsd(b, a%b);   // Тут a%b – остача від ділення a на b
  return c;
}
// Нерекурсивна функція обчислення найменшого спільного кратного
int nsk(int a, int b)
{ int x, y, i;              // y – менше, x – більше
  if(abs(a)<abs(b))
    { x=b; y=a; }
  else
    { x=a; y=b; }
  for(i=1; i<=abs(y); i++)
    if((i*x)%y==0) return x*i;
}
```

Функцію обчислення найменшого спільного кратного було розглянуто раніше в підрозд. 8.5 у прикладі 8.19.

Слід звернути увагу на директиви

```
#include "mat.h"
#include <math.h>
```

Перша залучає до програми заголовний файл mat.h, який має міститися у теці з проектом, і стандартну бібліотеку математичних функції math.h, яку розміщено у спеціально відведеній для цього теці разом з Borland C++ Builder.

Слід звернути увагу на те, що створювана програма має опрацьовувати не тільки раціональні дроби, а й цілі числа. Головний файл проекту Unit1.cpp для опрацювання дробів використовуватиме функції, оголошені у файлі ratio.h, а для обчислення НСД та НСК двох цілих чисел – функції, оголошені у mat.h. Це означає, що обидва файли – ratio.h та mat.h – слід долучити до Unit1.cpp. Але заголовний файл mat.h вже буде долученим до ratio.h і тому долучення цього файла до Unit1.cpp викличе повторне долучення, чого слід уникнути. Для цього долучимо цей файл до ratio.h (а не до ratio.cpp) і при долученні ratio.h до Unit1.cpp всі функції, оголошені у mat.h, вже будуть доступні без додаткового долучення його до Unit1.cpp.

Створимо ще один заголовний файл ratio.h, в якому оголосимо чотири функції для опрацювання раціональних дробів:

```
#ifndef ratioH
#define ratioH
#include "mat.h"                  //Долучення заголовного файла mat.h
void sokr(int &a, int &b);        //Скорочення
int znam(int a1, int b1, int a2, int b2);   //Спільний знаменник
bool rivn(int a1, int b1, int a2, int b2); //Порівняння на рівність
bool bilsh(int a1,int b1,int a2,int b2);//Порівняння на більше/менше
#endif
```

Функція sokr() отримує два параметри – чисельник і знаменник раціонального дробу. Вона має можливість змінювати їхні значення, оскільки параметри передаються за посиланням.

Функція znam() обчислює найменший спільний знаменник двох дробів (їхні чисельники і знаменники передаються як параметри).

Функція rivn() перевіряє, чи є два дроби (чисельники і знаменники яких передаються як параметри) рівними. Результат – логічне значення типу bool.

Функція bilsh() перевіряє, чи є один дріб більше другого. Результат – логічне значення типу bool.

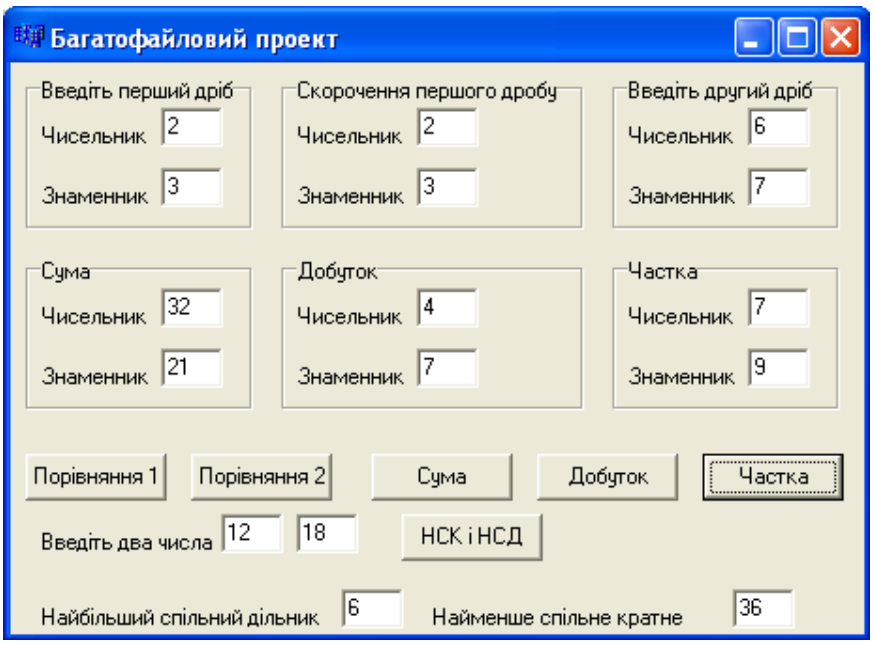
Файл реалізації ratio.cpp:

```
#pragma hdrstop
#include "ratio.h"
#pragma package(smart_init)
void sokr(int &a, int &b)
{ int z=nsd(a,b);    // Обчислюємо найбільший спільний дільник
  a=a/z;  b=b/z;     // Ділимо чисельник і знаменник на НСД
}
int znam(int a, int b)
{ int z=nsk(a,b);     /* Спільний знаменник є найменшим
}                        спільним кратним чисельника і знаменника */
bool rivn(int a1, int b1, int a2, int b2)
{ int z=nsk(b1, b2); // Спільний знаменник.
  int x,y;
  x=z/b1; y=z/b2;      /* Обчислюємо коефіцієнти, на які слід помножити
                    чисельники і знаменники, щоб привести до спільного дільника */
  if(a1*x == a2*y)    // Порівнюємо чисельники здобутих дробів
    return true;
  else return false;
}
bool bilsh(int a1, int b1, int a2, int b2)
{ int z=nsk(b1, b2), x, y;
  x=z/b1; y=z/b2;                // Приводимо до спільного дільника
  if(a1*x > a2*y) return true;  // Порівнюємо чисельники
  else return false;
}
```

Файл Unit1.cpp (головний файл проекту):

```
#include <vcl.h>
#pragma  hdrstop
#include "Unit1.h"
#include "ratio.h"   // Долучаємо заголовний файл ratio.h
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//------------ Перевірка на рівність дробів --------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int a, b, c,  d;
  a=StrToInt(Edit1->Text);  b=StrToInt(Edit2->Text);
  sokr(a, b);
  Edit3->Text=IntToStr(a);  Edit4->Text=IntToStr(b);
  c=StrToInt(Edit5->Text);  d=StrToInt(Edit6->Text);
  sokr(c, d);
  if (rivn(a,b,c,d)) ShowMessage("a/b = c/d");
  else ShowMessage("a/b != c/d");
}
//----------- Порівняння дробів ---------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{ int a, b, c, d;
  a=StrToInt(Edit1->Text);  b=StrToInt(Edit2->Text);
  sokr(a, b);
  Edit3->Text=IntToStr(a);  Edit4->Text=IntToStr(b);
  c=StrToInt(Edit5->Text);  d=StrToInt(Edit6->Text);
  sokr(c, d);
  if (bilsh(a,b,c,d)) ShowMessage("a/b > c/d");
  else ShowMessage("a/b <=c/d");
}
//-------------- Сума дробів ---------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{ int a, b, c, d, s1, s2, x, y, z;
  a=StrToInt(Edit1->Text);  b=StrToInt(Edit2->Text);
  sokr(a, b);
  Edit3->Text=IntToStr(a);  Edit4->Text=IntToStr(b);
  c=StrToInt(Edit5->Text);  d=StrToInt(Edit6->Text);
  sokr(c, d);
  z=znam(b, d);
  x=z/b; y=z/d;
  s2=z;  s1=a*x+c*y;
  sokr(s1, s2);
  Edit7->Text=IntToStr(s1);  Edit8->Text=IntToStr(s2);
}
```

```
//------------ Добуток дробів --------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{ int a, b, c, d, s1, s2, z;
  a=StrToInt(Edit1->Text);  b=StrToInt(Edit2->Text);
  sokr(a, b);
  Edit3->Text=IntToStr(a);  Edit4->Text=IntToStr(b);
  c=StrToInt(Edit5->Text);  d=StrToInt(Edit6->Text);
  sokr(c, d);
  s1=a*c;  s2=b*d;
  sokr(s1,s2);
  Edit9->Text=IntToStr(s1); Edit10->Text=IntToStr(s2);
}
//--------------- Частка дробів ------------------------------
void __fastcall TForm1::Button5Click(TObject *Sender)
{ int a, b, c, d, s1, s2, z;
  a=StrToInt(Edit1->Text);  b=StrToInt(Edit2->Text);
  sokr(a, b);
  Edit3->Text=IntToStr(a);  Edit4->Text=IntToStr(b);
  c=StrToInt(Edit5->Text);  d=StrToInt(Edit6->Text);
  sokr(c, d);
  s1=a*d;  s2=b*c;
  sokr(s1,s2);
  Edit11->Text=IntToStr(s1);  Edit12->Text=IntToStr(s2);
}
//------------ НСД і НСК ---------------------------------
void __fastcall TForm1::Button6Click(TObject *Sender)
{ int a, b;
  a=StrToInt(Edit13->Text);
  b=StrToInt(Edit14->Text);
  Edit15->Text=IntToStr(nsd(a,b));
  Edit16->Text=IntToStr(nsk(a,b));
}
```



Приклад 9.5 Створити багатофайловий проект для опрацювання раціональних дробів і обчислення найбільшого спільного дільника і найменшого спільного кратного. Надати такі можливості: скорочення дробів, обчислення спільного знаменника двох дробів, порівняння на рівність/нерівність та більше/менше, додавання, множення і ділення дробів.

Розв'язок. Створимо новий тип "drib". Цей тип буде структурою з полями `a` (чисельник) і `b` (знаменник). Додамо до файла `ratio.h` заголовки функцій для обчислення суми, добутку і частки дробів, а до файла `ratio.cpp` – реалізацію цих функцій. Програма використовуватиме файл `mat.h`, який було створено у попередньому прикладі.

Файл `ratio.h`:

```
#ifndef ratioH
#include "mat.h"
#define ratioH
//---------------------------------------------------------------
struct drib { int a,b; };
void sokr(drib &d);           // Прототипи функцій скорочення дробів,
int znam(drib d1, drib d2);   // обчислення спільного знаменника,
bool rivn(drib d1, drib d2); // порівняння дробів на рівність,
bool bilsh(drib d1, drib d2);// порівняння дробів на більше/менше,
drib sum(drib d1, drib d2);   // обчислення суми двох дробів,
drib dob(drib d1, drib d2);   // обчислення добутку двох дробів
drib chast(drib d1, drib d2);// і обчислення частки двох дробів.
#endif
```

Файл ratio.cpp:

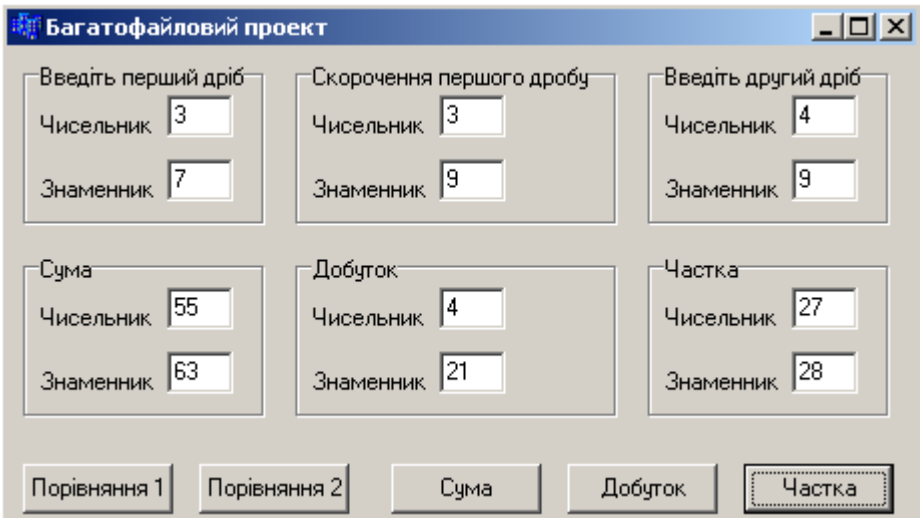
```
#pragma hdrstop
#include "ratio.h"
//---------------------------------------------------------------
#pragma package(smart_init)
void sokr(drib &d)
{ int z=nsd(d.a, d.b);
  d.a=d.a/z;   d.b=d.b/z;
}
int znam(drib d1, drib d2)
{ int z=nsk(d1.b, d2.b);
  return z;
}
bool rivn(drib d1, drib d2)
{ int x, y, z=nsk(d1.b, d2.b);
  x=z/d1.b; y=z/d2.b;
  if(d1.a*x == d2.a*y) return true;
  else return false;
}
bool bilsh(drib d1, drib d2)
{ int x, y, z=nsk(d1.b, d2.b);
```

```
  x=z/d1.b; y=z/d2.b;
  if(d1.a*x > d2.a*y) return true;
  else return false;
}
drib sum(drib d1, drib d2)
{ int x,y,z;  drib s;
  sokr(d1);   sokr(d2);
  z=znam(d1,d2);
  x=z/d1.b;   y=z/d2.b;
  s.b=z;      s.a=d1.a*x+d2.a*y;
  sokr(s);
  return s;}
drib dob(drib d1, drib d2)
{ drib p;
  sokr(d1);    sokr(d2);
  p.a=d1.a*d2.a;  p.b=d1.b*d2.b;
  sokr(p);
  return p;
}
drib chast(drib d1, drib d2)
{ drib r;
  sokr(d1); sokr(d2);
  r.a=d1.a*d2.b;  r.b=d1.b*d2.a;
  sokr(r);
  return r;
}
```

Файл Unit1.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#include "ratio.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//--------------- Порівняння дробів 1 -----------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ drib d1, d2;
  d1.a=StrToInt(Edit1->Text);  d1.b=StrToInt(Edit2->Text);
  sokr(d1);
  Edit3->Text=IntToStr(d1.a);  Edit4->Text=IntToStr(d1.b);
  d2.a=StrToInt(Edit5->Text);  d2.b=StrToInt(Edit6->Text);
  sokr(d2);
  if(rivn(d1,d2)) ShowMessage("a/b = c/d");
```

```
  else ShowMessage("a/b != c/d");
}
//-------------- Порівняння дробів 2 ----------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{ drib d1, d2;
  d1.a=StrToInt(Edit1->Text);  d1.b=StrToInt(Edit2->Text);
  sokr(d1);
  Edit3->Text=IntToStr(d1.a);  Edit4->Text=IntToStr(d1.b);
  d2.a=StrToInt(Edit5->Text);  d2.b=StrToInt(Edit6->Text);
  sokr(d2);
  if (bilsh(d1,d2)) ShowMessage("a/b > c/d");
  else ShowMessage("a/b <=c/d");
}
//---------------- Сума дробів -------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{ drib d1, d2, s;
  d1.a=StrToInt(Edit1->Text);  d1.b=StrToInt(Edit2->Text);
  sokr(d1);
  Edit3->Text=IntToStr(d1.a);  Edit4->Text=IntToStr(d1.b);
  d2.a=StrToInt(Edit5->Text);  d2.b=StrToInt(Edit6->Text);
  sokr(d2);
  s = sum(d1, d2);
  Edit7->Text=IntToStr(s.a);   Edit8->Text=IntToStr(s.b);
}
//----------------- Добуток дробів ----------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{ drib d1, d2, p;
  d1.a=StrToInt(Edit1->Text);  d1.b=StrToInt(Edit2->Text);
  sokr(d1);
  Edit3->Text=IntToStr(d1.a);  Edit4->Text=IntToStr(d1.b);
  d2.a=StrToInt(Edit5->Text);  d2.b=StrToInt(Edit6->Text);
  sokr(d2);
  p = dob(d1,d2);
  Edit9->Text=IntToStr(p.a);   Edit10->Text=IntToStr(p.b);
}
//----------------- Частка дробів -----------------------------
void __fastcall TForm1::Button5Click(TObject *Sender)
{ drib d1, d2, r;
  d1.a=StrToInt(Edit1->Text);  d1.b=StrToInt(Edit2->Text);
  sokr(d1);
  Edit3->Text=IntToStr(d1.a);  Edit4->Text=IntToStr(d1.b);
  d2.a=StrToInt(Edit5->Text);  d2.b=StrToInt(Edit6->Text);
  sokr(d2);
  r = chast(d1, d2);
  Edit11->Text=IntToStr(r.a);  Edit12->Text=IntToStr(r.b);
}
```



9.3 Бібліотеки функцій

Функції, прототипи яких містяться у стандартних заголовних файлах, зазвичай визначено у *бібліотеках* функцій. Бібліотеки функцій C++ – це набір функцій, але не у формі коду, а у вже скомпільованому вигляді. Файли бібліотек C++ мають розширення `lib`. При використанні бібліотек неможливо обійтися без заголовних файлів, оскільки це є єдиний спосіб надати програмі інформацію про функції, які зберігаються у бібліотеках.

Файли бібліотек у Borland C++ Builder можна поділити на три категорії:

✓ файли стандартної бібліотеки C++ (які містять функції та інші елементи, спільні для всіх реалізацій C++);

✓ файли бібліотек Windows;

✓ файли бібліотек Borland C++ Builder, які реалізують специфічні функції C++ Builder.

Для того, щоб використовувати бібліотеку у програмі, яка створюється у Borland C++ Builder, слід долучити до програми цю бібліотеку. Це можна зробити двома способами:

1) скористатися командою інтегрованого середовища **Project / Add To Project**. Для цього слід налагодити діалогове вікно, яке відкриється в такий спосіб, щоб у ньому відображалися файли з розширенням `lib`, та обрати потрібний файл бібліотеки;

2) долучити бібліотеку директивою `#pragma link`. У загальному вигляді використання цієї директиви виглядає як

```
#pragma link "<ім'я файла бібліотеки>"
```

9.4 Директиви препроцесора

Препроцесор – це комп'ютерна програма, яка приймає дані на вході і видає дані, призначені для входження іншої програми, приміром, такої як компілятор.

У мови програмування C та C++ вбудовано підтримку препроцесора. Рядки в початковому коді, які мають бути опрацьовані препроцесором у вигляді `#define` та `#include`, називаються препроцесорними директивами.

Директиви препроцесора є інструкціями, записаними в тексті С-програми, котрі виконуються до трансляції програми. Директиви препроцесора дозволяють змінювати текст програми, наприклад, замінювати певні лексеми в тексті, вставляти текст з іншого файла, забороняти трансляцію частини тексту тощо. Всі директиви препроцесора розпочинаються зі знаку **#.** Після директив препроцесора крапка з комою не ставиться.

У програмуванні термін "директива" (вказівка) за використанням є схожий до терміну "команда", оскільки так само використовується для опису певних конструкцій мови програмування (тобто вказівок компіляторові чи асемблерові стосовно особливостей опрацювання при компіляції).

Узагальнений формат директиви препроцесора є

<ім'я_директиви> <лексеми_препроцесора>

Тут до *лексем препроцесора* належать символьні константи, імена файлів, які долучаються, ідентифікатори, знаки операцій, розділові знаки, рядкові константи (рядки) й будь-які символи, відмінні від пробілу.

Залежно від препроцесора перед символом # і після нього в директиві можуть бути дозволені пробіли. Пробіли також дозволені перед лексемами препроцесора, поміж ними і після них. За завершення препроцесорної директиви слугує кінець текстового рядка (за наявності символу `'\'`, який позначає перенесення рядка, завершенням препроцесорної директиви буде ознака кінця наступного рядка тексту).

У препроцесорах мови С визначені такі препоцессорні директиви:

`#define` – визначення макросу чи препроцесорного ідентифікатора;

`#include` – долучення тексту з файла;

`#undef` – скасування визначення макросу чи препроцесорного ідентифікатора;

`#if` – перевірка умови-виразу;

`#ifdef` – перевірка визначеності ідентифікатора;

`#ifndef` – перевірка невизначеності ідентифікатора;

`#else` – початок альтернативної гілки для `#if`;

`#endif` – завершення умовної директиви `#if`;

`#elif` – складена директива `#else/#if`;

`#line` – змінення номера наступного нижче рядка;

`#error` – формування тексту повідомлення про помилку при трансляції;

`#pragma` – дії, передбачені реалізацією;

`#` – порожня директива.

Окрім препроцесорних директив, є три препроцесорні операції, які використовуються разом з командою `#define`:

`defined` – перевірка істинності операнда;

`##` – конкатенація препроцесорних лексем;

`#` – перетворення операнда на рядок символів.

Директива **`#define`** має кілька модифікацій. Вони передбачають визначення макросів та препроцесорних ідентифікаторів, кожному з яких ставиться у відповідність певна послідовність символів. У подальшому тексті програми

препроцесорні ідентифікатори замінюються на раніш визначені послідовності символів. Так відбувається, наприклад, при визначенні констант за допомогою директиви `#define`. Директива `#define` слугує для заміни констант, які часто використовуються, ключових слів, операторів чи виразів на певні ідентифікатори. Ідентифікатори, які замінюють текстові чи то числові константи, називають *іменованими константами*. Ідентифікатори, які замінюють фрагменти програм, називають макровизначеннями, причому макровизначення можуть мати аргументи. Директива `#define` має дві синтаксичні форми:

```
#define <ідентифікатор> <текст>
#define <ідентифікатор> <(список параметрів)> <текст>
```

Ця директива замінює всі подальші входження ідентифікатора на текст. Такий процес називається *макропідстановкою*. Текст може бути яким завгодно фрагментом програми на С, а також може бути відсутнім. В останньому разі всі екземпляри ідентифікатора вилучаються з програми.

Щоб задати константу `n` зі значенням 10 слід написати директиву

```
#define n 10
```

Ця директива є аналогом такої інструкції

```
const int n = 10;
```

Відмінності полягають у тому, що:

1) при використанні константи пам'ять під константну змінну виділяється, а при використанні директиви – ні;

2) при використанні директиви не виконується перевірка типу значення, що може спричинитися до помилок у програмі;

3) тип значення `n` при використанні директиви визначається автоматично, тобто, якщо у програмі є суттєво, щоб `n` мала тип `unsigned short`, зробити це за допомогою директиви неможливо.

Отже, використання директиви `#define` у C++ без особливої на те потреби є небажаним.

Директива **`#include`**, яку було розглянуто вище, долучає до тексту програми вміст зазначеного файла. Ця директива широко використовується для долучення до програм так званих заголовних файлів. Існують три форми цієї директиви:

```
#include <ім'я_заголовного_файла>
#include "ім'я_заголовного_файла"
#include  ідентифікатор_макроса
```

Відмінність між першими двома формами директиви полягає у методі пошуку препроцесором файла, що долучається. Якщо ім'я файла записано у кутових дужках, то послідовність пошуку препроцесором заданого файла у каталогах визначається заданими каталогами включення (include directories). Якщо ім'я файла записано у лапках, препроцесор шукає файл, переглядаючи каталоги у такій послідовності:

- ✓ каталог того файла, який містить директиву `#include`;
- ✓ каталоги файлів, які долучили цей файл директивою `#include`;

- ✓ поточний каталог;
- ✓ каталоги, вказані опцією компілятора `/I`;
- ✓ каталоги, задані змінною оточення `INCLUDE`.

Якщо ім'я файла зазначено зі шляхом, то препроцесор ніде більше цей файл не буде шукати.

Третя форма директиви `#include` припускає наявність макросу, який визначає файл, що долучається. Ідентифікатор макроса не повинен починатися з символів "<" та "".

Наприклад, для долучення файла `vcl.h`, який шукається у стандартному каталозі, директива має вигляд:

```
#include <vcl.h>
```

Наведемо поширений приклад директиви долучення файла `Unit1.h`, який шукається передусім у каталозі, де розташований файл, який містить цю директиву:

```
#include "Unit1.h"
```

Покажемо директиви, які долучають файл `C:\Test\My.h`, що шукатиметься лише у каталозі `C:\Test`:

```
#define myincl "C:\Test\My.h"
#include myincl
```

Директива **`#undef`** використовується для відмінення чинності поточного визначення директиви `#define` для зазначеного ідентифікатора. Не є помилкою використання директиви `#undef` для ідентифікатора, який не було визначено директивою `#define`. Синтаксис цієї директиви є таким:

```
#undef <ідентифікатор>
```

Як було зазначено вище, заголовний файл також може містити директиви `#include`. Тому іноді складно зрозуміти, які саме заголовні файли долучено до програмного коду, і деякі заголовні файли може бути долучено декілька разів. Уникати цього дозволяють умовні директиви препроцесора.

Розглянемо приклад:

```
#ifndef BIBL_H
#define BIBL_H
// Вміст файла bibl.h
#endif
```

Умовна директива **`#ifndef`** перевіряє, чи не було значення `BIBL_H` визначено раніше (`BIBL_H` – це константа препроцесора; такі константи узвичаєно писати великими літерами). Препроцесор опрацьовує наступні рядки аж до директиви `#endif`. Інакше він пропускає рядки від `#ifndef` до `#endif`. Директива `#define BIBL_H` визначає константу препроцесора `BIBL_H`. Розмістивши цю директиву безпосередньо після директиви `#ifndef`, можна гарантувати, що змістовну частину заголовного файла `bibl.h` буде включено до початкового тексту лише один раз, скільки б разів не долучався до тексту сам цей файл.

Директива **#if** та її модифікації **#ifdef**, **#ifndef** разом з директивами **#else**, **#endif**, **#elif** дозволяють організувати умовне опрацювання тексту програми. При використовуванні цих засобів компілюється не весь текст, а лише ті його частини, які обираються за допомогою вищенаведених директив. Головна ідея полягає у тому, що, якщо вираз, який розміщено після директив `#if`, `#ifdef`, `#ifndef` виявиться істинним, то буде скомпільовано код, розташований поміж однією з цих трьох директив та директивою `#endif`, інакше цей код буде опущений. Директива `#endif` використовується для позначення завершення блока `#if`. Директиву `#else` можна використовувати з кожною із наведених вище директив для надання альтернативного варіанта компіляції.

Загальна форма запису директиви `#if`:

```
#if <константний_вираз>
```

Якщо *константний вираз* є істинним, буде скомпільовано код, розташований безпосередньо за цією директивою.

Директива **#line** дозволяє зазначати ім'я файла і бажаний початковий номер рядка. Базова форма запису цієї команди є такою:

```
#line <номер> * <ім'я_файла> *
```

Тут *номер* – це певне додатне число, а *ім'я файла* – допустимий ідентифікатор файла. Значення параметру *номер* стає номером поточного початкового рядка, а значення параметру *ім'я файла* – ім'ям початкового файла. Параметр *ім'я файла* є необов'язковий.

Директива **#line** переважно використовується з метою налагодження програм і в спеціальних додатках.

Директива **#error** дозволяє задавати текст діагностичного повідомлення, яке виводиться при виявленні помилок. За наявності цієї директиви відображується задане повідомлення і номер рядка.

Дія директиви **#pragma** залежить від конкретної реалізації компілятора. Директива дозволяє видавати компіляторові різні інструкції.

Директива **#** є порожньою директивою і завжди ігнорується.

9.5 Область дії та простір імен

Реальний програмний проект розробляється як кілька вихідних файлів, які компілюються окремо один від одного, а потім поєднуються редактором зв'язків. Як було зазначено у підрозд. 8.1, змінні, оголошені всередині тіла функції, називають *локальними*. Такі змінні діють лише всередині своєї функції. Функції з різних файлів можуть використовувати глобально доступні зовнішні змінні. Самі функції зазвичай є зовнішніми і доступними з будь-яких файлів. Кожна змінна характеризується областю дії, областю видимості і тривалістю життя. Схеми розподілу пам'яті C++ визначають, як довго змінні лишаються в пам'яті, та які частини програми мають до них доступ.

Область видимості змінної – це частина тексту програми, в якій може бути використано змінну. Змінна вважається за видиму, якщо відомі її тип та ім'я. Змінна може бути видимою у межах блока, файла чи всієї програми. Якщо

змінна оголошена у файлі на зовнішньому рівні, вона є видима від оголошення і до кінця цього файла. Змінну можна оголосити глобально видимою за допомогою відповідних специфікаторів у вихідних файлах програми. Якщо змінна оголошена у блоці, вона є видимою всередині цього блока і блоків, вкладених у нього.

Нагадаємо, що блоком у програмі є складений оператор чи послідовність операторів, обмежених операторними дужками `{}`. При цьому з точки зору синтаксису блок вважається єдиним оператором. Окрім того, блоки можуть містити складені оператори, але не можуть містити визначення функцій, тобто всередині функції не можна визначити іншу функцію.

Тривалість життя змінної – це інтервал виконання програми, під час якого змінна існує в пам'яті. Змінна з глобальним часом існування має відведену для неї пам'ять упродовж всього часу виконання програми, а змінні з локальним часом життя – лише під час виконання відповідного блока програми. Функції у програмі на С++ мають глобальний час життя.

Як відомо, синтаксис оголошення змінної у програмі є такий:

`[`<*клас пам'яті*>`][` <*тип*>`]` <*ім'я*> `=[`<*ініціалізатор*>`];`

Існують такі специфікатори класів пам'яті:

1) `auto`; 3) `extern`; 5) `volatile`;
2) `register`; 4) `static`; 6) `mutable`.

Специфікатор `auto` (так звана автоматична змінна) використовується для оголошування лише локальних змінних, які є видимі лише в тому блоці, в якому вони оголошені. Цей специфікатор призначається за замовчуванням, а тому його можна явно не зазначати:

```
auto int X=10;      // Те саме, що  int X=10;
```

Специфікатор `register` вказує компілятору по можливості розмістити змінну в регістрах процесора. Оскільки кількість регістрів процесора є обмежена, кількість таких змінних має бути невеликою. За невдалої спроби такого розміщення змінна буде вести себе подібно до локальної змінної типу `auto`. Розміщення змінних у регістрах оптимізує програмний код за швидкістю, оскільки процесор оперує зі змінними, розміщеними у регістрах, набагато швидше, ніж з пам'яттю. Регістрова пам'ять, якщо вона є, може бути виділена лише для змінної типу `int` і вказівників:

```
register int Y;
```

Специфікатори `auto` та `register` застосовуються лише для локальних змінних. Спроба застосування цих специфікаторів для оголошення глобальних змінних призведе до помилки.

Специфікатор `extern` (зовнішній) використовують для посилання на змінну, оголошену в іншому файлі (див. підрозд. 9.1). Якщо визначення зовнішньої змінної розміщено у тому ж самому файлі, але пізніше, ніж її використання, слід теж застосовувати оголошення з специфікатором `extern`.

Специфікатор `static` (статичний) застосовують для оголошення статичних змінних, які зберігають власні значення від одного входження до функції (блока) до наступного входження до цієї функції (блока) протягом усього часу виконання програми. Статичну змінну не можна оголосити в іншому файлі як

зовнішню. Змінні з специфікатором `static` було розглянуто у підрозділах 8.2 та 9.1.

Специфікатор `volatile` (нестабільний) зазначає, що значення змінної (комірки пам'яті) може бути змінено апаратними засобами чи іншою системною програмою протягом виконання програми.

Специфікатор `mutable` (змінюваний) використовується винятково для класів і зазначає, що значення елемента даних завжди є доступне для модифікації, навіть, якщо цей елемент належить до константного об'єкта.

Правила видимості змінних і функцій:

1) змінні, визначені на зовнішньому рівні, є видимі від точки оголошення до кінця файла. Можна оголосити змінні видимими і в інших вихідних файлах за допомогою специфікатора `extern`. Однак, якщо змінна визначена на зовнішньому рівні з специфікатором `static`, вона є видима лише в тому файлі, в якому вона визначена;

2) змінна, оголошена чи визначена всередині блока, є видима від точки оголошення до кінця блока;

3) змінні з зовнішніх блоків, включаючи змінні, оголошені на зовнішньому рівні, є видимі у внутрішніх блоках. Якщо змінна, оголошена всередині блока, має те ж саме ім'я, що і змінна, оголошена у зовнішньому блоці, то визначення змінної в блоці змінює зовнішню змінну;

4) функції з класом пам'яті `static` є видимі лише у тому файлі, в якому вони визначені. Решта функцій є глобально видимими, однак для їхнього використання необхідно оголошувати прототипи. Мітка у тілі функції є видима у цьому тілі і тільки у ньому. Імена формальних параметрів функції видимі від точки оголошення до кінця тіла функції.

Правила визначення тривалості життя змінної:

1) змінна, оголошена ззовні блоків програми, має глобальну тривалість життя;

2) змінна, оголошена всередині блока, має локальну тривалість життя, якщо вона оголошена без специфікатора `static`.

Області дії

Кожен програмний об'єкт має область дії, яка визначається видом і місцем його оголошення. Існують такі категорії області дії: блок, файл, функція, прототип функції, клас і поіменована область.

Блок. Ідентифікатори, оголошені всередині блока, є локальними. Область дії ідентифікатора розпочинається з точки визначення і поширюється до кінця блока, видимість ідентифікатора – в межах блока і його внутрішніх блоків, тривалість життя ідентифікатора – до виходу з блока.

Файл. Ідентифікатори, оголошені за межами будь-якого блока, функції, класу чи простору імен, мають глобальну видимість і постійну тривалість життя, і можуть використовуватися з моменту їх визначення у файлі.

Функція. Існує лише один тип ідентифікаторів – мітки операторів, – для яких областю видимості є все тіло функції (незалежно від того, в якому блоці

оголошено мітку). У межах однієї функції всі мітки мають бути різними, але імена міток у різних функціях можуть збігатися.

Прототип функції. Ідентифікатори, які зазначено у переліку параметрів прототипу (заголовка, оголошення) функції, мають область дії лише прототип функції.

Клас. Елементи структур, об'єднань і класів (за винятком статичних елементів) є видимими лише у межах класу. Вони утворюються при створюванні змінної зазначеного типу і руйнуються при її знищенні.

Іменована область. C++ дозволяє явно задавати область визначення імен як частину глобальної області за допомогою оператора `namespace`.

Область видимості співпадає з областю дії, за винятком ситуації, коли у вкладеному блоці оголошено змінну з таким самим ім'ям. У цьому разі зовнішня змінна у вкладеному блоці є невидима, хоча цей блок і входить в її область дії. До цієї змінної, якщо вона є глобальна, можна звертатися, використовуючи операцію визначення діапазону доступу (`::`).

Простори імен

У кожній області дії розрізняють так звані простори імен (`namespace`). Простори імен надають можливість створювати іменовані області пам'яті, в яких можна оголошувати ідентифікатори. Вони призначені для зниження вірогідності конфліктів імен, коли імена з одного простору пам'яті не конфліктують з тими самими іменами, оголошеними в інших просторах пам'яті. Це особливо важливо для великих програм, які використовують програмні коди, розроблені різними постачальниками програмних продуктів. Ідентифікатори у просторах імен можуть стати доступними при застосуванні операції доступу (`::`) або директиви `using`.

У різних просторах імена можуть збігатися, наприклад:

```
struct Node
{ int Node;  int i;
} Node;
```

Тут протиріч немає, оскільки імена типу, змінної та елементи структури належать до різних просторів.

У C++ визначено чотири класи ідентифікаторів, у межах кожного з них імена мають бути унікальними:

- ✓ до першого класу належать імена змінних, функцій, типів користувача (визначених за допомогою `typedef`) і перераховних констант у межах однієї області видимості. Всі вони, окрім імен функцій, можуть бути перевизначені у вкладених блоках;
- ✓ другий клас імен утворюють імена типів перерахувань, структур, класів і об'єднань. Кожне ім'я має відрізнятися від імен інших типів у тій самій області видимості;
- ✓ третій клас становлять елементи структури, класу й об'єднання. Ім'я елемента має бути унікальним усередині кожної структури, але може збігатися з іменами елементів інших структур;
- ✓ мітки операторів утворюють окремий клас імен.

Іменовані області слугують для логічного групування оголошень та обмеження доступу до них. Чим більше є програма, тим більш актуальним є використання іменованих областей. Найпростішим прикладом такого використання є відокремлення коду, написаного однією людиною, від коду, написаного іншою людиною. За використання єдиної глобальної області видимості формувати програму з окремих частин надто складно через можливість збігу й конфлікту імен.

Оголошення іменованої області (її називають також простором імен) має вигляд

```
namespace [<ім'я області>] { <оголошення> }
```

Іменовані області можуть оголошуватись неодноразово, всі наступні оголошення розглядаються як розширення попередніх. Отже, іменована область може поновлюватися і змінюватися за рамками одного файла.

Якщо ім'я області не є задане, компілятор визначає його самостійно за допомогою унікального ідентифікатора, який є різним для кожного модуля. Оголошення об'єкта в неіменованій області є рівнозначне до його оголошення як глобального з специфікатором `static`. Розміщувати змінні в такій області корисно для того, щоб зберегти локальність коду. Не можна отримати доступ з одного файла до елемента неіменованої області іншого.

Приклад іменованої області:

```
namespace demo
{ int i=1, k=0;
  void func1(int);
  void func2(int){/* … */}
}
namespace demo
{                          // int i=2; Неправильно – подвійне визначення.
  void func1(double);      // Перевантаження.
  void func2(int);         // Правильно (повторне оголошення).
}
```

В оголошенні іменованої області можуть бути присутні як оголошення, так і визначення. Логічно розміщувати у ній лише оголошення, а визначати пізніше за допомогою імені області й операції доступу до області видимості (`::`), наприклад:

```
void demo :: func1(int) {/* … */}
```

Це застосовується для розділення інтерфейсу й реалізації. У такий спосіб не можна оголосити новий елемент простору імен.

Об'єкти, які було оголошено всередині області, є видимими з моменту оголошення. До них можна явно звертатися за допомогою імені області й операції доступу до області видимості `::`, наприклад:

```
demo::i=100; demo::func2(10);
```

Якщо ім'я часто використовується зовні свого простору, можна оголосити його доступним за допомогою оператора `using`:

```
using demo::i;
```

Після цього можна використовувати ім'я без явного зазначання області.

Якщо є потреба зробити доступними всі імена з певної області, використовується оператор `using namespace`:

```
using namespace demo;
```

Оператори `using` та `using namespace` можна використовувати всередині іменованої області, щоб зробити в ній доступними оголошення з іншої області:

```
namespace Department
{ using demo::i;
  . . .
}
```

Імена, оголошені в іменованій області явно чи за допомогою оператора `using` мають пріоритет щодо імен, оголошених за допомогою оператора `using namespace` (це має значення при долученні кількох іменованих областей, які містять збіжні імена).

Короткі імена просторів імен можуть увійти у конфлікт один з одним, а довгі є непрактичні при написанні реального коду, тому дозволяється вводити синоніми імен:

```
namespace DIT = Department_of_Informational_Technologies;
```

Об'єкти стандартної бібліотеки визначено у просторі імен `std`.

Механізм просторів імен разом з директивою `#include` забезпечує потрібну при написанні великих програм гнучкість завдяки логічному групуванню пов'язаних величин разом з обмеженням доступу.

Здебільшого, у кожному завершеному фрагменті програми можна виокремити інтерфейсну частину (наприклад заголовки функцій, опис типів), потрібну для використання цього фрагмента, і частину реалізації, тобто допоміжні змінні, функції та інші засоби, доступ до яких зовні не потрібний. Простори імен дозволяють приховати деталі реалізації і в такий спосіб спростити структуру програми та зменшити кількість потенційних помилок. Добре помірковане розбиття програми на модулі, чітка специфікація інтерфейсів й обмеження доступу дозволяють організувати ефективну роботу над проектом групи програмістів.

Створимо багатофайловий проект у консолі, в якому проілюструємо деякі властивості простору імен.

Перший файл – заголовний; він містить деякі з елементів, які зазвичай розміщуються у заголовних файлах, – константи, визначення структур і прототипи функцій. У цьому прикладі елементи розміщено у двох просторах імен. Перший простір з ім'ям `pers` містить визначення структури `Person` і прототипи двох функцій: перша вносить до структури ім'я людини, а друга – відбиває вміст структури. Другий простір імен, `debts`, визначає структуру для зберігання імені людини і суми грошей, яку має ця людина. У цій структурі використовується структура `Person`, тобто простір імен `debts` містить директиву `using`, щоб зробити імена з `pers` доступними у просторі імен `debts`. Простір імен `debts` містить також деякі прототипи.

Другий файл складено за звичайним шаблоном, згідно з яким слід мати файл з реалізацією (кодом), у котрому визначаються функції із заголовного файла. Імена функцій, оголошені у просторі імен, мають діапазон доступу простору імен, тобто визначення імен мають бути в тому самому просторі імен, що й оголошення. Долучення просторів імен виконується за допомогою долучення заголовного файла `namesp.h`.

Третій файл програми – це файл з кодом, в якому використовуються структури та функції, визначені у просторі імен.

Слід звернути увагу на те, що в `using`-оголошенні використовується лише ім'я. Наприклад, в оголошенні

```
using debts::showDebt;
```

не визначається тип, що повертається, чи сигнатура функції `showDebt`, а міститься лише ім'я цієї функції (отже, якщо функцію перевантажено, одне `using`-оголошення буде імпортовано до всіх її версій). Окрім того, хоча в обох функціях, `Debt` та `showDebt`, використовується тип `Person`, немає потреби імпортувати ім'я `Person`, оскільки у просторі імен `debt` вже міститься `using`-директива, яка долучає простір імен `pers`.

У функції `other()` використовується менш вдалий спосіб імпортування всього простору імен за допомогою директиви `using`.

Оскільки директива `using` в `debts` імпортує простір імен `pers`, у функції `other()` можна буде використовувати тип `Person` та функцію `showPerson()`.

У функції `another()` використовується `using`-оголошення та операція визначення діапазону доступу для доступу до окремих імен.

Файл namespace.h

```
namespace pers
{ const int LEN=40;
  struct Person
  { char fname[LEN], lname[LEN];  };
  void getPerson(Person &);
  void showPerson(const Person &);
}
namespace debts
{ using namespace pers;
  struct Debt
   { Person name; double amount;   };
  void getDebt(Debt &);
  void showDebt(const Debt &);
  double sumDebts(const Debt ar[], int n);
}
```

Файл namespace.cpp

```
namespace pers
{ void getPerson(Person & rp)
  { cout<<"Enter first name: ";
    cin>>rp.fname;
```

```
    cout<<"Enter last name: ";
    cin>>rp.lname;
  }
  void showPerson(const Person & rp)
  { cout<<rp.lname<<", "<<rp.fname;  }
}
namespace debts
{ void getDebt(Debt & rd)
  { getPerson(rd.name);
    cout<<"Enter debt: ";
    cin>>rd.amount;
  }
  void showDebt(const Debt & rd)
  { showPerson(rd.name);
    cout<<": $"<<rd.amount<<endl;
  }
  double sumDebts(const Debt ar[], int n)
  { double total=0;
    for(int i=0; i<n; i++)  total+=ar[i].amount;
    return total;
  }
}
```

Файл Unit1.h

```
#include <iostream.h>
#include <conio.h>
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include "namesp.h"
void other();
void another();
int main()
{ using debts::Debt;      // Робить доступним визначення структури Debt
  using debts::showDebt;  // Робить доступною функцію showDebt
  Debt golf={{"Benny","Goatsniff"},120.0};
  showDebt(golf);
  other();
  another();
  getch(); return 0;
}
void other()
{ using namespace debts;// Робить доступними для функції other() всі імена
                        // з просторів debts та pers.
  Person dg={"Doodles","Glister"};
  showPerson(dg);
  cout<<endl;
  Debt zippy[3];  int i;
  for(i=0; i<3; i++) getDebt(zippy[i]);
```

```
  for(i=0; i<3; i++) showDebt(zippy[i]);
  cout<<"Total debt: $"<<sumDebts(zippy, 3)<<endl;
}
void another()
{ using pers::Person;
  Person collector={"Milo", "Rightshift"};
  pers::showPerson(collector);      // Визначення діапазону доступу для
                                    // доступу до окремих імен.
  cout<<endl;
}
```

Вікно з результатами виконання програми:

Питання та завдання для самоконтролю

1) В якому файлі містяться прототипи математичних функцій?

2) Чим відрізняється оголошення від визначення змінних у багатофайлових проектах?

3) Назвіть можливі варіанти розміщення прототипів функцій?

4) Яке призначення заголовних файлів?

5) Що міститься у файлах реалізації?

6) У чому полягають особливості оголошення простої змінної з використанням зарезервованого слова `extern`?

7) За допомогою якої препоцессорної директиви долучають заголовні файли?

8) Що являють собою бібліотеки функцій C++?

9) Чим різняться директиви `#include "bibl.h"` та `#include <bibl.h>`?

10) Для чого у мові C++ використовують директиви препроцесора?

11) Назвіть відомі Вам препроцесорні директиви.

12) Назвіть умовні директиви опрацювання текстів програм.

13) Назвіть та охарактеризуйте категорії області дії програмних об’єктів.

14) Розтлумачте відмінності між такими задаваннями констант:

```
#define nom 256
const int nom = 256;
```

15) Що розуміють під поняттям “тривалість життя змінної”?

16) Які існують специфікатори класів пам’яті при оголошенні змінних у програмі? Який саме специфікатор призначається за замовчуванням?

17) Назвіть правила видимості змінних і функцій.

18) Які існують категорії області дії програмних об’єктів? Охарактеризуйте кожний з них.

19) З якою метою оголошуються іменовані області (простори імен)?

20) Охарактеризуйте використання операції доступу “`::`”.

Розділ 10

Типи опрацювання дати і часу

10.1 Тип дата-час у C++

Функції й типи даних, необхідні для опрацювання дати і часу, оголошено у заголовному файлі `time.h`. Зокрема цей файл містить визначення типів даних `time_t`:

```
typedef long time_t;
typedef long clock_t;
```

і структури `tm`, оголошеної в такий спосіб:

```
struct tm
{ int   tm_sec;
  int   tm_min;
  int   tm_hour;
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
};
```

Основні з цих функцій з описом їхньої роботи наведено у табл. 10.1.

Таблиця 10.1

Функції опрацювання дати і часу

Синтаксис функції	Опис роботи функції
`char *asctime (struct tm time);`	Перетворює дату і час `time` з формату структури типу `tm` на символьний рядок з 26-ти символів, який може мати вигляд `Mon Jan 04 02:03:55 2010\n\0` Функція повертає вказівник на цей рядок
`clock_t clock();`	Повертає процесорний час, який сплинув від початку виконання програми, чи –1, якщо він є невідомий
`char *ctime(long *time);`	Перетворює час `time` формату `long*` у символьний рядок з 26 символів, який може вигляд: `Mon Jan 04 02:03:55 2010\n\0` Функція повертає вказівник на цей рядок
`double difftime (time_t time2, time_t time1);`	Повертає різницю поміж `time2` і `time1` типу `time_t`, тобто час у секундах, який сплине від `time2` до `time1`, як число з подвійною точністю

Закінчення табл. 10.1

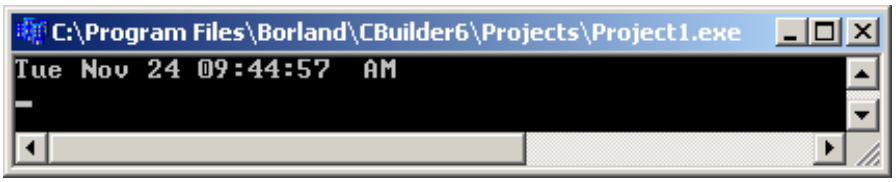
Синтаксис функції	Опис
`struct tm *gmtime (long *time);`	Перетворює `time` типу `long*` до формату структури `tm`
`struct tm *localtime (long *time);`	Перетворює `time` типу `long*` до формату структури `tm` з урахуванням зони місцевого часу, якщо попередньо було задано глобальну змінну часової зони `TZ`
`time_t mktime (struct tm *time);`	Перетворює час і дату `time` формату структури `tm` до формату `time_t`
`char * _strdate (char *date);`	Перетворює поточну дату на символьний рядок `time` з 9 символів формату `MM/DD/YY`
`char * _strtime (char *time);`	Перетворює поточний час на символьний рядок `time` з 9 символів формату `HH:MM:SS`
`size_t strftime(char *s, size_t maxsize, char *fmt, struct tm *t);`	Перетворює дату і час `t` формату структури `tm` на символьний рядок `s` розміром `maxsize` у форматі `fmt`
`time_t time (time_t *timer);`	Повертає поточний календарний час (тобто час в секундах, який сплинув з півночі (00:00:00) 1 Січня 1970 за Грінвічем)
`void tzset();`	Функція встановлює значення глобальних змінних `daylight` (ненульове значення типу `int` означає дозвіл переходу на літній час), `timezone` (значення типу `long` задає різницю в секундах поміж місцевим часом і часом за Грінвічем (GMT)) і `tzname` (рядок типу `*char` трилітерних часових зон)

Проілюструємо роботу деяких з вищенаведених функцій на прикладі консольної програми для виведення поточної дати й часу в заданому форматі. Час виводитиметься у 12-годиному форматі.

```
#include <vcl.h>
#pragma hdrstop
#include <stdio.h>
#include <time.h>
#include <conio.h>
#pragma argsused
//---------------------------------------------------------
int main(int argc, char* argv[])
{ struct tm *newtime;   char *am_pm="PM";
  time_t long_time;          // Оголошення змінної long_time.
  time(&long_time);          // Запис поточного часу до long_time.
  newtime=localtime(&long_time);  // Перетворення long_time
                             //  на структуру newtime.
  if(newtime->tm_hour<12)    //Якщо проминуло менше за 12 годин доби,
     am_pm="AM";       //відбудеться відповідне визначення рядку формату.
```

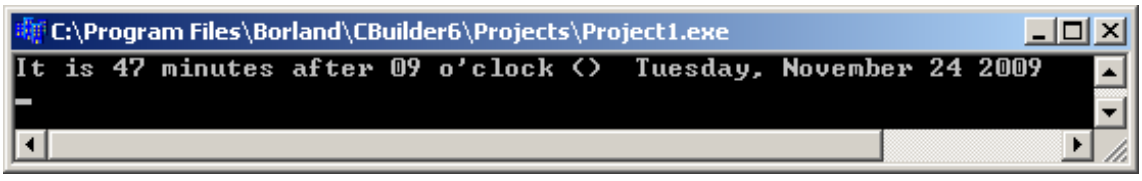
```
    if(newtime->tm_hour>12)    // Якщо проминуло більше за 12 годин доби,
       newtime->tm_hour-=12;   // відбудеться віднімання 12.
    printf("%.20s %s\n", asctime(newtime), am_pm);
    getch();    return 0;
}
```

Результат виконання наведеної вище консольної програми матиме вигляд



Таке само завдання можна реалізувати дещо інакше:

```
#include <vcl.h>
#pragma hdrstop
#include <stdio.h>
#include <time.h>
#include <conio.h>
#pragma argsused
//---------------------------------------------------------
int main(int argc, char* argv[])
{  struct tm *time_now;
   time_t s_now;
   char str[80];
   tzset();
   time(&s_now);
   time_now = localtime(&s_now);
   strftime(str, 80,
   "It is %M minutes after %I o'clock (%Z)  %A, %B %d 20%y",
       time_now);
   printf("%s\n",str);
   getch();    return 0;
}
```



10.2 Тип дата-час у C++ Builder

Для опрацювання дати й часу в C++ Builder існує спеціальний тип `TDateTime`. Змінна такого типу займає 8 байтів і за суттю є дійсним числом з фіксованою крапкою, ціла частина якого задає кількість днів, що минули починаючи з 30.12.1899, а дійсна частина цього числа – збіглу частину поточного дня, починаючи з 0 годин (час). Для роботи с даними типу `TDateTime` існує багато функцій і методів. У табл. 10.2 наведено функції перетворювання дати й часу на рядок та навпаки.

Таблиця 10.2

Функції перетворювання дати й часу

Синтаксис функції	Опис
AnsiString **DateToStr** (TDateTime Date);	Перетворює дату Date на рядок
TDateTime **StrToDate** (AnsiString S);	Перетворює рядок S на дату
AnsiString **TimeToStr** (TDateTime Time);	Перетворює час Time на рядок
TDateTime **StrToTime** (AnsiString S);	Перетворює рядок S на час
AnsiString **DateTimeToStr** (TDateTime dt);	Перетворює значення dt типу TDateTime на рядок
TDateTime **StrToDateTime** (AnsiString S);	Перетворює рядок S на дату і час типу TDateTime
DateTimeToString (AnsiString &S, AnsiString Format, TDateTime dt);	Перетворює значення dt типу TDateTime на рядок S за форматом Format
AnsiString **FormatDateTime** (AnsiString Format, TDateTime dt);	Повертає рядок, як результат перетворення значення dt типу TDateTime за форматом Format
TDateTime **StrToDateDef** (AnsiString S, TDateTime Default);	Функція намагається перетворити рядок S на дату типу TDateTime. Якщо перетворення відбулося, функція повертає це значення, а за випадку неможливості перетворення – значення Default
TDateTime **StrToDateTimeDef** (AnsiString S, TDateTime Default);	Функція намагається перетворити рядок S на дату і час типу TDateTime. Якщо перетворення відбулося, функція повертає це значення, а за випадку неможливості перетворення – значення Default
TDateTime **StrToTimeDef** (AnsiString S, TDateTime Default);	Функція намагається перетворити рядок S на час типу TDateTime. Якщо перетворення відбулося, функція повертає це значення, а за випадку неможливості перетворення – значення Default
bool **TryStrToDate** (AnsiString S, TDateTime &dt);	Перетворює рядок S на дату dt типу TDateTime. Якщо перетворення відбулося, функція повертає true, інакше повертає false
bool **TryStrToTime** (AnsiString S, TDateTime &dt);	Перетворює рядок S на час dt типу TDateTime. Якщо перетворення відбулося, функція повертає true, інакше повертає false
bool **TryStrToDateTime** (AnsiString S, TDateTime &dt);	Перетворює рядок S на дату і час dt типу TDateTime. Якщо перетворення відбулося, функція повертає true, інакше повертає false

У функціях `DateTimeToString` та `FormatDateTime` з табл. 10.2, використовується рядок форматування дат. У табл. 10.3 наведено значення специфікаторів формату дат.

Таблиця 10.3

Специфікатори формату дат

Специфікатор	Опис
`C`	Виведення дати і часу за заданою глобальною змінною `ShortDateFormat`
`d`	Виведення дня без початкових нулів (1 – 31)
`dd`	Виведення дня з початковим нулем (01 – 31)
`ddd`	Виведення дня у вигляді позначення (пн – нд)
`dddd`	Виведення дня у вигляді повної назви дня тижня (понеділок – неділя)
`m`	Виведення місяця без початкових нулів (1 – 12)
`mm`	Виведення місяця з початковими нулями (01 – 12)
`mmm`	Виведення місяця у вигляді позначень (січ – грд)
`mmmm`	Виведення повної назви місяця (січень – грудень)
`yy`	Виведення двох останніх цифр року (00 – 99)
`yyyy`	Виведення року у повному форматі (0000 – 9999)
`h`	Виведення годин без початкового нуля (0 – 23)
`hh`	Виведення годин з початковим нулем (00 – 23)
`n`	Виведення хвилин без початкового нуля (0 – 59)
`nn`	Виведення хвилин з початковим нулем (00 – 59)
`s`	Виведення секунд без початкового нуля (0 – 59)
`ss`	Виведення секунд з початковим нулем (00 – 59)
`am/pm` `a/p`	Використання 12-годинної шкали та символів `am` або `pm` Використання 12-годинної шкали і символів `a` або `p`

У табл. 10.4 подано деякі поширені у застосуванні методи для роботи з даними типу `TDateTime`. Слід звернути увагу на синтаксис при використанні методів:

<змінна>.<метод>;

Наприклад: `d.TimeString();`

Таблиця 10.4

Деякі методи роботи з датою та часом

Методи	Опис
`CurrentDate()`	Повертає поточну дату
`CurrentTime()`	Повертає поточний час
`CurrentDateTime()`	Повертає поточні дату і час
`operator int`	Перетворює `TDateTime` на ціле число
`operator -` `operator +`	Дозволяють зі значення дати (часу) віднімати, додавати дати, числа типу `double` чи `int`
`DateString()`	Перетворює дату `TDateTime` на `AnsiString` рядок
`TimeString()`	Перетворює час `TDateTime` на `AnsiString` рядок

У табл. 10.5 в алфавітному порядку подано функції заголовних файлів DateUtils.hpp та SysUtils.hpp, поширені у застосовуванні при роботі з даними типу дати і часу.

Таблиця 10.5

Функції роботи з датою і часом

Функції	Опис
int **CompareDate** (TDateTime A, TDateTime B);	Порівнює дві дати A та B і повертає: значення 0, якщо дати є однакові; –1, якщо значення A є менше за B; 1, якщо значення A є більше за B
int **CompareDateTime** (TDateTime A, TDateTime B);	Порівнює два значення дати і часу A та B типу TDateTime і повертає значення, подібне до результату функції CompareDate()
int **CompareTime** (TDateTime A, TDateTime B);	Порівнює два значення часу A та B типу TDateTime і повертає значення, подібне до результату у функції CompareDate()
Word **CurrentYear**();	Повертає поточний рік (чотири цифри)
TDateTime **Date**();	Повертає поточну системну дату
Word **DayOf**(TDateTime dt);	Повертає номер дня місяця дати dt (значення від 1 до 31). Функція є подібною до DayOfTheMonth()
Word **DayOfTheMonth** (TDateTime dt);	Повертає номер дня місяця дати dt. Функція є подібною до функції DayOf()
Word **DayOfTheWeek** (TDateTime dt);	Визначає по dt день тижня від 1 (понеділок) до 7 (неділя)
int **DayOfWeek** (TDateTime dt);	Визначає по dt день тижня від 1 (неділя) до 7 (субота)
int **DaysBetween** (TDateTime Now, TDateTime Athen);	Повертає цілу кількість днів поміж датами Now і Athen
Word **DaysInAMonth** (Word Year, Word Month);	Повертає кількість днів у місяці Month року Year
Word **DayOfTheYear** (TDateTime dt);	Повертає кількість днів поміж датою dt і 31 грудня попереднього року
Word **DaysInYear** (TDateTime dt);	Повертає кількість днів зазначеного у даті dt року
double **DaySpan** (TDateTime Now, TDateTime Athen)	Повертає дійсне значення кількості днів поміж Now і Athen (з урахуванням частини години)
DecodeDate (TDateTime Date, Word &Year, Word &Month, Word &Day);	Виокремлює з дати Date рік Year, місяць Month і день Day
DecodeDateDay (TDateTime Date, Word &Year, Word &DayOfYear);	Виокремлює з Date рік Year, і кількість днів DayOfYear цього року

Продовження табл. 10.5

Синтаксис функції	Опис
bool **DecodeDateFully** (TDateTime Date, Word &Year, Word &Month, Word &Day, Word &DayOfWeek);	Виокремлює з Date рік Year, місяць Month, день Day і день тижня DayOfWeek; якщо рік високосний, повертає true
DecodeDateMonthWeek (TDateTime Date, Word &Year, Word &Month, Word &WeekOfMonth, Word &DayOfWeek);	Виокремлює з дати Date рік Year, місяць Month, тиждень місяця WeekOfMonth, день тижня DayOfWeek (1 – понеділок)
DecodeDateTime(TDateTime dt, Word &Year, Word &Month, Word &Day, Word &Hour, Word &Minute, Word &Second, Word &MSec);	Виокремлює з dt рік Year, місяць Month, день Day, годину Hour, хвилину Minute, секунду Second, мілісекунду MSec
DecodeDateWeek (TDateTime Date, Word &Year, Word &WeekOfYear, Word &DayOfWeek);	Виокремлює з дати Date рік Year, тиждень року WeekOfYear і день тижня DayOfWeek (1 – понеділок)
DecodeDayOfWeekInMonth (TDateTime Date, Word &Year, Word &Month, Word &NthDayOfWeek, Word &DayOfWeek);	Виокремлює з Date рік Year, місяць Month, котрий раз NthDayOfWeek зустрічається в місяці цей день тижня, день тижня DayOfWeek (1 – понеділок)
DecodeTime(TDateTime Time, Word &Hour, Word &Min, Word &Sec, Word &MSec);	Виокремлює з Time години Hour, хвилини Min, секунди Sec і мілісекунди Msec
TDateTime **EncodeDate** (Word Year, Word Month, Word Day);	Формує значення типу TDateTime по року Year, місяцю Month і дню Day
TDateTime **EncodeDateDay** (Word AYear, Word DayOfYear);	Формує значення TDateTime по року Year і дню року DayOfYear
TDateTime **EncodeDateMonthWeek** (Word Year, Word Month, Word WeekOfMonth, Word DayOfWeek);	Формує значення TDateTime по року Year, місяцю Month, тижню місяця WeekOfMonth і дню тижня DayOfWeek
TDateTime **EncodeDateTime** (Word Year, Word Month, Word Day, Word Hour, Word Minute, Word Sec, Word MSec);	Формує значення типу TDateTime по року Year, місяцю Month, дню Day, годині Hour, хвилині Minute, секунді Sec, мілісекунді MSec
TDateTime **EncodeDateWeek** (Word Year, Word WeekOfYear, Word DayOfWeek);	Формує значення TDateTime по року Year, дню тижня DayOfWeek і тижню року WeekOfYear
TDateTime **EncodeDayOfWeekInMonth** (Word Year, Word Month, Word NthDayOfWeek, Word DayOfWeek);	Формує значення типу TDateTime по року Year, місяцю Month, дню тижня DayOfWeek і номера, котрий раз цей день тижня зустрічається в цьому місяці NthDayOfWeek

Продовження табл. 10.5

Синтаксис функції	Опис
TDateTime **EncodeTime** (Word Hour, Word Min, Word Sec, Word MSec)	Формує час типу TDateTime по заданим значенням години Hour, хвилини Min, секунди Sec, милісекунди MSec
TDateTime **EndOfADay**(Word Year, Word DayOfYear);	Повертає час завершення заданого дня року DayOfYear певного року Year
TDateTime **EndOfADay**(Word Year, Word Month, Word Day);	Повертає час завершення дня Day місяця Month року Year
TDateTime **EndOfAMonth** (Word Year, Word Month);	Повертає час завершення останнього дня місяця Month року Year
TDateTime **EndOfAWeek** (Word Year,Word WeekOfYear, Word DayOfWeek);	Повертає час завершення дня тижня DayOfWeek тижня року WeekOfYear року Year
TDateTime **EndOfAYear** (Word AYear);	Повертає час завершення року Year
TDateTime **EndOfTheDay** (TDateTime Date);	Повертає час завершення дати Date
TDateTime **EndOfTheMonth** (TDateTime Date);	Повертає час завершення місяця зазначеної дати Date
TDateTime **EndOfTheWeek** (TDateTime Date);	Повертає час завершення тижня зазначеної дати Date
TDateTime **EndOfTheYear** (TDateTime Date);	Повертає час завершення року зазначеної дати Date
Word **HourOf**(TDateTime dt);	Повертає значення години (від 0 до 23) заданого часу dt. Функція є подібною до HourOfTheDay()
Word **HourOfTheDay** (TDateTime dt);	Повертає значення години (від 0 до 23) заданого часу dt. Функція є подібною до HourOfDay()
Word **HourOfTheMonth** (TDateTime dt);	Повертає кількість годин від початку місяця зазначеної дати dt
Word **HourOfTheWeek** (TDateTime dt);	Повертає кількість годин від початку тижня зазначеної дати dt
Word **HourOfTheYear** (TDateTime dt);	Повертає кількість годин від початку року зазначеної дати dt
int **HoursBetween** (TDateTime Now, TDateTime Athen)	Повертає цілу кількість годин поміж Now і Athen
double **HourSpan** (TDateTime Now, TDateTime Athen)	Повертає дійсне значення кількості годин поміж Now і Athen (з урахуванням частини години)
void **IncAMonth**(Word &Year, Word &Month, Word &Day, int NumbOfMonths);	Збільшує значення року, місяця і дня на задану кількість місяців NumbOfMonths; значення NumbOfMonths може бути від'ємним
TDateTime **IncDay** (TDateTime dt, int NumbOfDays);	Збільшує дату dt на кількість днів NumbOfDays; значення NumbOfDays може бути і від'ємним

Продовження табл. 10.5

Синтаксис функції	Опис
TDateTime **IncHour** (TDateTime dt, int NumbOfHours);	Збільшує дату і час dt на кількість годин NumbOfHours. Значення NumbOfHours може бути і від'ємним
TDateTime **IncMilliSecond** (TDateTime dt, int NumbOfMSec);	Збільшує дату і час dt на кількість NumbOfMSec. Значення NumbOfMSec може бути і від'ємним
TDateTime **IncMinute** (TDateTime dt, int NumbOfMinutes);	Збільшує дату і час dt на кількість хвилин NumbOfMinutes. Значення NumbOfMinutes може бути і від'ємним
TDateTime **IncMonth** (TDateTime dt, int NumbOfMonths);	Збільшує дату dt на кількість місяців NumbOfMonths. Значення NumbOfMonths може бути і від'ємним
TDateTime **IncSecond** (TDateTime dt, int NumbOfSec);	Збільшує дату і час dt на кількість секунд NumbOfSec. Значення NumbOfSec може бути і від'ємним
TDateTime **IncWeek** (TDateTime dt, int NumbOfWeeks);	Збільшує дату dt на кількість тижнів NumbOfWeeks. Значення NumbOfWeeks може бути і від'ємним
TDateTime **IncYear** (TDateTime dt, int NumbOfYears);	Збільшує дату dt на кількість років NumbOfYears. Значення NumbOfYears може бути і від'ємним
bool **IsInLeapYear** (TDateTime Date);	Визначає належність дати Date до високосного року
bool **IsLeapYear** (Word Year);	Визначає належність року Year до високосного року
bool **IsPM**(TDateTime dt);	Визначає належність часу dt до другої половини дня
bool **IsSameDay**(TDateTime dt1, TDateTime dt2);	Повертає true, якщо обидві дати dt1 і dt2 (день, місяць і рік) збігаються
bool **IsToday** (TDateTime dt);	Повертає true, якщо дата dt є сьогоднішнім днем
bool **IsValidDate** (Word Year, Word Month, Word Day);	Визначає допустимість значень року, місяця та дня і повертає true, якщо Year належить діапазону від 1 до 9999 , Month – діапазону від 1 до 12, Day – діапазону від 1 до кількості днів цього місяця; інакше повертає false
bool **IsValidDateDay** (Word Year, Word DayOfYear);	Подібно до функції IsValidDate() визначає допустимість значень року Year і дня року DayOfYear
bool **IsValidDateMonthWeek** (Word Year, Word Month, Word WeekOfMonth, Word DayOfWeek);	Подібно до функції IsValidDate() визначає допустимість значень року Year, місяця Month, номера тижня цього місяця WeekOfMonth і дня тижня DayOfWeek (від 1 до 7)

Продовження табл. 10.5

Синтаксис функції	Опис
`bool` **`IsValidDateTime`** `(Word Year, Word Month, Word Day, Word Hour, Word Minute, Word Sec, Word MSec);`	Визначає допустимість значень року, місяця, дня, години, хвилини, секунди та мілісекунди і повертає `true` якщо `Year` належить діапазону від 1 до 9999 , `Month` – від 1 до 12, `Day` – від 1 до кількості днів цього місяця, години `Hour` – від 0 до 24 (якщо `Hour` є 24, то `Minute`, `Sec` та `MSec` мають бути 0), `Minute` та `Sec` – від 0 до 59, `MSec` – від 0 до 999; інакше повертає `false`
`bool` **`IsValidDateWeek`** `(Word Year, Word WeekOfYear, Word DayOfWeek);`	Подібно до функції `IsValidDate()` визначає допустимість значень року `Year`, номера тижня цього року `WeekOfYear` і дня тижня `DayOfWeek` (від 1 до 7)
`bool` **`IsValidTime`**`(Word Hour, Word Minute, Word Second, Word MSec);`	Подібно до функції `IsValidDateTime()` визначає допустимість параметрів часу
`Word` **`MilliSecondOf`** `(TDateTime dt);`	Виокремлює значення мілісекунд (в межах від 0 до 999) заданого часу `dt`
`unsigned` **`MilliSecondOfTheDay`** `(TDateTime Time);`	Повертає кількість мілісекунд, які сплинули від початку цього дня до часу `Time`
`unsigned` **`MilliSecondOfTheHour`** `(TDateTime Time);`	Повертає кількість мілісекунд, які сплинули від початку години заданого часу `Time`
`unsigned` **`MilliSecondOfTheMinute`** `(TDateTime Time);`	Повертає кількість мілісекунд, які сплинули від початку хвилини заданого часу `Time`
`unsigned` **`MilliSecondOfTheMonth`** `(TDateTime dt);`	Повертає кількість мілісекунд, які сплинули від початку місяця заданої дати `dt`
`unsigned` **`MilliSecondOfTheSecond`** `(TDateTime Time);`	Повертає кількість мілісекунд, які сплинули від початку секунди заданого часу `Time`
`unsigned` **`MilliSecondOfTheWeek`** `(TDateTime dt);`	Повертає кількість мілісекунд, які сплинули від початку тижня заданої дати `dt`
`int` **`MilliSecondOfTheYear`** `(TDateTime dt);`	Повертає кількість мілісекунд, які сплинули від початку року заданої дати `dt`
`int` **`MilliSecondsBetween`** `(TDateTime Now, TDateTime Athen);`	Повертає цілу кількість мілісекунд поміж `Now` і `Athen`
`double` **`MilliSecondSpan`** `(TDateTime Now, TDateTime Athen);`	Повертає дійсне значення кількості мілісекунд поміж `Now` і `Athen` (з урахуванням частини мілісекунди)
`Word` **`MinuteOf`** `(TDateTime Time);`	Виокремлює значення хвилин заданого часу `Time` (значення від 0 до 59)

Продовження табл. 10.5

Синтаксис функції	Опис
`Word` **`MinuteOfTheDay`** `(TDateTime dt);`	Повертає кількість хвилин, які сплинули від початку дня до заданого значення `dt`
`Word` **`MinuteOfTheHour`** `(TDateTime Time);`	Аналогічно до `MinuteOf()` повертає кількість хвилин, які сплинули від початку години заданого часу `Time` (значення від 0 до 59)
`Word` **`MinuteOfTheMonth`** `(TDateTime dt);`	Повертає кількість хвилин, які сплинули від початку місяця заданої дати `dt`
`Word` **`MinuteOfTheWeek`** `(TDateTime dt);`	Повертає кількість хвилин, які сплинули від початку тижня заданої дати `dt`
`unsigned` **`MinuteOfTheYear`** `(TDateTime dt);`	Повертає кількість хвилин, які сплинули від початку року заданої дати `dt`
`int` **`MinutesBetween`** `(TDateTime Now, TDateTime Athen);`	Повертає цілу кількість хвилин поміж `Now` і `Athen`
`double` **`MinuteSpan`** `(TDateTime Now, TDateTime Athen);`	Повертає дійсне значення кількості хвилин поміж `Now` і `Athen` (з урахуванням частини хвилини)
`Word` **`MonthOf`** `(TDateTime dt);`	Повертає номер місяця заданої дати `dt` (значення від 1 до 12)
`Word` **`MonthOfTheYear`** `(TDateTime dt);`	Аналогічно до `MonthOf()` повертає кількість місяців від початку року заданої дати `dt` (від 1 до 12)
`int` **`MonthsBetween`** `(TDateTime Now, TDateTime Athen);`	Повертає цілу кількість місяців поміж датами `Now` і `Athen`
`double` **`MonthSpan`** `(TDateTime Now, TDateTime Athen);`	Повертає дійсне значення кількості місяців поміж датами `Now` і `Athen` (з урахуванням частини року)
`TDateTime` **`Now`**`();`	Повертає поточні системні дату і час
`Word` **`NthDayOfWeek`** `(TDateTime dt);`	Визначає для дати `dt` порядковий номер цього дня тижня у цьому місяці
`TDateTime` **`RecodeDate`** `(TDateTime dt, Word Year, Word Month, Word Day);`	Змінює дату `dt` на нові значення року `Year`, місяця `Month`, дня `Day`. Значення року має бути в межах від 1 до 9999, місяць – від 1 до 12, день – не перевищувати кількості днів відповідного місяця
`TDateTime` **`RecodeDateTime`** `(TDateTime dt, Word Year, Word Month, Word Day, Word Hour, Word Minute, Word Second, Word MSec);`	Змінює дату і час `dt` на нові значення року `Year`, місяця `Month`, дня `Day`, години `Hour`, хвилини `Minute`, секунди `Second` та мілісекунди `MSec`
`TDateTime` **`RecodeDay`** `(TDateTime dt, Word Day);`	Змінює день дати `dt` на значення `Day`. Можливе значення `Day` має не перевищувати кількості днів відповідного місяця

Продовження табл. 10.5

Синтаксис функції	Опис
`TDateTime` **`RecodeHour`** `(TDateTime dt, Word Hour);`	Змінює годину заданого часу `dt` на значення `Hour`. Нове значення `Hour` має перебувати в межах від 0 до 24
`TDateTime` **`RecodeMilliSecond`** `(TDateTime dt, Word MSec);`	Змінює мілісекунду заданого часу `dt` на значення `MSec`. Нове значення `MSec` має бути в межах від 1 до 999
`TDateTime` **`RecodeMinute`** `(TDateTime dt, Word Minute);`	Змінює хвилину заданого часу `dt` на значення `Minute`. Нове значення `Minute` має бути в межах від 0 до 59
`TDateTime` **`RecodeMonth`** `(TDateTime dt, Word Month);`	Змінює місяць дати `dt` на нове значення `Month`. Нове значення `Month` має бути в межах від 1 до 12
`TDateTime` **`RecodeSecond`** `(TDateTime dt, Word Sec);`	Змінює секунду заданого часу `dt` на значення `Sec`. Нове значення `Sec` має бути в межах від 0 до 59
`TDateTime` **`RecodeTime`** `(TDateTime dt, Word Hour, Word Minute, Word Second, Word MSec);`	Змінює час `dt` на нові значення години `Hour`, хвилини `Minute`, секунди `Second` та мілісекунди `MSec`
`TDateTime` **`RecodeYear`** `(TDateTime dt, Word Year);`	Змінює рік дати `dt` на нове значення `Year`. Нове значення `Year` має бути в межах від 1 до 9999
`void` **`ReplaceDate`** `(TDateTime &dt, TDateTime NewDate);`	Змінює дату `dt` на нове значення `NewDate`; час залишається без змін
`void` **`ReplaceTime`** `(TDateTime &dt, TDateTime NewTime);`	Змінює час `dt` на нове значення `NewTime`; дата залишається без змін
`bool` **`SameDate`**`(TDateTime d1, TDateTime d2);`	Повертає `true`, якщо обидві дати `d1` і `d2` (день, місяць та рік) збігаються; значення часу ігнорується
`bool` **`SameDateTime`** `(TDateTime dt1, TDateTime dt2);`	Повертає `true`, якщо обидва значення `dt1` та `dt2` (рік, місяць, день, година, хвилина, секунда і мілісекунда) збігаються
`bool` **`SameTime`** `(TDateTime t1, TDateTime t2);`	Повертає `true`, якщо всі параметри обох значень часу `t1` і `t2` (година, хвилина, секунда і мілісекунда) збігаються; значення дат ігнорується
`Word` **`SecondOf`** `(TDateTime t);`	Повертає значення секунд заданого часу `t`. Функція є подібною до `SecondOfTheMinute()`
`unsigned` **`SecondOfTheDay`** `(TDateTime dt);`	Повертає кількість секунд, що сплили від початку дня до часу `dt`
`Word` **`SecondOfTheHour`** `(TDateTime dt);`	Повертає кількість секунд, що сплили від початку години заданого часу `dt`

Продовження табл. 10.5

Синтаксис функції	Опис
`Word` **`SecondOfTheMinute`** `(TDateTime dt);`	Повертає кількість секунд, що сплили від початку хвилини заданого часу `dt`. Функція є подібною до `SecondOf()`
`unsigned` **`SecondOfTheMonth`** `(TDateTime dt);`	Повертає кількість секунд, що сплили від початку першого місяця заданої дати `dt`
`unsigned` **`SecondOfTheWeek`** `(TDateTime dt);`	Повертає кількість секунд, що сплили від початку тижня заданої дати `dt`
`unsigned` **`SecondOfTheYear`** `(TDateTime dt);`	Повертає кількість секунд, що сплили від початку року заданої дати `dt`
`int` **`SecondsBetween`** `(TDateTime Now, TDateTime Athen)`	Повертає цілу кількість секунд поміж `Now` і `Athen`
`double` **`SecondSpan`** `(TDateTime Now, TDateTime Athen)`	Повертає дійсне значення кількості секунд поміж `Now` і `Athen` (з урахуванням частини секунди)
`TDateTime` **`StartOfADay`** `(Word Year, Word Month, Word Day);`	Формує значення типу TDateTime на початок дня `Day` місяця `Month` року `Year`
`TDateTime` **`StartOfAMonth`** `(Word Year, Word Month);`	Формує значення типу `TDateTime` на початок першого дня місяця `Month` року `Year`
`TDateTime` **`StartOfAWeek`** `(Word Year, Word WeekOfYear,Word DayOfWeek);`	Формує значення типу `TDateTime` на початок дня тижня `DayOfWeek` тижня року `DayOfWeek` року `Year`
`TDateTime` **`StartOfAYear`** `(Word Year);`	Формує значення типу `TDateTime` на початок року `Year`
`TDateTime` **`StartOfTheDay`** `(TDateTime dt);`	Формує значення типу `TDateTime` на початок зазначеного дня `dt`
`TDateTime` **`StartOfTheMonth`** `(TDateTime dt);`	Формує значення типу `TDateTime` на початок місяця заданої дати `dt`
`TDateTime` **`StartOfTheWeek`** `(TDateTime dt);`	Формує значення типу `TDateTime` на початок тижня заданої дати `dt`
`TDateTime` **`StartOfTheYear`** `(TDateTime dt);`	Формує значення типу `TDateTime` на початок року заданої дати `dt`
`TDateTime` **`Time`**`();`	Повертає поточний системний час
`TDateTime` **`Today`**`();`	Повертає поточну системну дату
`TDateTime` **`Tomorrow`**`();`	Повертає дату завтрашнього дня
`bool` **`TryEncodeDate`**`(Word Year, Word Month, Word Day, TDateTime &Date);`	Перетворює рік `Year`, місяць `Month`, день `Day` на дату `Date`. Повертає `false` у випадку помилки
`bool` **`TryEncodeTime`**`(Word Hour, Word Minute, Word Sec, Word MSec, TDateTime &Time);`	Перетворює годину `Hour`, хвилину `Minute`, секунду `Minute` і мілісекунду `MSec` на час `Time`. Повертає `false` у випадку помилки

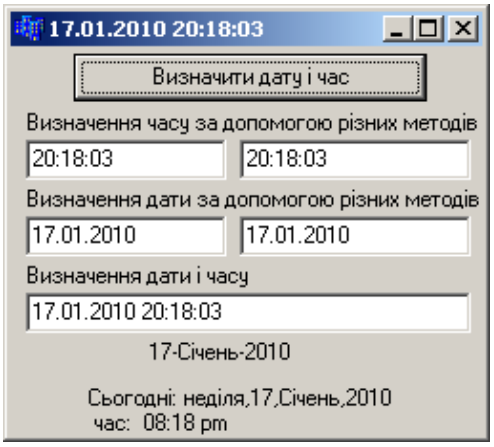
Закінчення табл. 10.5

Синтаксис функції	Опис
`bool` **`TryEncodeDateDay`** `(Word Year, Word DayOfYear, TDateTime &Date);`	Перетворює рік `Year`, день року `DayOfYear` на дату `Date`. Повертає `false` у випадку помилки
`bool` **`TryEncodeDateMonthWeek`** `(Word Year, Word Month, Word WeekOfMonth, Word DayOfWeek, TDateTime &Date);`	Перетворює рік `Year`, місяць `Month`, тиждень місяця `WeekOfMonth`, день тижня `DayOfWeek` (1 – понеділок) на дату `Date`. Повертає `false` у випадку помилки
`bool` **`TryEncodeDateTime`** `(Word Year, Word Month, Word Day, Word Hour, Word Minute, Word Sec, Word MSec, TDateTime &dt);`	Формує `dt` типу `TDateTime` за значеннями року `Year`, місяця `Month`, дня `Day`, години `Hour`, хвилини `Minute`, секунди `Sec` та мілісекунди `MSec`. Повертає `false` у випадку помилки
`bool` **`TryEncodeDateWeek`** `(Word Year, Word WeekOfYear, TDateTime &dt, Word DayOfWeek);`	Формує `dt` типу `TDateTime` за заданими значеннями року `Year`, тижня року `WeekOfYear` і дня тижня `DayOfWeek`. Повертає `false` у випадку помилки
`bool` **`TryEncodeDayOfWeekInMonth`** `(Word Year, Word Month, Word NthDayOfWeek, Word DayOfWeek, TDateTime &dt);`	Формує `dt` типу `TDateTime` за заданими значеннями року `Year`, місяця `Month`, тижня `NthDayOfWeek` і дня тижня `DayOfWeek`. Повертає `false` у випадку помилки
`Word` **`WeekOf`** `(TDateTime dt)`	Повертає номер тижня року заданої дати `dt` (значення від 1 до 53)
`Word` **`WeekOfTheMonth`** `(TDateTime dt);`	Повертає номер тижня місяця заданої дати `dt` (значення від 1 до 6)
`Word` **`WeekOfTheYear`** `(TDateTime dt)`	Повертає номер тижня року заданої дати `dt` (від 1 до 53). Функція є подібною до функції `WeekOf()`
`int` **`WeeksBetween`**`(TDateTime Now, TDateTime Then);`	Повертає кількість тижнів поміж `Now` і `Then`
`Word` **`WeeksInAYear`**`(Word Year);`	Повертає кількість тижнів року `Year`
`Word` **`WeeksInYear`** `(TDateTime dt);`	Повертає кількість тижнів року зазначеної дати `dt`
`double` **`WeekSpan`** `(TDateTime Now, TDateTime Then);`	Повертає дійсне значення кількості тижнів поміж датами `Now` і `Athen` (з урахуванням частини тижня)
`Word` **`YearOf`**`(TDateTime dt)`	Повертає рік зазначеної дати `dt` (значення від 1 до 9999)
`int` **`YearsBetween`**`(TDateTime Now, TDateTime Athen)`	Повертає цілу кількість років поміж датами `Now` і `Athen`
`double` **`YearSpan`** `(TDateTime Now, TDateTime Athen)`	Повертає дійсне значення кількості років поміж датами `Now` і `Athen` (з урахуванням частини року)
`TDateTime` **`Yesterday`**`();`	Повертає дату вчорашнього дня

Для того щоб у програмі були доступними наведені вище функції й методи, треба директивою `#include` долучити відповідні заголовні файли `DateUtils.hpp` чи `SysUtils.hpp`. Розглянемо на прикладах роботу цих функцій і методів.

Використовуючи різні функції, наведемо кілька варіантів виведення поточної дати і часу. Нечислові результати виконання цих функцій виводяться мовою, яку визначено в налаштуваннях операційної системи. У наведеному нижче прикладі подано різні можливості форматування дати і часу, в тому числі можливі символи-розділювачі для дати і часу.

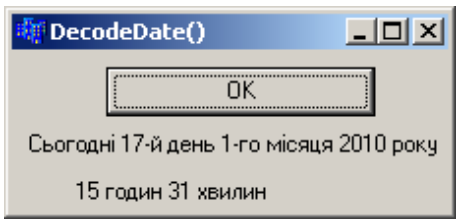
```
#include "DateUtils.hpp"
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{ TDateTime fd;
 fd=Time();
 Edit1->Text=fd.CurrentTime();
 Edit2->Text=fd.TimeString();
 fd=Date();
 Edit3->Text=fd.DateString();
 Edit4->Text=DateToStr(fd);
 Edit5->Text=DateTimeToStr(Now());
 // Можливий розділювач годин і хвилин '–'
 DateSeparator = '-';
 ShortDateFormat="dd/mmmm/yyyy";
 Label4->Caption=DateToStr(Date());
 Label5->Caption="Сьогодні: "+FormatDateTime("dddd,dd,mmmm,yyyy
' \n час:  'hh:mm am/pm", Now());
}
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{ Form1->Caption=Now();
}
```



Тут виведення поточної системної дати і часу у якості надпису форми здійснюється за допомогою компонента `Timer1` (див. стор. 40).

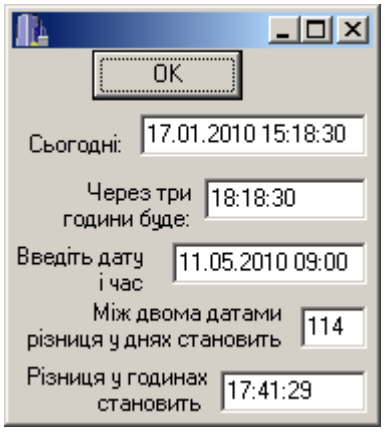
Проілюструємо роботу функцій `DecodeDate()` та `DecodeTime()` для виведення поточної дати і часу.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ Word Year, Month, Day, Hour, Min, Sec, MSec;
  TDateTime dt = Now();
  DecodeDate(dt, Year, Month, Day);
  Label1->Caption = AnsiString("Сьогодні ")+
    IntToStr(Day) + AnsiString("-й день ")+
    IntToStr(Month) + "-го місяця "+
    IntToStr(Year) + " року ";
  DecodeTime(dt, Hour, Min, Sec, MSec);
  Label2->Caption = IntToStr(Hour) +
  " годин " + IntToStr(Min) + " хвилин";
}
```

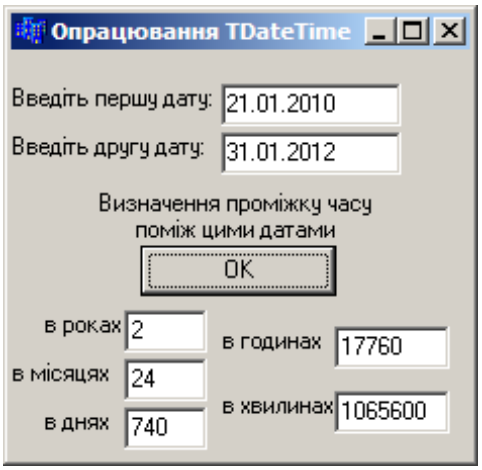


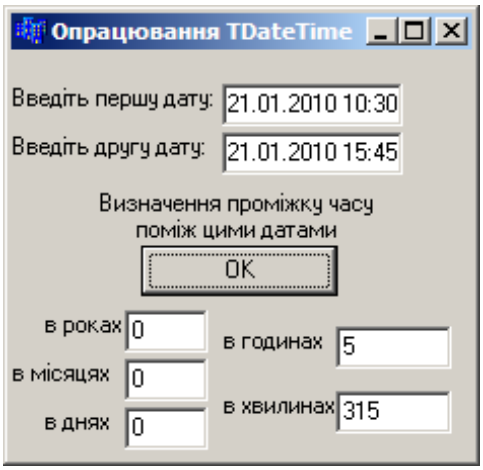
Наведемо приклади використання `operator`:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ TDateTime d1,d2,d3,d4;
  d1=Now();
  Edit1->Text=d1.DateTimeString();
  d2=d1.operator+(0.125);
  Edit2->Text=d2.TimeString();
  d3=StrToDateTime(Edit3->Text);
  int k1 = d1.operator int();
  int k2 = d3.operator int();
  Edit4->Text=IntToStr(k2-k1);
  d4=d1.operator-(d3);
  Edit5->Text=d4.TimeString();
}
```



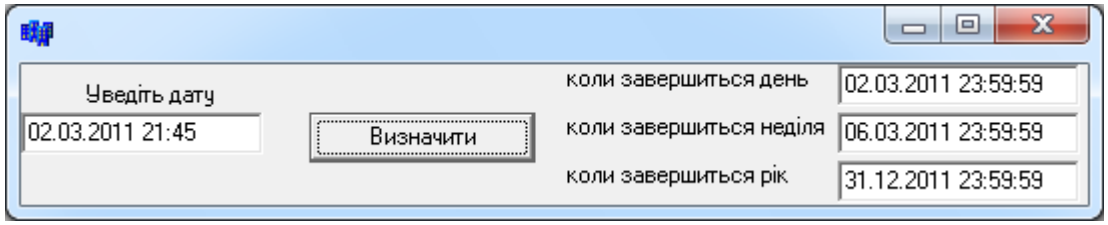
Проілюструємо роботу вищенаведених функцій на прикладі програми для визначення різниці в роках, місяцях, днях, годинах і хвилинах.





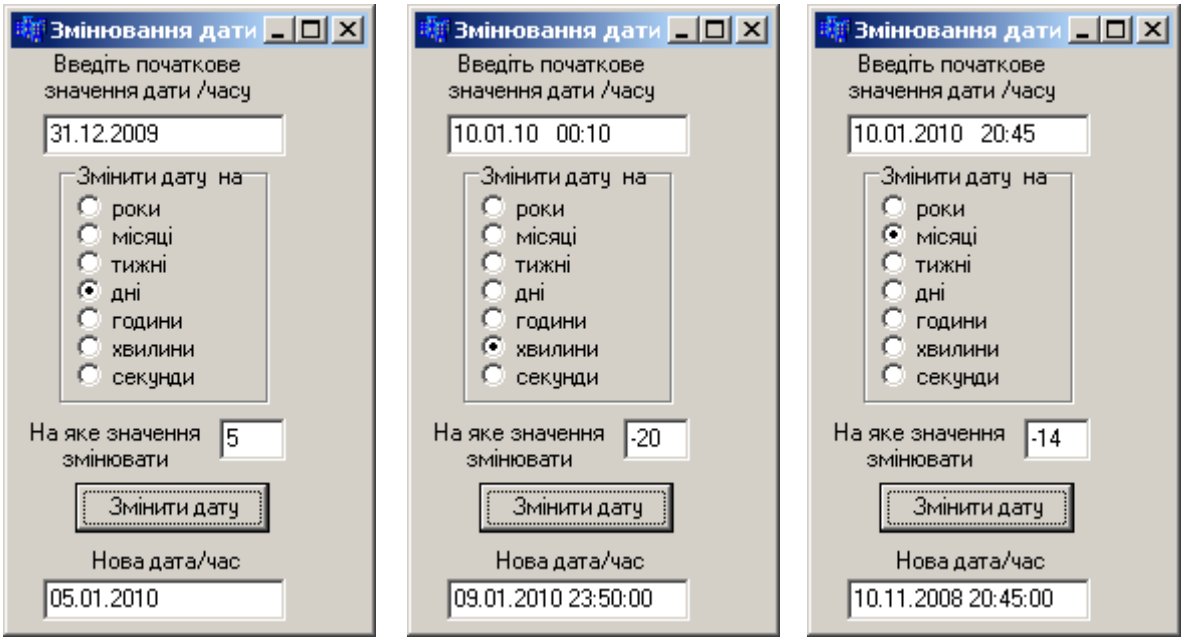

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ TDateTime d1,d2;
 d1=StrToDateTime(Edit1->Text);
 d2=StrToDateTime(Edit2->Text);
 int y,m,d,h,min,s;
 y=YearsBetween(d2,d1);      Edit3->Text=IntToStr(y);
 m=MonthsBetween(d2,d1);     Edit4->Text=IntToStr(m);
 d=DaysBetween(d2,d1);       Edit5->Text=IntToStr(d);
 h=HoursBetween(d2,d1);      Edit6->Text=IntToStr(h);
 min=MinutesBetween(d2,d1); Edit7->Text=IntToStr(min);
}
```

Наведемо приклад використання функцій для визначення кінцевого строку завершення дня, неділі, року для певної заданої дати.



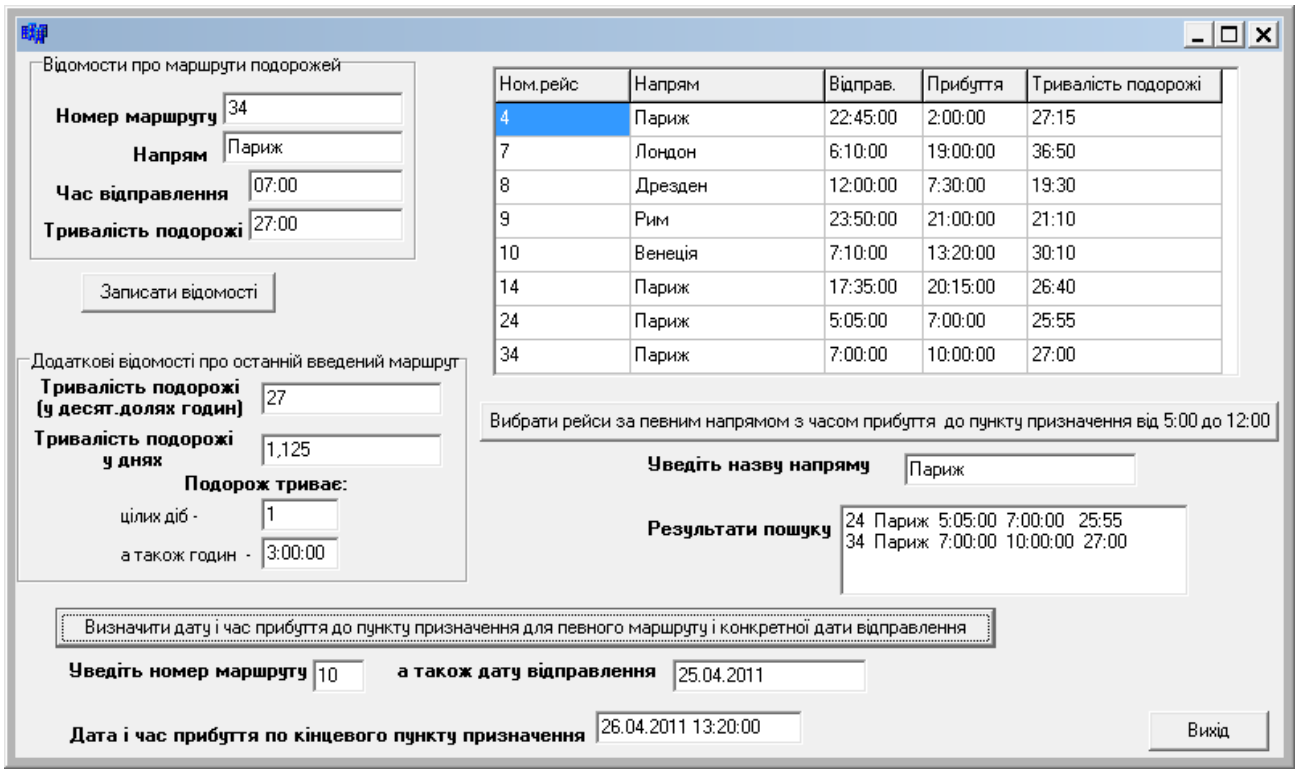
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{  TDateTime dt1,dt2,dt3,dt4;
   dt1=Edit1->Text;          // Введення заданої дати
   dt2=EndOfTheDay(dt1);      Edit2->Text=dt2;
   dt3=EndOfTheWeek(dt1);     Edit3->Text=dt3;
   dt4=EndOfTheYear(dt1);     Edit4->Text=dt4;
}
```

Наведемо ще одну програму для ілюстрації роботи функцій змінювання дати і часу на задану величину.




```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ TDateTime dt=StrToDateTime(Edit1->Text);
  int x=StrToInt(Edit2->Text);
  switch(RadioGroup1->ItemIndex)
  { case 0: dt=IncYear(dt,x);   break;
    case 1: dt=IncMonth(dt,x);  break;
    case 2: dt=IncWeek(dt,x);   break;
    case 3: dt=IncDay(dt,x);    break;
    case 4: dt=IncHour(dt,x);   break;
    case 5: dt=IncMinute(dt,x); break;
    case 6: dt=IncSecond(dt,x); break;
  }
  Edit3->Text=DateTimeToStr(dt);
}
```

Наступний приклад програми продемонструє роботу відразу декількох функцій опрацювання даних `TDateTime`.




```
#include <DateUtils.hpp>
struct avio
{ int nomr;
  char reis[20],  all[10];
  TDateTime start,last;
  double dli;
  AnsiString info()
  { AnsiString s=IntToStr(nomr)+"  "+(AnsiString)reis+"  "+
                start.TimeString()+" "+last.TimeString()+" "+
                (AnsiString)all;
    return s;
  }
};
avio w[10];
int n=0;            // Кількість маршрутів
//---------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{ SG1->Cells[0][0]="Ном.рейс";
  SG1->Cells[1][0]="Напрям";
  SG1->Cells[2][0]="Відправ.";
  SG1->Cells[3][0]="Прибуття";
  SG1->Cells[4][0]="Тривалість подорожі";
}
//---------------------------------------------------------
// Записати відомості
void __fastcall TForm1::Button1Click(TObject *Sender)
{ SG1->RowCount=n+2;
  AnsiString s1,s2;
```

```
  w[n].nomr=StrToInt(Edit1->Text);          // Увести номер маршруту,
  strcpy(w[n].reis,Edit2->Text.c_str());    // його назву,
  w[n].start=Edit3->Text;                   // час відправлення
  strcpy(w[n].all,Edit4->Text.c_str());     // та тривалість подорожі
  s1=(AnsiString)w[n].all;                  // Визначити окремо
  int n1=StrToInt(s1.SubString(1,s1.Pos(":")-1));//кількість годин
  s1.Delete(1,s1.Pos(":"));                 // та кількість
  int n2=StrToInt(s1);                      // хвилин подорожі
  w[n].last=IncHour(w[n].start,n1);         // Визначити дати і часу
  w[n].last=IncMinute(w[n].last,n2);        // прибуття
  // Тривалість подорожі у десят.долях годин
  double f=HourSpan(w[n].last,w[n].start);
  Edit5->Text=FloatToStr(f);
  double ff=DaySpan(w[n].last,w[n].start);// Кількість діб подорожі
  Edit6->Text=FloatToStr(ff);
  int d=DaysBetween(w[n].last,w[n].start);// Кількість цілих діб
  Edit7->Text=IntToStr(d);
  w[n].dli=ff;                          // Тривалість подорожі у днях
  Edit8->Text=TimeToStr(ff);            // Вивести складову годин подорожі
  if (w[n].last.operator double()>1)
    w[n].last=w[n].last-(int(w[n].last));
  SG1->Rows[n+1]->DelimitedText=w[n].info();
  n++;
}
//---------------------------------------------------------------
// Вибрати рейси за певним напрямом з часом прибуття до пункту призначення
//  від 5:00 до 12:00
void __fastcall TForm1::Button2Click(TObject *Sender)
{ int nom=0;  Memo1->Clear();
  if (Edit9->Text=="") {ShowMessage("Забули ввести!"); return;}
  AnsiString nazva=Edit9->Text;      // Ввести назву маршруту
  for (int i=0;i<n;i++)
  if ( AnsiString(w[i].reis)==nazva &&
       TTime(w[i].last)>=TTime("05:00") &&
       TTime(w[i].last)<=TTime("12:00"))
    { nom=i; Memo1->Lines->Add(w[nom].info()); }
  if (!nom) ShowMessage("Немає таких рейсів");
}
//---------------------------------------------------------------
// Визначити дату і час прибуття до пункту призначення для певного маршруту
//  і конкретної дати відправлення
void __fastcall TForm1::Button3Click(TObject *Sender)
{ int m=0;    TDateTime dt1, dt2;
  if (Edit10->Text=="" || Edit11->Text=="")
   { ShowMessage("Забули ввести!"); return;}
  int k = StrToInt(Edit10->Text);   // Ввести номер маршруту
  for (int i=0;i<n;i++)
   if (w[i].nomr==k) {m=i; break;} // Пошук потрібного маршруту
```

```
    if (!m) {ShowMessage("Немає такого маршруту"); return;}
    dt1 = Edit11->Text;                    // Ввести дату відправлення
    // Визначити дату і час прибуття до кінцевого пункту призначення
    dt2 = dt1 + TTime(w[m].start) + TTime(w[m].dli);
    Edit12->Text = dt2.DateTimeString();
}
```

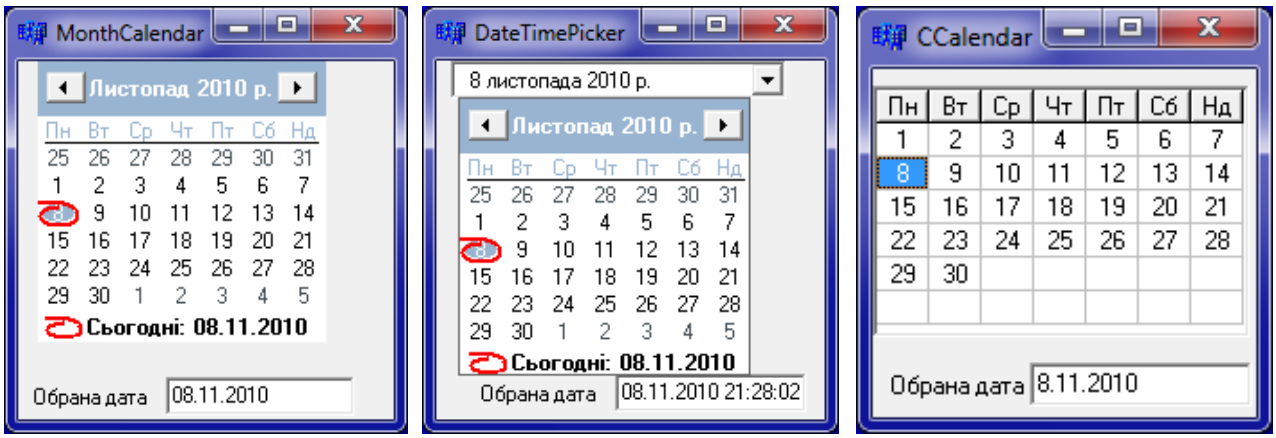
Інші приклади використання даних `TDateTime` буде наведено при вивченні файлів.

У C++ Builder для роботи з датою і часом існують спеціальні компоненти `MonthCalendar` та `DateTimePicker`, розміщені на сторінці Win32. Вони власним зовнішньою формою нагадують календар. Основна візуальна відмінність між ними полягає у тому, що компонент `DateTimePicker` у своєму звичному стані є згорнутий, а календар з'являється лише при клацанні на відповідному трикутничкові, у той час як компонент `MonthCalendar` завжди є повністю видимий на формі.

Крім того, `DateTimePicker` дозволяє безпомилково з точки зору синтаксису вводити не лише дати, як для компонента `MonthCalendar`, а й час. Властивість `Kind` визначає режим роботи: `dtkDate()` – введення дати, `dtkTime()` – введення часу, а властивість `DateFormat` задає скорочений (`dfShort`) чи повний (`dfLong`) формат відбиття дати. Але головними властивостями компонента `MonthCalendar` є `Date` та `Time`, за допомогою якої користувач зчитує уведені значення дати чи дати і часу залежно від режиму роботи. Тип цих властивостей – `TDateTime`.

Проте компонент `MonthCalendar` має свої додаткові можливості: можна дозволити множинний вибір дат у деякому діапазоні (властивість `MultiSelect`), можна виводити у календарі номера тижнів з початку року (властивість `WeekNumbers`), перебудовувати календар, задаючи перший день кожного тижня (властивість `FirstDayOfWeek`) тощо.

Крім наведених існує ще один компонент-календар `CCalendar` (закладка `Samples`), який має три окремі властивості цілого типу `Year` – рік, `Month` – місяць, `Day` – день, з якими іноді зручніше мати справу, аніж з типом `TDateTime`. Щоби цей компонент показував календар на потрібний місяць потрібного року, слід попередньо задати значення властивостей `Year` та `Month`.



```
void __fastcall TForm1::MonthCalendar1Click(TObject *Sender)
{ Edit1->Text=MonthCalendar1->Date; }
//---------------------------------------------------------
void __fastcall TForm1::DateTimePicker1CloseUp(TObject *Sender)
{ Edit1->Text=DateTimePicker1->DateTime; }
//---------------------------------------------------------
void __fastcall TForm1::CCalendar1Change(TObject *Sender)
{ Edit1->Text=IntToStr(CCalendar1->Day)+"."+
    IntToStr(CCalendar1->Month)+"."+ IntToStr(CCalendar1->Year);
}
```

Питання та завдання для самоконтролю

1) Які заголовні файли слід долучити до програмного проекту для використання функцій опрацювання дати і часу:

а) у консольному режимі; б) у додатку C++ Builder?

2) В який спосіб організовано тип `TDateTime`?

3) Наведіть функції та методи C++ Builder, які визначають поточну дату.

4) Наведіть функції та методи C++ Builder, які визначають поточний час.

5) Яка функція C++ Builder дозволяє визначити і дату і час?

6) В який спосіб можна за датою визначити номер тижня року?

7) Яка функція дозволяє за датою визначити день тижня?

8) Запишіть функцію, яка дозволяє визначити належність дати до високосного року.

9) В який спосіб можна збільшити (чи зменшити) дату:

а) на один день;

б) на один тиждень;

в) на один місяць?

10) Наведіть функцію, яка визначає належність зазначеного часу до другої половини дня.

11) Які функції дозволяють перетворити значення типу `TDateTime` на рядок?

12) Яка функція дозволяє перетворити рядок на значення типу `TDateTime` з перевіркою можливості перетворення?

13) Запишіть функцію, яка формує значення типу `TDateTime` на початок місяця заданої дати.

14) Наведіть функцію, яка дозволяє змінити на нове значення день дати.

15) Яка функція дозволяє порівняти дві дати?

16) Запишіть функцію, яка дозволяє визначити годину заданого часу.

17) Яка функція із заданої дати виокремлює значення року, місяця і дня в окремі змінні?

18) Наведіть функції, які дозволяють перетворити рік, місяць і день цілого типу на значення типу `TDateTime`.

19) В який спосіб можна перевірити правильність перетворення року, місяця і дня цілого типу на значення типу `TDateTime`?

20) Назвіть спеціальні компоненти C++ Builder для роботи з датою і часом.

Розділ 11

Типи користувача

11.1 Перейменовування типів (typedef)

C++ надає можливість моделювання нових типів даних на базі вже існуючих типів. Як приклад доцільності застосовування таких типів розглянемо оголошення

```
double A[5][7];
```

Тут оголошується змінна `A` як матриця (двовимірний масив) дійсних чисел розмірності 5 рядків і 7 стовпчиків. Якщо у програмі оголошується кілька аналогічних масивів або якщо програма містить функції, параметрами яких є матриця 5×7, зручніше є створити окремий тип "матриця" для оголошення змінної `A`, щоб уникнути помилок і покращити читабельність тексту програми. У мові C++ для створювання типів користувача на базі вже існуючих типів використовується службове слово `typedef` (від англійської "type definition" – означення типу). Отже оголошення описаного типу `matrica` може мати вигляд

```
typedef double matrica [5][7];
```

Цей тип є синонімом дійсного двовимірного масиву розмірності 5×7, і його можна використовувати як звичайний тип для оголошення змінної `A`:

```
matrica A;
```

Це оголошення означає, що змінна `A` має тип `matrica`, тобто є дійсним двовимірним масивом. Після перейменування типу можна використовувати новий тип `matrica` для оголошення всіх дійсних двовимірних масивів 5×7, наприклад:

```
matrica B, C;                 // B і C – матриці 5×7
double sumr(matrica C);       /* Заголовок функції, параметром якої є
                                 матриця 5×7  */
matrica M[4];                 // Масив з 4-х матриць
```

Порівняймо останнє оголошення з еквівалентним до нього, але менш зручним, без перейменування типу:

```
double M[4][5][7];
```

Перейменування типу у загальному вигляді має синтаксис

`typedef` *<тип> <нове_і'мя>* `[`*<розмірність>*`];`

Тут квадратні дужки є елементом синтаксису. Розмірність може бути відсутньою.

Наведемо приклади перейменовування типів:

```
typedef unsigned char byte;  /* Перейменування типу
                                "беззнаковий char" на byte  */
typedef char MS[50];         /* Перейменування типу "рядок
                                з 50-ти символів" на MS  */
```

Розглянемо ще один приклад оголошення без перейменовування:

```
char s[20];                   // Слово s – рядок з 20-ти символів,
```

```
char mas[10][20];          // mas – масив з 10 слів
```

Перейменування типу “рядок з 20-ти символів” на тип “слово”:

```
typedef char slovo [20];
slovo s;                   // Оголошення змінної s типу slovo
slovo mas[10];             // і масиву з 10-ти слів
```

Взагалі `typedef` є просто засобом спрощення запису операторів оголошення змінних.

Доволі часто визначення типів даних використовується разом зі структурами, що надає можливість створювання складних типів даних, які поєднують різнотипні характеристики. Приклади такого використання буде наведено у наступному підрозділі.

11.2 Структури (struct)

У попередніх розділах для зберігання даних розглянуто прості типи даних: числа, символи, рядки тощо. Але часто на практиці перед програмістом постає завдання запрограмувати той чи інший об’єкт, який характеризується водночас кількома параметрами, приміром, точка на площині задається парою дійсних чисел (*x*, *y*), а дані про людину можна задавати кількома параметрами: прізвище, ім’я, по-батькові – рядки, рік народження – число тощо. Для поєднання кількох параметрів в одному об’єктові в C++ існує спеціальний тип даних, який має назву структура. На відміну від масивів, які також є структурованим типом даних і містять елементи одного типу, структури дозволяють поєднувати дані різних типів.

Структура – це складений тип даних, змінні (поля) якого можуть містити різноманітну й різнотипну інформацію.

Опис структури має синтаксис

```
struct  <ім’я_типу_структури>
{ <тип1> <поле1>;
  <тип2> <поле2>;
   . . .
  <типN> <полеN>;
} <список_змінних>;
```

Ім’я типу структури задається програмістом виходячи зі змісту даних, які зберігатимуться у структурі. Список змінних наприкінці оголошення структури може бути відсутнім, у цьому разі після фігурної дужки має бути крапка з комою.

Наведемо приклад оголошення структури з ім’ям `sportsman`, яка поєднає в собі інформацію про змагання спортсменів: прізвище і вік спортсмена, країна, кількість його очок, утворюючи відповідні поля:

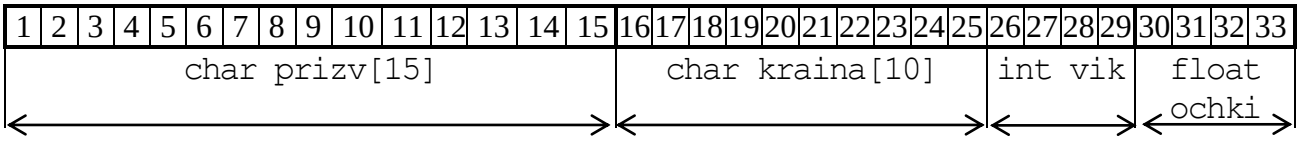
```
struct  sportsman
{ char prizv[15]; char kraina[10];    // Прізвище та країна,
  int vik;        float ochki;        // вік та кількість очок.
};
```

Як наслідок цього оголошення створено новий тип `sportsman`. Тепер у програмі цей тип може використовуватись нарівні зі стандартними типами

для оголошення змінних. Оголошені змінні типу `sportsman` матимуть чотири поля: два рядки (`prizv` і `kraina`), ціле (`vik`) і дійсне (`ochki`) числа.

```
sportsman Sp;          /* Змінна Sp типу sportsman може зберігати
                          інформацію про одного спортсмена. */
sportsman Mas[15];     /* Масив Mas типу sportsman може містити
                          інформацію про 15 спортсменів. */
```

При оголошуванні змінної типу структури пам'ять під усі поля структури виділяється послідовно для кожного поля. У наведеному прикладі структури `sportsman` під змінну `Sp` послідовно буде виділено 15, 10, 4, 4 байти – усього 33 байти. Поля змінної `Sp` у пам'яті буде розміщено у такому порядку:



Слід зазначити, що сумарну пам'ять, яка виділяється під структуру, може бути збільшено компілятором на вимогу процесора для так званого вирівнювання адрес. Реальний обсяг пам'яті, яку займає структура, можна визначити за допомогою `sizeof()`.

При оголошуванні структур їхні поля можна ініціалізовувати початковими значеннями. Наприклад, оголошення змінної `Sp` типу `sporsman` й ініціалізація її даними про 20-річного спортсмена з України за прізвищем Бойко, який набрав 75.3 очок, матиме вигляд:

```
sporsman Sp={"Бойко", "Україна", 20, 75.3};
```

Оголошення типу "точка на площині" `POINT` з полями-координатами – `x` та `y` і змінної `spot` цього типу – точки з координатами (20, 40) може бути таким:

```
struct POINT
{ int x, y;
} spot = { 20, 40 };      // Точка має координати x = 20, y = 40
```

При ініціалізації масивів структури кожен елемент масиву слід записувати у фігурних дужках, наприклад при створенні двовимірного масиву `compl` розміром 2×3 типу структури `complex` з полями `real` та `im` ініціалізація початкових значень елементів може мати вигляд:

```
struct  complex
{ float real, im;
} compl[2][3]={{{1.4}, {5}, {-3.1}},{{2}, {-5.2}, {0}}};
```

Доступ до полів структури здійснюється за допомогою операції "крапка":

<структурна змінна>`.`*<ім'я поля>*

Наприклад, для розглянутої змінної `Sp` типу `sporsman` надання значень її полям може мати вигляд:

```
strcpy(Sp.prizv, "Бойко");
strcpy(Sp.kraina, "Україна");
Sp.vik = 20;
Sp.ochki = 75.3;
```

Можна створювати тип структури за допомогою `typedef`, наприклад:

```
typedef struct
{  AnsiString priz, gruppa;
   int Bal_math, Bal_inform;
} student;                    // Оголошення типу "студент"
student W;                    // і змінної W цього типу.
```

Тут означено тип структури за ім'ям `student` для зберігання даних про успішність студентів з полями: прізвище, група, екзаменаційні оцінки з математики та інформатики й оголошено змінну `W` цього типу.

Структури можуть розміщуватись у динамічній пам'яті за допомогою оператора `new` і оголошуватися як покажчики. У такому разі доступ до полів структури здійснюється за допомогою операції вибору – "стрілка" (->, на клавіатурі послідовно натискаються клавіші "мінус" і "більше"):

```
sportsman *p=new sportsman;     // Виділення пам'яті,
p->ochki=95.3;                  // присвоєння очок полю ochki
(*p).vik=23;                    // і віку полю vik.
```

Зверніть увагу на те, що якщо `p` – покажчик на структуру, то `(*p)` – сама структура і доступ до її полів здійснюється за допомогою крапки. Детальніше динамічні структури розглянуті в розд. 13.

Якщо змінні типу "структура" оголошуються у програмі лише один раз, то, зазвичай, їх записують наприкінці оголошення структури і при цьому ім'я структури можна явно не зазначати, наприклад:

```
struct
{ char prizv[15];
  char kraina[10];
  int vik;  float ochki;
}
Sp;
```

Тут оголошено змінну `Sp` як структуру з чотирма полями без створення типу "спортсмен".

Поля структури можуть бути якого завгодно типу, у тому числі й масивом, покажчиком чи структурою, окрім типу тієї ж самої структури (але можуть бути покажчиком на неї). За приклад створимо дві структури для опрацювання інформації про співробітників у відділі кадрів: прізвище, ім'я, по-батькові співробітника, домашня адреса, домашній телефон, дата народження, дата вступу на роботу, зарплатня.

```
struct date
{ int day, month, year;              // День, місяць та рік
};
struct person
{ char prizv[20], adresa[150];       // Прізвище, адреса,
  char phone[10]; date birth_date;   // телефон, дата народження,
  date work_date;                    // дата вступу на роботу
  float salary;                      // та зарплатня.
};
```

У цьому прикладі оголошується новий тип – структура `date` для зберігання дати (день, місяць, рік). Окрім того, оголошується тип – структура `person`, яка використовує тип `date` для полів `birth_date` і `work_date`. Посилання на поля вкладеної структури формується з імені структурної змінної, імені структурного поля й імені поля вкладеної структури. Те, що поля структур самі можуть бути структурами, надає можливість цілісно описувати доволі складні об'єкти реального світу. Оскільки структури, на відміну від масивів, зберігають дані різних типів, їх відносять до неоднорідних типів даних.

Оголосимо змінну створеного типу `person`:

```
person P;
```

Присвоювання прізвища у змінну `P`:

```
strcpy(P.prizv, "Шкуропат");
```

Присвоювання дати народження 5 березня 1987 року у змінну `P`:

```
P.birth_date.day=5;
P.birth_date.month=3;
P.birth_date.year=1987;
```

Розмістимо змінну `man` типу `person` у динамічній пам'яті:

```
person *man = new person;
```

І присвоїмо ті ж самі значення:

```
strcpy(man->prizv, "Шкуропат");
(man->birth_date).day=5;
(man->birth_date).month=3;
(man->birth_date).year=1987;
```

Оголосимо масив з даними про 100 осіб:

```
person P[100];
```

Попередні присвоювання для четвертої людини:

```
strcpy(P[3].prizv, "Шкуропат");
P[3].birth_date.day=5;
P[3].birth_date.month=3;
P[3].birth_date.year=1987;
```

Уведемо значення всіх полів для всіх співробітників до масиву:

```
for(i=0; i<100; i++)
{ cout<<"Введіть призвіще: "
  gets(P[i].prizv);
  cout<<"Введіть адресу: "
  gets(P[i].adresa);
  cout<<"Введіть телефон: "
  gets(P[i].phone);
  cout<<"Введіть дату народження: "<<endl;
  cout<<"День: "; cin>>P[i].birth_date.day;
  cout<<"Місяць: "; cin>>P[i].birth_date.month;
  cout<<"Рік: " cin>>P[i].birth_date.year;
  cout<<"Введіть дату вступу до роботи: "endl;
  cout<<"День: "; cin>>P[i].work_date.day;
```

```
  cout<<"Місяць: "; cin>>P[i]. work_date.month;
  cout<<"Рік: " cin>>P[i]. work_date.year;
}
```

Структурні змінні одного типу можна переприсвоювати одну одній, при цьому виконується присвоювання усіх полів, наприклад:

```
person man=P[0];
```

Як вже було зазначено вище, поля структури можуть бути якого завгодно типу, у тому числі масивами. Наприклад, оголошення структури "погода за місяць" може мати вигляд

```
struct pogoda
{ int month;    // Номер місяця,
  int temp[31];// масив температур за 31 день.
};
pogoda M;       // Оголошення змінної M типу pogoda (погода за один місяць).
pogoda G[12];   // і масиву G  типу pogoda (погода за рік).
```

Визначимо значення температури повітря за 14 липня:

```
M.month = 7;
M.temp[13] = 27;
```

Присвоїмо випадкові значення температури за весь лютий:

```
M.month = 2;
for(int i=0; i<28; i++)  M.temp[i] = random(5)-10;
```

Виведення температури повітря восени (за 3 місяці):

```
for(int i=9; i<=11; i++)
{ G[i-1].month = i;
  for(int j=0; j<31; j++)
   if(j==30 && (i==9 || i==11))
                              continue;  // Пропустити 31.09 та 31.11
    else
     cout << G[i-1].temp[j] << " ";
  cout << endl;
}
```

У C++ структура при описуванні, окрім звичних полів, може містити елементи-функції. Наприклад, можна дописати до структури `pogoda` функцію `month_name`, яка визначатиме рядок – назву місяця за номером.

```
struct pogoda
{ int month, temp[31]; // Номер місяця і масив температур за 31 день.
  char *month_name()
  { switch (month)
    { case 1: return "Січень";
      case 2: return "Лютий";
      case 3: return "Березень";
      case 4: return "Квітень";
      case 5: return "Травень";
      case 6: return "Червень";
      case 7: return "Липень";
      case 8: return "Серпень";
```

```
        case 9: return "Вересень";
        case 10: return "Жовтень";
        case 11: return "Листопад";
        case 12: return "Грудень";
        default: return "Помилковий номер";
    } }
};
```

У програмі цю функцію можна викликати в такий спосіб:

```
pogoda M;
 . . .
cin >> M.month ;
cout << M.month_name();
```

Після виконання цих операторів на екран буде виведено рядок, утворений за допомогою функції `month_name()`.

Наведемо ще один приклад елемента-функції. Зорганізуємо структуру `student` з функцією `Show()`, яка будуватиме рядок типу `AnsiString`, поєднуючи усі поля цієї структури. Надалі цей рядок можна буде використовувати для виведення записів на екран:

```
struct student
{ AnsiString prizv, grupa;
  int Bal_math, Bal_inform;
  AnsiString Show()
   { return  prizv+" "+ grupa+" "+IntToStr(Bal_math)+"  "+
                                          IntToStr(Bal_inform);
   }
};
```

У програмі цю функцію можна викликати в такий спосіб:

```
student z;
 . . . .
Memo1->Lines->Add(z.Show());
```

Після виконання цих операторів до компонента `Memo1` буде виведено рядок, утворений за допомогою функції `Show()`. Отже, звертання до елементів-функцій відбувається так само, як і до звичних полів. Наявність функцій у структурі може істотно спростити програму.

До функцій структури передаються як звичайні змінні.

Наведемо приклад консольної програми, в якій організовано дві функції, параметрами яких є структури.

Приклад 11.1 Створити програму для опрацювання інформації про результати сесії студентів: прізвище студента, курс, група і результати п'яти екзаменів. У програмі передбачити можливість введення і виведення даних та переведення студентів, які склали сесію, на наступний курс, змінивши відповідно з цим номер групи.

Розв'язок. У наведеному нижче програмному коді зорганізовано дві функції: `print()` та `func()`. Параметром функції `print()` є структура, поля якої

почергово виводяться на екран консолі. Функція `func()` одержує посилання на структуру, що надає їй можливість редагувати дані структури. Ця функція перевіряє оцінки студента, і, якщо зустрічається оцінка нижче за 60 балів, функція перериває своє виконування. Якщо всі оцінки студента були не нижче за 60 балів, підвищується значення курсу. Зазвичай назва групи складається з абревіатури спеціальності, дефіса, курсу й номера групи. Для студента, якого буде переведено до наступного курсу, у назві групи слід змінити номер курса. Для цього слід віднайти позицію дефіса і замінити символ, який слідує за дефісом, на нове значення курсу. Позицію дефіса може бути обчислено як різницю покажчиків на віднайдений дефіс і початок рядка `S.gr`. Для перетворювання номера курсу на рядок, використовується функція `atoi()`, яка розглядалась у розділі 7.

Текст програми:

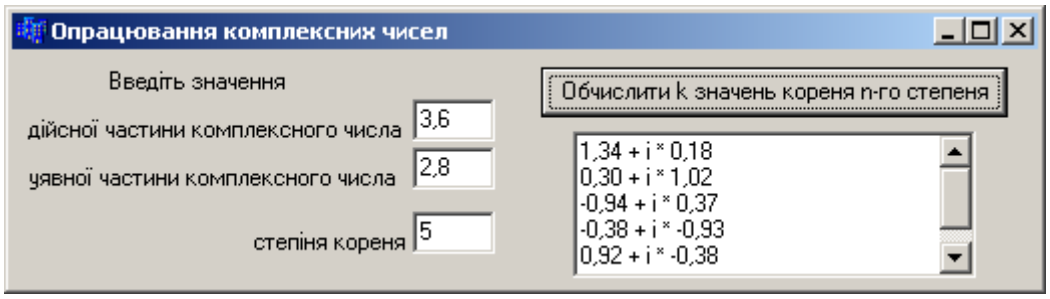
```
const int N=2;
struct student                  // Оголошення типу структури з полями:
{ char prizv[10];               // прізвище,
  int kurs;                     // курс,
  char gr[10];                  // група,
  int ekz[5];                   // і масив екзаменаційних оцінок.
};
void func(student &S)           // Функція переведення на наступний курс
{ int j,k; char* ss="";
  for(j=0; j<5; j++) if(S.ekz[j]<60) return;
  S.kurs++;
  k=strchr(S.gr,'-')-S.gr;
  itoa(S.kurs,ss,10);
  S.gr[k+1]=ss[0];
}
void print(student S)       // Функція виведення на екран даних структури
{ cout<<"Прізвище "; puts(S.prizv);
  cout<<"Курс "<<S.kurs<<endl;
  cout<<"Група "; puts(S.gr);
  cout<<"Екзамени ";
  for(int j=0; j<5; j++) cout << S.ekz[j] << "  ";
  cout<<endl;
  }
}
int main(int argc, char* argv[])
{ student s[N];         // Оголошення масиву студентів
  int i,j;
  cout<<"Введіть інформацію про студентів: "<<endl;
  for(i=0; i<N; i++)  // Введення даних до масиву студентів
  { cout<<"Прізвище: "; cin>>s[i].prizv;
    cout<<"Курс: "; cin>>s[i].kurs;
    cout<<"Група: "; cin>>s[i].gr;
    cout<<"Екзамени: "<<endl;
    for (j=0; j<5; j++) cin>>s[i].ekz[j];
  }
```

```
  for(i=0; i<N; i++)
    func(s[i]);        // Виклик функції func() для кожного студента
  cout<<"Масив після змін: "<<endl;
  for(i=0; i<N; i++)
    print(s[i]);       // Виклик функції print() для кожного студента
  getch();
  return 0;
}
```

Результати виконання програми:

```
Введіть інформацію про студентів:
Прізвище: Іванов
Курс: 1
Група: ІМ-1.02
Екзамени:
65
87
50
60
65
Прізвище: Петров
Курс: 2
Група: ТС-2.01
Екзамени:
90
90
95
60
65
Масив після змін:
Прізвище: Іванов
Курс: 1
Група: ІМ-1.02
Екзамени: 65  87  50  60  65
Прізвище: Петров
Курс: 2
Група: ТС-2.01
Екзамени: 90  90  95  60  65
```

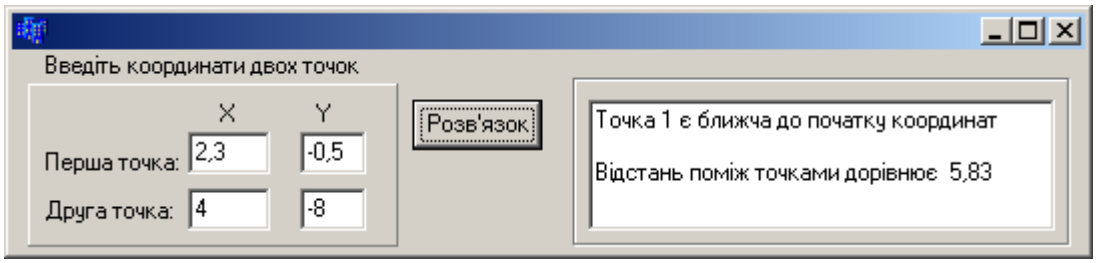
Приклад 11.2 Створити структуру для роботи з комплексними числами. До структури повинні входити поля для дійсної й уявної частин комплексного числа, а також функції, які задають операції з комплексними числами: перетворювання на тригонометричну форму і навпаки, обчислення кореня *n*-го степеня. Написати програму, що використовує описану структуру і обчислює корінь *n*-го степеня з комплексного числа. Число задавати у вигляді $a+ib$. Як результат роздрукувати всі *n* значень кореня *n*-го степеня у формі $a+ib$.



Текст програми:

```
typedef struct
{ float x,y;              // Дійсна і уявна частини числа
  // Функція перетворювання числа на тригонометричну форму
  void trig(float &mod,float &arg)
  { mod=sqrt(x*x+y*y); // Модуль числа у тригонометричній формі
    arg=acos(x/mod);    // і аргумент числа у тригонометричній формі
  }
  // Функція перетворювання числа з тригонометричної форми до звичної
  void i(float &a, float &b)
  { a=x*cos(y);         // Дійсна
    b=x*sin(y);         // і уявна частини числа
  }
  // Функція обчислення k-го значення кореня n-го степеня
  // з комплексного числа у тригонометричній формі.
  void sqrtn(int n, int k, float &modn, float &argn)
  { modn=pow(x,1/(float) n);
    argn=(y+2*M_PI*k) /n;
  }
}
TComplex;             // Кінець опису структури TComplex

void __fastcall TForm1::Button1Click(TObject *Sender)
{ TComplex c,z;
  c.x=StrToFloat(Edit1->Text); // Введення дійсної частини
  c.y=StrToFloat(Edit2->Text); // і уявної частини комплексного числа
  int n=StrToInt(Edit3->Text); // Введення степеня кореня
  float mod,arg,a,b,modn,argn;
  c.trig(mod,arg);             // Перетворення комплексного числа у
  z.x=mod; z.y=arg;            // тригонометричну форму (число z)
  for(int k=0;k<n;k++)         // Обчислення всіх k значень кореня n-го
   { z.sqrtn(n,k,modn,argn);   // степеня і перетворення
     z.x=modn; z.y=argn;       // з тригонометричної форми на звичну
     z.i(a,b);
     Memo1->Lines->Add(FormatFloat("0.00",a)+" + i * "+
                       FormatFloat("0.00",b));
   }
}
```

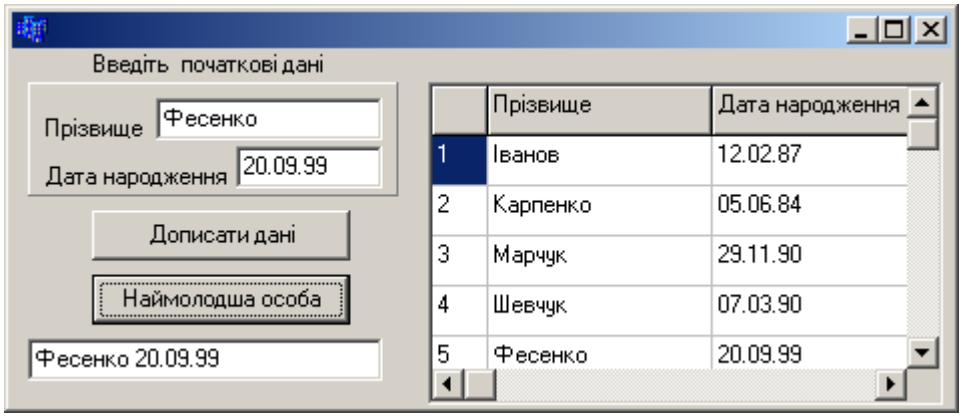
Приклад 11.3 Задано дві точки зі своїми координатами. Визначити, яка з цих точок є ближча до початку координат, та обчислити відстань поміж ними.



Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ struct Dec
  { double x, y; };
  Dec a1={2.0,1.0};   Dec a2={5.0,6.0};
  AnsiString S;
  double d1,d2;
  d1=sqrt(a1.x*a1.x+a1.y*a1.y);
  d2=sqrt(a2.x*a2.x+a2.y*a2.y);
  if(fabs(d1-d2) < 1E-6) S = "Відстані є однакові";
    else
      if(d1<d2) S = "Точка 1 є ближча до початку координат" ;
        else    S = "Точка 2 є ближча до початку координат";
  Memo1->Lines->Add(S);
  Memo1->Lines->Add("");
  double d= sqrt(pow((a1.x-a2.x),2)+pow((a1.y-a2.y),2));
  Memo1->Lines->Add("Відстань між точками дорівнює  " +
                    FloatToStrF(d,ffFixed,15,2));
}
```

Приклад 11.4 Увести відомості про прізвища і дати народження людей і визначити наймолодшого з них.



Текст програми:

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{SG1->RowCount=2;
 SG1->Cells[1][0]="Прізвище"; SG1->Cells[2][0]="Дата народження";
}
```

```
struct person
{ char priz[15];  char birth[10]; };
  person p[10];
  int n=0;
// Кнопка "Дописати дані"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ if(n>10) { ShowMessage("Переповнення масиву"); return; }
  strcpy(p[n].priz, Edit1->Text.c_str());
  strcpy(p[n].birth, Edit2->Text.c_str());
  SG1->Cells[0][n+1]=IntToStr(n+1);
  SG1->Cells[1][n+1]=AnsiString(p[n].priz);
  SG1->Cells[2][n+1]=AnsiString(p[n].birth);
  n++;
  SG1->RowCount++;
}
person young(person p[], int n)
{ person max_p=p[0];
  for(int i=0; i<n; i++)
    if(TDate(max_p.birth) < TDate(p[i].birth)) max_p=p[i];
  return max_p;
}
// Кнопка "Наймолодша особа"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ person max=young(p,n);
  Edit3->Text=AnsiString(max.priz)+" "+AnsiString(max.birth);
}
```

Структури дозволяють описати майже всі навколишні об'єкти. Однак існують певні винятки. Наприклад, припустімо, що структура `person` має для чоловіків містити найменування військової спеціальності, а для жінок – кількість дітей. Розв'язок "у лоба" може мати вигляд

```
struct person
{ date birth;        // Дата народження,
  char sex;          // стать (ч/ж),
  char wars[20];     // військова спеціальність
  int  kids;         // і кількість дітей.
};
```

Синтаксично начебто все правильно, але виникає можливість описати чоловіка, який народжував дітей (на жінок-військових ми покищо не зважаємо). Щоб уникнути такого становища, слід оголосити поля `wars` і `kids` у такий спосіб, щоб вони не могли мати значення водночас, тобто об'єднати ці поля в одне ціле. Для цього в C++ існують поєднання (`union`), які докладніше буде розглянуті у наступному підрозділі:

```
struct person
{ date birth;  char sex;
  union { char wars[20]; int kids; };
};
```

При використанні об'єднань водночас розв'язується й завдання економії пам'яті. Однак комп'ютер не відстежує, яке саме з двох полів, `wars` чи `kids`, має існувати даного моменту. Для компілятора водночас існують обидва поля.

11.3 Об'єднання (union)

Об'єднання – формат даних, який може містити різні типи даних, але лише один тип водночас. Можна сказати, що об'єднання є окремим випадком структури, всі поля якої розташовуються за однією й тією самою адресою. Формат опису об'єднання є такий самий як і у структури, лише замість ключового слова `struct` використовується слово `union`. Але, тоді як структура може містити, скажімо, елементи типу і `int`, і `short`, і `double`, об'єднання може містити чи то `int`, чи `short`, чи `double`:

```
union prim
 { int x;
   short y;
   double z;
 };
```

Обсяг пам'яті, яку займає об'єднання дорівнює найбільшому з розмірів його полів. Об'єднання застосовують для економії пам'яті у тих випадках, коли відомо, що понад одного поля водночас не потрібно. Часто об'єднання використовують як поле структури. При цьому до структури зручно включити додаткове поле, яке визначає, який саме елемент об'єднання використовується даного моменту:

```
struct
{ int type;
  union id
  { long id_num; char id_ch[20];  };
} price;
 . . .
if (price.type == 1) cin >> price.id.id_num;
else    cin >> price.id.id_ch;
```

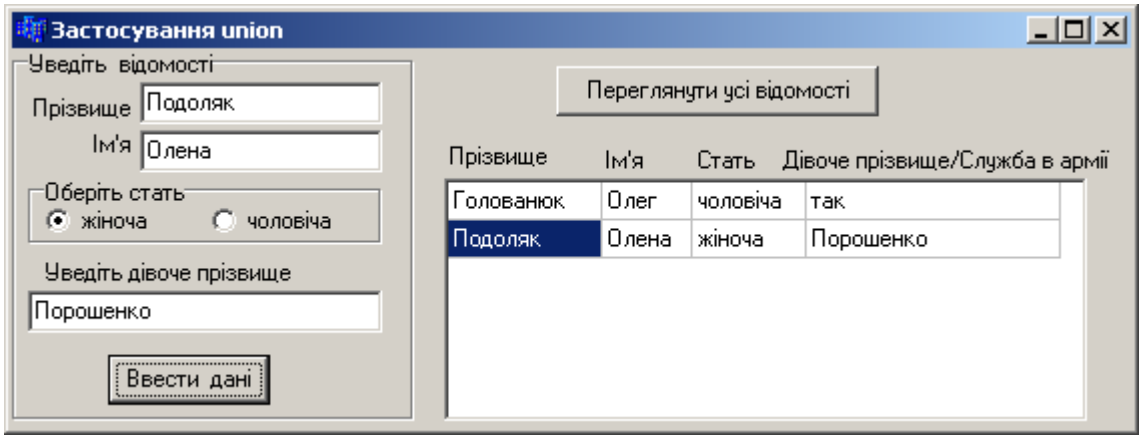
Тут структура `price` містить ціле поле `type` та поле `id`, яке може набувати лише одне зі значень чи то цілого числа `id_num`, чи то рядка `id_ch` залежно від значення поля `type`. У наведеному прикладі ім'я об'єднання можна не зазначати, а звертатися безпосередньо до його полів. У цьому разі об'єднання є анонімним, його елементи стають змінними, розташованими за однією адресою.

```
struct
{ int type;
  union
  { long id_num;
    char id_ch[20];
  };
} price;
. . .
if(price.type == 1) cin >> price.id_num;
else  cin >> price.id_ch;
```

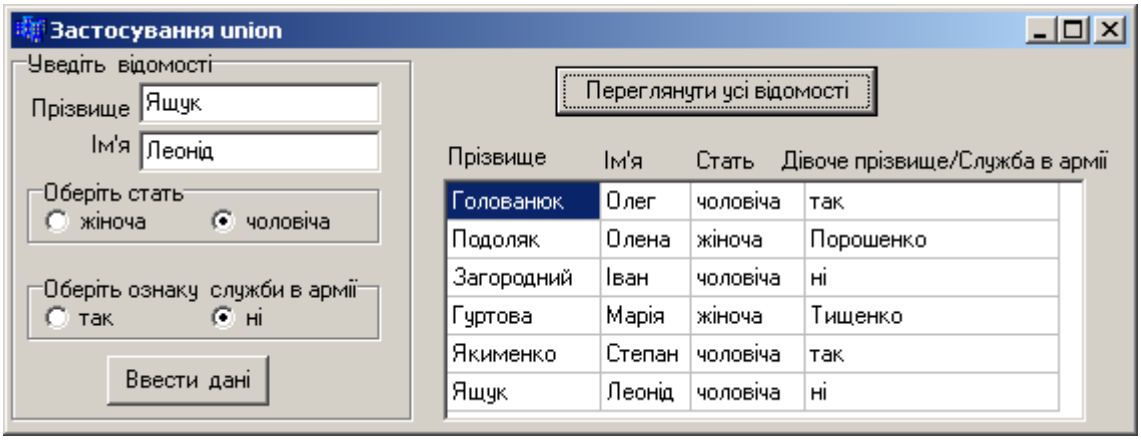
Приклад 11.5 Створити програму для опрацювання даних про людей: прізвище, ім'я, стать, а також дівоче прізвище для жінок та ознаку служби в лавах Збройних Сил для чоловіків.

Розв'язок. У програмі передбачено змінювання вигляду форми залежно від вибору особи статі, дані якої вводитимуться: для жінок виводитиметься вікно `Edit3` для введення дівочого прізвища, а для чоловіків – компонент `RadioGroup2` для вибору ознаки служби в армії. Програмний код для такої видозміни форми записано у функції відгуку події `RadioGroup1Click()`, тобто ця програма спрацьовуватиме автоматично при виборі особи статі.

При введенні даних особи жіночої статі вікно форми матиме вигляд



При перегляданні даних вікно форми матиме вигляд



Текст програми:

```
struct osoba
{ AnsiString prizv, imja;
  bool st;
  union                    // Об'єднання може зберігати
  { char VirgSername[25]; // чи то дівоче прізвище жінки,
    bool ArmySrv;          // чи ознаку служби в Збройних Силах для чоловіків
  } P;
} ;
 osoba ch[30];             // Масив із 30-ти структур
 int i=0;                  // Кількість введених даних
```

```
// При обиранні статі особи відбуватиметься приховування зайвих компонентів та
// виведення потрібних компонентів для введення відповідних даних
void __fastcall TForm1::RadioGroup1Click(TObject *Sender)
{ if(RadioGroup1->ItemIndex==0)
   { ch[i].st=0;  RadioGroup2->Visible=0;
     Edit3->Visible=1; Label3->Visible=1; }
  else
   { ch[i].st=1;  RadioGroup2->Visible=1;
     Label3->Visible=0; Edit3->Visible=0; }
}
//---------------------------------------------------------
// Кнопка "Ввести дані"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ch[i].prizv=Edit1->Text;
  ch[i].imja=Edit2->Text;
  ch[i].st=RadioGroup1->ItemIndex;
  if(ch[i].st) ch[i].P.ArmySrv=RadioGroup2->ItemIndex;
  else  strcpy(ch[i].P.VirgSername,Edit3->Text.c_str());
  i++;
}
//---------------------------------------------------------
// Кнопка "Переглянути усі відомості"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ SG1->RowCount=i;
  for(int k=0;k<i;k++)
  { SG1->Cells[0][k]=ch[k].prizv;
    SG1->Cells[1][k]=ch[k].imja;
    if(ch[k].st)
    { SG1->Cells[2][k]="чоловіча";
      if(ch[k].P.ArmySrv)SG1->Cells[3][k]="ні";
      else SG1->Cells[3][k]="так";
    }
    else { SG1->Cells[2][k]="жіноча";
           SG1->Cells[3][k]=ch[k].P.VirgSername;
} }      }
```

11.4 Перерахування (enum)

При написанні програм часто виникає потреба у визначенні наперед відомої кількості іменованих констант, які мають мати різні значення (при цьому конкретні значення можуть бути неважливими). Для цього зручно користуватися типом перерахування enum, всі можливі значення якого задаються списком цілочисельних констант:

enum [<*ім'я_типу*>] {<*список_констант*>};

Ім'я типу задається в тому разі, якщо у програмі потрібно визначати змінні цього типу. Змінним перераховного типу можна присвоювати кожне значення зі списку констант, зазначених при оголошуванні типу. Імена перераховних констант мають бути унікальними. Перераховні константи подаються й опра-

цьовуються як цілі числа і можуть ініціалізуватися у звичайний спосіб. Окрім того вони можуть мати однакові значення. За відсутності ініціалізації перша константа обнулюється, а кожній наступній присвоюється значення на одиницю більше, аніж попередній.

```
enum week { sat=0, sun=0, mon, tue, wed, thu, fri } rob_den;
```

Тут описано перерахування `week` з відповідною множиною значень і оголошено змінну `rob_den` типу `week`. Перераховні константи `sat` та `sun` мають значення 0, `mon` – значення 1, `tue` – 2, `wed` – 3, `thu` – 4, `fri` – 5.

До перераховних змінних можна застосовувати арифметичні операції й операції відношення, наприклад:

```
enum days {sun, mon, tue, wed, thu, fri, sat};
days day1 = mon,  day2 = thu;
int diff = day2 - day1;
cout << "Різниця у днях: " << diff << endl;
if(day1 < day2)
cout << "day1 настане раніш, аніж day2 \n";
```

Арифметичні операції над змінними перераховних типів зводяться до операцій над цілими числами. Проте, не зважаючи на те, що компіляторові відомо про цілочисельну форму подання перераховних значень, варто використовувати цей факт надто обережно. Якщо спробувати виконати присвоювання

```
day1 = 5;
```

компілятор видасть повідомлення (хоча компіляція відбудеться без помилок). Рекомендовано без нагальної потреби не використовувати цілочисельну інтерпретацію перераховних значень.

Значення перераховним константам можна встановлювати явно, ініціалізуючи їх при оголошенні, причому значення мають бути лише цілими числами, наприклад:

```
enum {first, second = 100, third};
```

У цьому разі `first` за замовчуванням дорівнює 0, а наступні неініціалізовані константи перерахування перевищують своїх попередників на 1. Приміром, `third` матиме значення 101.

Істотним недоліком типів перерахування є те, що вони не розпізнаються засобами введення-виведення C++. При виведенні такої змінної виводиться не її формальне значення, а її внутрішнє подання, тобто ціле число. Наведемо приклад програми, яким можна скористатись за потреби виведення значень таких змінних у зрозумілому для сприйняття вигляді.

```
#include <iostream>
#include <conio.h>
#include <string>
using namespace std;
enum day {mon=1, tue, wed, thu, fri, sat, sun};
string dayToString(const day& one)
{ string arr[]={"Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday","Saturday","Sunday"};
  return arr[one-1];  }
```

```
day intToDay(int one)
{ return static_cast<day>(one); }
int main()
{ int numb;
  cout << "Enter number of day (1,2,3,4,5,6 or 7): ";
  cin >> numb;
  day one = intToDay(numb);
  cout << dayToString(one) << '\n';
  getch();  return 0;
}
```

Використаний у функції `intToDay()` оператор `static_cast` має такий формат:

`static_cast <` *тип* `> (` *вираз* `)`

і перетворює без жодних перевірок відповідності під час виконання програми *вираз* до зазначеного *типу*. У нашому разі відбуватиметься перетворення введеної цілої змінної `one` до перерахованого типу `day`.

Інші приклади програм із використанням типу `enum` наведено у підрозд. 13.7 (див. приклади 13.11 – 13.14).

11.5 Множини (Set)

Множина – це група елементів, яка асоціюється з її ім'ям і з якою можна порівнювати інші величини, щоб визначати їхню належність до цієї множини. В окремих випадках множина може бути порожньою.

Множину в C++ Builder реалізовано як шаблон класу, визначений у заголовному файлі <sysset.h>.

Оголошується множина оператором:

`Set <`*type, minval, maxval*`> змінні;`

Параметр *type* визначає тип елементів множини, зазвичай типу `int`, `char` чи `enum`. Параметри *minval* та *maxval* визначають мінімальне й максимальне значення елементів множини типу `unsigned char`. Мінімальне значення завжди має бути не менше за 0, а максимальне – не перевищувати 255.

Наприклад,

```
Set <char, 'A', 'Z'> s1;
```

оголошує змінну `s1` множиною всіх заголовних літер латиниці.

Ініціалізація множини здійснюється за допомогою долучення елемента до множини операцією <<, а вилучення елементів з множини – операцією >>. Наприклад:

```
s1 << 'A' << 'B' << 'C' << 'Z';
```

долучає до множини `s1` символи `'A'`, `'B'`, `'C'`, `'Z'`, а операція

```
s1 >> 'B' >> 'C';
```

вилучає з множини `s1` символи `'B'`, `'C'`.

Для переглядання значень, записаних у множині, можна скористатися методом `Contains(`<*значення*>`)`, який дає результат `true`, якщо певне значення записано у множині, і `false`, – якщо воно є відсутнє у множині.

Наведений далі приклад показує, в який спосіб можна вивести на екран усі значення, записані у множині:

```
AnsiString s="";  char c;
Set <char, 'A', 'Z'> s1;
s1 << 'A' << 'B' << 'Z';
for(c='A'; c<='Z'; c++)
  if(s1.Contains(c))   s = s + AnsiString(c)+" ";
Edit1->Text = s ;      //s="ABZ"
```

Ще один приклад є аналогічний до попереднього, але в ньому у множині записуються цілі додатні числа:

```
AnsiString s = "";  char c;
Set <short int, 0, 200> f;
f << 1 << 5 << 19 << 119;
f << 114;
for (int i=1; i<=200; i++)
  if (f.Contains(i))   s += IntToStr(i) + "  ";
Edit1->Text = s ;      //s="1 5 19 114 119"
```

Окрім розглянутого методу `Contains()`, існують метод очищення множини `Clear()` і метод `Empty()`, який визначає, чи не є множина порожньою (табл. 11.1).

Таблиця 11.1

Методи роботи з множинами

Формат методу	Опис
`Set& **Clear**();`	Очищення заданої множини
`bool **Contains**(T el);`	Перевірка наявності у множині елемента `el`
`bool **Empty**();`	Перевірка того, чи не є множина порожньою

Подані нижче оператори оголошують множину `S`, елементами якої є дані типу перерахування `E`: `red`, `yellow`, `green`:

```
enum E (white, red, yellow, green);
Set <E, red, green> S;
```

Поширені у застосовуванні операції для множин наведені в табл. 11.2.

Таблиця 11.2

Операції роботи з множинами

Операція	Формат	Опис
`-` `-=`	`Set operator **-** (Set& rhs);` `Set operator **-=** (Set& rhs);`	Із заданої множини вилучаються елементи множини `rhs`
`*` `*=`	`Set operator **\*** (Set& rhs);` `Set operator **\*=** (Set& rhs);`	Перетин множин, тобто спільні елементи заданої множини та `rhs`
`+` `+=`	`Set operator **+** (Set& rhs);` `Set operator **+=** (Set& rhs);`	До заданої множини долучаються елементи множини `rhs`
`==` `!=`	`bool operator **==**(Set& rhs);` `bool operator**!=**(Set& rhs);`	Еквівалентність і нееквівалентність двох множин: заданої та `rhs`

Розглянемо кілька прикладів застосовування операцій, поданих у табл. 11.2:

```
Set <short, 0, 200> f;
f << 1 << 5 << 19 << 119; // f складається з 1, 5, 19, 119
Set <short, 0, 200> f1;
f1 << 2 << 45;            // f1 складається з 2, 45
f = f.operator +(f1);     //або f+=f1; тепер f складається з 1,2,5,19,45,119
Set <short, 0, 200> f2;
f2 << 2 << 5 << 19 << 47; // f2 складається з 2, 5, 19, 47
f = f.operator * (f2);    // Операція operator* або f *= f1;
// f є перетином множин f та f2, в якому залишилися числа 2, 5 та 19
```

Наведемо приклади застосовування множин на практиці. Якщо треба обмежити можливість вводити до вікна редактора `Edit1` лише числа від 0 до 9, для цього у відгуку події `OnKeyPress` вікна `Edit1` слід записати такі оператори:

```
Set <char, '0', '9'> Dig;
Dig <<'0'<<'1'<<'2'<<'3'<<'4'<<'5'<<'6'<<'7'<<'8'<<'9';
if(!Dig.Contains(Key)) Key=0;
```

Відомо, що для компонента `StringGrid` властивість `Options` визначена як множина такого вигляду:

```
enum TGridOption { goFixedVertLine, goFixedHorzLine,
goVertLine, goHorzLine, goRangeSelect, goDrawFocusSelected,
goRowSizing, goColSizing, goRowMoving, goColMoving, goEditing,
goTabs, goRowSelect, goAlwaysShowEditor, goThumbTracking };
typedef Set <TGridOption, goFixedVertLine, goThumbTracking>
TGridOptions;
```

Тому, для долучення певних опцій компонента `StringGrid1` із зазначених значень множини, наприклад для можливості редагування даних у комірках при виконуванні програми і для переміщення від однієї комірки до іншої натисненням клавіші <Tab>, у програмі можна записати оператор

```
StringGrid1->Options<<goEditing<<goTabs;
```

а для вилучення однієї з опцій властивості `Options`, наприклад `goTabs` – оператор

```
StringGrid1->Options>>goTabs;
```

Питання та завдання для самоконтролю

1) В який спосіб можна створювати нові типи даних на базі вже існуючих? Запишіть оголошення типу `Str` як масиву з 60-ти символів.

2) Дайте означення структури як типу даних.

3) У чому полягає відмінність структури від масиву?

4) Як здійснюється доступ до полів структури?

5) Як визначається обсяг пам’яті, потрібний для зберігання структури?

6) Чи можуть в одній програмі збігатися імена полів структури і змінних?

7) Чи можуть збігатися в одній програмі імена полів двох структур?

8) Чи дозволяється створювати вкладені структури?

9) Придумайте оголошення структури, яка описуватиме для деякого електричного приладу такі характеристики: назва приладу, споживана потужність та номінальна напруга.

10) Чи може матриця бути полем структури? Якщо так, наведіть приклад.

11) Чи має значення порядок слідування полів у описі структури?

12) Чи може структура бути типом елемента масиву? Якщо так, наведіть приклад.

13) Оголосіть структуру з ім'ям `Student`, що містить такі поля: прізвище, назва групи, масив з п'яти екзаменаційних оцінок. Запишіть команди, які нададуть початкові значення усім полям змінної типу `Student`.

14) Найдіть та виправте помилки припущені в описі структури:

```
structure { int arr[12], char string, int *sum }
```

15) Дайте означення типу даних "об'єднання".

16) Як визначається обсяг пам'яті, потрібний для зберігання усіх полів певного об'єднання?

17) Напишіть об'єднання, в якому може зберігатися чи то покажчик, чи ціле, чи дійсне число.

18) Охарактеризуйте тип перерахування.

19) Яких значень набувають константи перерахування за відсутності їхньої ініціалізації?

20) Якщо перерахування `COLOR` задано в такий спосіб, то яким є значення його елементів `white`, `red`, `blue` та `yellow`?

```
enum COLOR { white, black=100, red, blue, green=300, yellow};
```

21) Які помилки містять наведені оператори?

```
enum State { on, off };
enum YesNo { on, off };
```

22) Правильним чи ні є таке оголошення перерахування?

```
enum YesNo { no = 0, No = 0, yes = 1, Yes = 1 };
```

23) Створіть оголошення перерахування `response` зі значеннями "Так", "Ні" і "Можливо". Значення "Так" повинно мати порядковий номер 1, "Ні" – 0 та "Можливо" – 2.

24) Дайте означення множини в C++ Builder.

25) Якими є елементи множини `f`, якщо її визначено у такий спосіб:

```
Set <int, 100, 300> f;
f << 1 << 105 << 119 << 245 << 419;
```

а) 1, 105, 119, 245, 419;

б) 105, 119, 245;

в) усі цілі числа в межах від 100 до 300;

г) пуста множина, оскільки станеться помилка.

Розділ 12

Файли

12.1 Загальні відомості про файли

Файлами є іменовані області пам'яті на зовнішньому носії, призначені для довготривалого зберігання інформації. Файли мають *імена* й є організовані в ієрархічну деревовидну структуру з *каталогів* (тек) і простих файлів.

Для доступу до даних файла з програми в ній слід прописати функцію відкриття цього файла, тим самим встановити зв'язок поміж ім'ям файла і певною файловою змінною у програмі.

Файли відрізняються від звичайних масивів тим, що

- ✓ вони можуть змінювати свій розмір;
- ✓ звертання до елементів файлів здійснюється не за допомогою операції індексації [], а за допомогою спеціальних системних викликів та функцій;
- ✓ доступ до елементів файла відбувається з так званої позиції зчитування-записування, яка автоматично просувається при операціях зчитування-записування, тобто файл є видимим послідовно. Існують, щоправда, функції для довільного змінювання цієї позиції.

Часто використовується така метафора: якщо уявляти собі файли як книжки (лише зчитування) і блокноти (зчитування й записування), які стоять на полиці, то *відкриття* файла – це вибір книжки чи блокнота за заголовком на його обкладинці й відкриття обкладинки (на першій сторінці). Після відкриття можна читати, дописувати, викреслювати і правити записи, перегортати книжку. Сторінки можна зіставити з блоками файла, а полицю з книжками – з каталогом.

C++ Builder надає засоби опрацювання двох типів файлів: *текстових* та *бінарних*.

Текстові файли призначено для зберігання текстів, тобто сукупності символьних рядків змінної довжини. Кожен рядок завершується керувальною послідовністю '\n', а розділювачами слів та чисел у рядку є пробіли й символи табуляції. Оскільки вся інформація текстового файла є символьною, програмне опрацювання такого файла полягає в читанні рядків, виокремлюванні з рядка слів і, за потреби, перетворюванні цифрових символьних послідовностей на числа відповідними функціями перетворювання. Створювати, редагувати текстові файли можна не лише в програмі, а й у якому завгодно текстовому редакторі, наприклад Блокноті чи Word.

Бінарні файли зберігають дані в тому самому форматі, в якому вони були оголошені, і їхній вигляд є такий самий, як і в пам'яті комп'ютера. І тому відпадає потреба у використанні розділювачів: пробілів, керувальних послідовностей, а отже, обсяг використовуваної пам'яті порівняно з текстовими файлами з аналогічною інформацією є значно меншим. Окрім того, немає потреби у застосуванні функцій перетворення числових даних. Але кожне опрацювання даних

бінарних файлів можливе лише за наявності програми, якій має бути відомо, що саме і в якій послідовності зберігається у цьому файлі.

Послідовно розглянемо засоби створення, записування, зчитування й опрацювання файлів обох типів.

Робота з файлами в C++ Builder може виконуватися кількома різними способами:

- ✓ використання бібліотечних компонентів;
- ✓ робота з файлами як з потоками у стилі C;
- ✓ робота з файлами як з потоками у стилі C++;
- ✓ робота з файлами як з потоками, які використовують дескриптори.

Спочатку розглянемо можливості для створення й опрацювання текстових файлів.

12.2 Текстові файли

12.2.1 Зчитування і записування текстових файлів за допомогою компонентів C++ Builder

Робота з текстовими файлами може здійснюватися за допомогою методів `LoadFromFile()` та `SaveToFile()`, які є у класів `TStrings` та `TSringList`. Ці класи описують списки рядків і мають безліч методів маніпулювання рядками.

Для того щоб прочитати вміст певного текстового файла, створеного, наприклад у Блокноті, опрацювати його і зберегти змінення у файлі, слід виконати таку послідовність дій:

- ✓ визначити дві глобальні змінні – список типу `TSringList` і рядкову змінну типу `AnsiString`, наприклад:

```
TStringList *List = new TStringList;
AnsiString SFile = "Test.txt";
```

- ✓ завантажити файл з ім'ям `SFile` до свого списку за допомогою команди

```
List->LoadFromFile(SFile);
```

- ✓ зберегти свій файл після редагування за допомогою команди

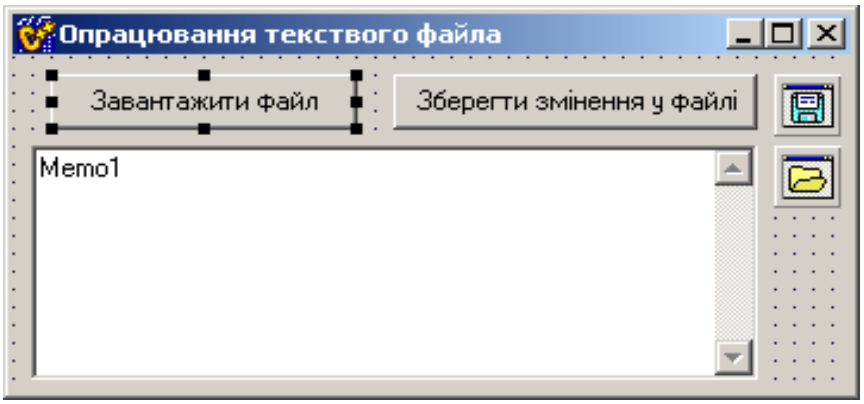
```
List->SaveToFile(SFile);
```

При відкриванні та зберіганні файла можна користуватися стандартними компонентами-діалогами `OpenDialog` та `SaveDialog`, розташованими на закладці `Dialogs` (див. табл. 2.4 у п. 2.1.1). Для відкривання, переглядання й редагування файла можна користуватися стандартними багаторядковими компонентами типу `TMemo` й `TRichEdit`. У останньому разі можна працювати з файлами у форматі `RTF`. Властивості `Lines` цих компонентів мають тип `TStrings`, що дозволяє безпосередньо користуватися методами `LoadFromFile()` та `SaveToFile()`, наприклад:

```
Memo1->Lines->LoadFromFile(SFile);
RichEdit1->Lines->LoadFromFile(SFile);
```

Через компоненти C++ Builder можна працювати не лише з текстовими файлами, але і з файлами зображень і мультимедіа.

Наведемо найпростіший приклад завантаження текстового файла і зберігання змін, якщо їх було внесено. Для цього на формі встановимо компоненти `Memo`, `OpenDialog`, `SaveDialog` та три кнопки `Button` й матимемо зображення



Текст програми в C++ Builder:

```
// Кнопка "Завантажити файл".
void __fastcall TForm1::Button1Click(TObject *Sender)
{ if(OpenDialog1->Execute())
    Memo1->Lines->LoadFromFile(OpenDialog1->FileName);
}
// Кнопка "Зберегти змінення у файлі".
// Зберігання файла  виконується при зміненні тексту в Memo1.
void __fastcall TForm1::Button2Click(TObject *Sender)
{ if(Memo1->Modified)
    if(SaveDialog1->Execute())
      Memo1->Lines->SaveToFile(SaveDialog1->FileName);
}
```

12.2.2 Робота з текстовими файлами у стилі C

У мові C *файл* розглядається як потік послідовності байтів. Інформація про файл заноситься до змінної типу `FILE*`. Цей тип оголошує вказівник потоку, який використовується надалі у всіх операціях з цим файлом. Тип `FILE` означено у бібліотеці `<stdio.h>`. Тому, якщо в програмі передбачається робота з файлами, цей заголовний файл слід долучити:

```
#include <stdio.h>
```

Тепер можна оголосити файлову змінну – вказівник потоку, який в подальшому буде передаватися у функції введення-виведення у якості параметра:

```
FILE *f;
```

Функція `fopen()` для відкривання файла має такий синтаксис:

```
FILE *fopen(const char *filename, const char *mode);
```

Перший параметр `filename` визначає ім'я файла, який відкривається. Другий параметр `mode` задає режим відкривання файла (табл. 12.1).

Таблиця 12.1

Специфікатори режиму відкривання файлів

Параметр	Опис
r	відкрити файл лише для зчитування даних
r+	відкрити існуючий файл для зчитування й записування даних
a	відкрити чи створити файл для записування даних у кінець файла
a+	відкрити чи створити файл для зчитування чи то записування даних у кінець файла
w	створити файл для записування даних
w+	створити файл для зчитування і записування даних

До зазначених специфікаторів наприкінці чи перед знаком "+" може дописуватись символ чи то "t" – для текстових файлів, чи "b" – для бінарних (двійкових) файлів.

Функція `fopen()` повертає вказівник на об'єкт, який керує потоком. Наприклад, створити файл з ім'ям `Prim.txt` можна так:

```
f = fopen("Prim.txt", "wt");
```

Якщо файл відкрити не вдалося, `fopen()` повертає нульовий вказівник `NULL`. Для уникання помилок після відкриття файла слід перевірити, чи насправді файл відкрився:

```
if(f == NULL) {ShowMessage("Файл не відкрито"); return;}
```

Припинити роботу з файлом можна за допомогою функції

```
fclose(FILE *f).
```

Ця функція закриває файл, на який посилається параметр функції.

Наведемо приклад відкривання текстового файла для зчитування.

```
FILE *f;
// Перевіряємо, чи повертає  ця функція нульовий вказівник
if((f=fopen("t.txt","rt"))==NULL)
 { ShowMessage("Неможливо відкрити файл");     return; }
. . .                  // Сюди вставляються команди  зчитування з файла.
fclose(f);             // Закриття файла.
```

З текстового файла можна читати цілі рядки й окремі символи. Слід звернути увагу на те, що тип `FILE*` перейшов в C++ зі стандартного C, і такий файл "розуміє" лише C-рядки, тобто рядки типу `char*`. Тому в C++ Builder усі рядки типу `AnsiString` доводиться перетворювати за допомогою методу `c_str()` на C-рядок і, навпаки, всі C-рядки для виведення на форму треба перетворювати на `AnsiString`.

Зчитування рядка з файла здійснюється функцією `fgets()`:

```
char *fgets(char *s, int m, FILE *stream);
```

де `s` – перший параметр – рядок типу `char*`;

`m` – кількість читаних символів (байтів);

`stream` – вказівник на потік даних файла.

Перевірка кінця файла здійснюється функцією `feof()`:

```
int feof (FILE *stream);
```

Розглянемо детальніше використання цих функцій:

```
char s[50]; Memo1->Clear();
do { fgets(s,50,f);
     Memo1->Lines->Add(s);
   } while (!feof(f));
```

Даний приклад ілюструє зчитування даних з файла порядко́во за допомогою функції `fgets()` і виведення рядків до компонента `Memo`. Перший параметр функції `fgets()` – це С-рядок, в який читається черговий рядок текстового файла. Зчитування рядка відбувається до появи символу кінця рядка `'\n'` або ж припиняється, коли прочитано `m-1` символ. У нашому прикладі `m=50`. Отже, якщо файл містить рядок, який має понад 50 символів, то буде прочитано лише перші 49 символів. При цьому поточна позиція файла залишиться в тому самому рядку й за подальшого використання функції `fgets()` читатиметься залишок рядка. Третій параметр зазначає потік, з якого здійснюється зчитування.

У даному прикладі при зчитуванні всіх даних з файла `f` використовується функція `feof(f)`, яка перевіряє, чи не прочитано символ кінця файла. Після зчитування цього символу функція `feof(f)` поверне ненульове значення – і цикл перерветься.

У наведеному прикладі є два недоліки:

1) річ у тім, що прочитаний рядок може завершуватися символом `'\n'`, який є ознакою кінця рядка. Цей символ також відобразиться у вікні `Memo` й "зіпсує" вигляд вихідних даних. Усунути цей недолік можна за допомогою перевірки командою

```
if(s[strlen(s)-1]=='\n')  s[strlen(s)-1]= '\0';
```

який прибере з рядка символ `'\n'`;

2) використання функції `feof()` часто призводить до появи дубліката останнього рядка файла, тому рекомендовано використовувати в якості умови циклу функцію зчитування даних, наприклад `fgets()`, або контролювати добігання кінця файла, тобто перевіряти, чи прочитано всі записані у файлі символи (нагадаємо, що 1 символ – 1 байт). Наведений приклад зчитування рядків з файла можна тепер записати в такий спосіб:

```
while(fgets(s,50,f))
 { if(s[strlen(s)-1]=='\n')  s[strlen(s)-1] = '\0';
   Memo1->Lines->Add(s);  }
```

Зчитування форматованих даних можна також здійснювати за допомогою функції `fscanf()`:

```
int fscanf(FILE *stream, const char *format[, address.]);
```

Наведемо приклад використання цієї функції:

```
float r;
Memo1->Clear();
while(fscanf(f,"%e",&r)>0) Memo1->Lines->Add(FloatToStr(r));
```

У даному прикладі здійснюється зчитування даних з файла, який містить дійсні числа. За розділювачі поміж числами вважаються пробіли. Перший параметр `f` функції `fscanf()` визначає файл, з якого відбувається зчитування. Другий параметр задає формат рядка аргументів, заданих їхніми адресами. Функція `fscanf()` є цілого типу і повертає, як своє вихідне значення, кількість прочитаних елементів. При форматованому читанні часто можуть виникати помилки через невідповідність форматів чи то кінець файла. Для уникання таких помилок доцільно організовувати перевірки кількості прочитаних функцією `fscanf()` елементів. У наведеному прикладі формат `"%e"` визначає дійсне число з рухомою крапкою. Прочитане дійсне число з файла зберігається за адресою змінної `r` типу `float`.

Рядок форматування параметра `format` будується з послідовності символів-специфікаторів типів читаних даних, найпоширеніші з яких наведено в табл. 12.2.

Таблиця 12.2

Специфікатори параметра `format`

Символ	Значення, що вводиться	Тип аргументу
`i, l`	десяткове, вісімкове чи шістнадцяткове ціле число	`int, long`
`d, D`	десяткове ціле число	`int, long`
`e, E`	дійсне число з рухомою крапкою	`float`
`f, F`	дійсне число з фіксованою крапкою	`float`
`s`	рядок символів	`char s[]`
`c`	символ	`char`

Записування даних до текстового файла можна здійснювати за допомогою функції

```
int fputs(const char *s, FILE *stream);
```

де `s` – рядок типу `char`,

`stream` – файловий потік.

Для записування даних до файла слід відкрити файл у форматі записування даних у кінець файла, за потреби перетворити рядок на тип `char` і записати дані до файла. Послідовність процесу показано у фрагменті програми:

```
f=fopen("a.txt","at+");
if(f==0)
 { ShowMessage("Не вдається створити файл ");
   return;
 }
char s[40];
// Копіюємо рядок з Edit1 до s, перетворюючи його на С-рядок.
strcpy(s,Edit1->Text.c_str());
// Долучаємо до рядка символ <Enter>, інакше у файлі
strcat(s,"\n");// все буде записано одним рядком.
fputs(s,f);      // Записуємо рядок s до файла.
fclose(f);
```

Записування до текстового файла можна здійснити також за допомогою функції `fprintf()`:

```
int fprintf(FILE *stream, const char *format [, argument]);
```

Ця функція є подібна до функції `fscanf()`, але має ширші можливості побудови рядка форматування, наприклад:

```
char s[20];
strcpy(s, "Іванов");
int year=1985;
fprintf(F, "ХАРАКТЕРИСТИКА\nспівробітник %s, %i р.н.\n",
           &s, year);
```

У результаті виконання цих команд до файла буде записано таке:

```
ХАРАКТЕРИСТИКА
співробітник Іванов, 1985 р.н.
```

Як бінарні, так і текстові файли дозволяють переміщувати поточну позицію зчитування-записування. Для визначення поточної позиції файла, яка автоматично зміщується на кількість опрацьованих байтів, використовується функція `ftell()`:

```
long int ftell(FILE *stream);
```

А змінити поточну позицію файла можна за допомогою функції `fseek()`:

```
int fseek(FILE *stream, long offset, int whence);
```

Ця функція задає зсув на кількість байтів `offset` щодо точки відліку, яка задається параметром `whence`. Параметр `whence` може набувати таких значень:

Константа	whence	Точка відліку
SEEK_SET	0	початок файла
SEEK_CUR	1	поточна позиція
SEEK_END	2	кінець файла

Наприклад, для переміщення поточної позиції на початок файла можна скористатися функцією

```
fseek(f, 0, SEEK_SET);
```

або

```
fseek(f, 0, 0);
```

За допомогою функцій `ftell()` та `fseek()` можна визначити сумарний обсяг пам'яті у байтах, який займає файл. Для цього є достатньо переміститися на кінець файла:

```
fseek(f, 0, SEEK_END);
int d = ftell(f);
```

Отже, послідовність роботи з файлом при записуванні даних є така:

1) Відкрити чи то створити файл `fname.txt` для записування (або для зчитування й записування) у кінець файла:

```
f=fopen("fname.txt", "at+");
```

2) Записати дані до файла `f` однією з команд:

```
fputs(s, f);   // Записування С-рядка s
//Форматоване записування С-рядка nazva, дійсного числа price і цілого числа kol
fprintf(f, "%s %6.2f %i\n", nazva, price, kol);
```

3) Закрити файл:

```
fclose(f);
```

Для зчитування даних з файла треба виконати таку послідовність дій:

1) Відкрити файл для зчитування

```
f=fopen("fname.txt","rt+");
```

2) Здійснити зчитування даних з файла однією з команд:

```
fgets(s, 80, f);   // Зчитування С-рядка s довжиною у 80 символів
// Форматоване зчитування за адресами відповідних змінних
fscanf(f, "%s %f %i\n", &nazva, &price, &kol);
```

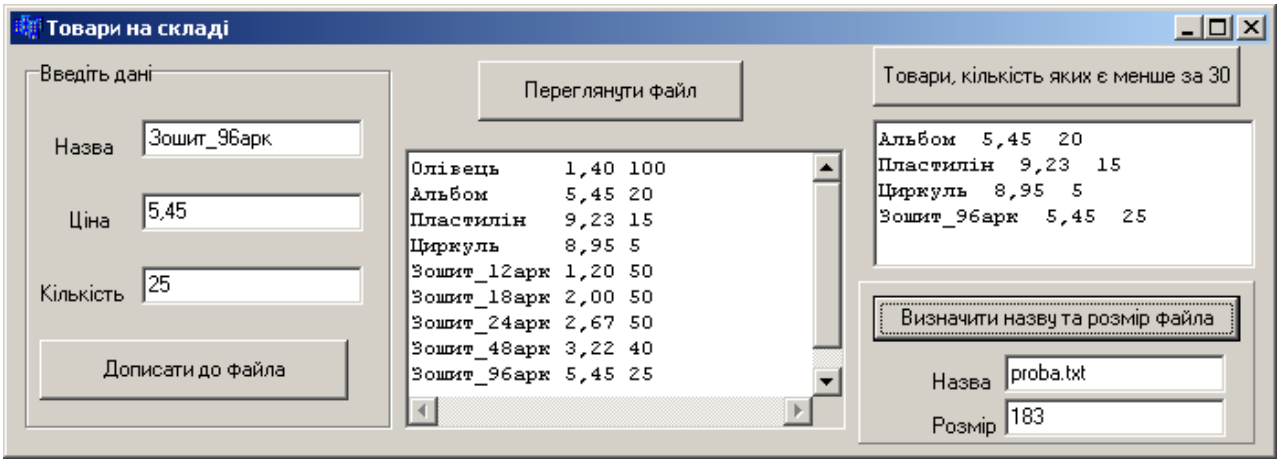
3) Для зчитування послідовно всіх даних з файла треба організувати цикл з умовою, яка перевіряє досягання кінця файла, використовуючи чи то функцію `feof(f)`, яка є ідентична до `true` при досяганні кінця файла, чи функцію зчитування даних, яка дає нульовий результат за досягання кінця файла. Можна використовувати інформацію про сумарний розмір файла і контролювати досягання кінця файла функцією `ftell(f)`.

4) Закрити файл: `fclose(f);`

У прикладах, що наводяться нижче, всі ці можливості послідовно подано у програмах.

Приклад 12.1 Увести відомості про товари до текстового файла: назву, ціну товару та кількість товару на складі. Прочитати дані з файла і віднайти відомості про товари, кількість яких є менше за 30. Визначити назву і розмір файла.

Форма додатка з результатами роботи матиме вигляд



Текст програми:

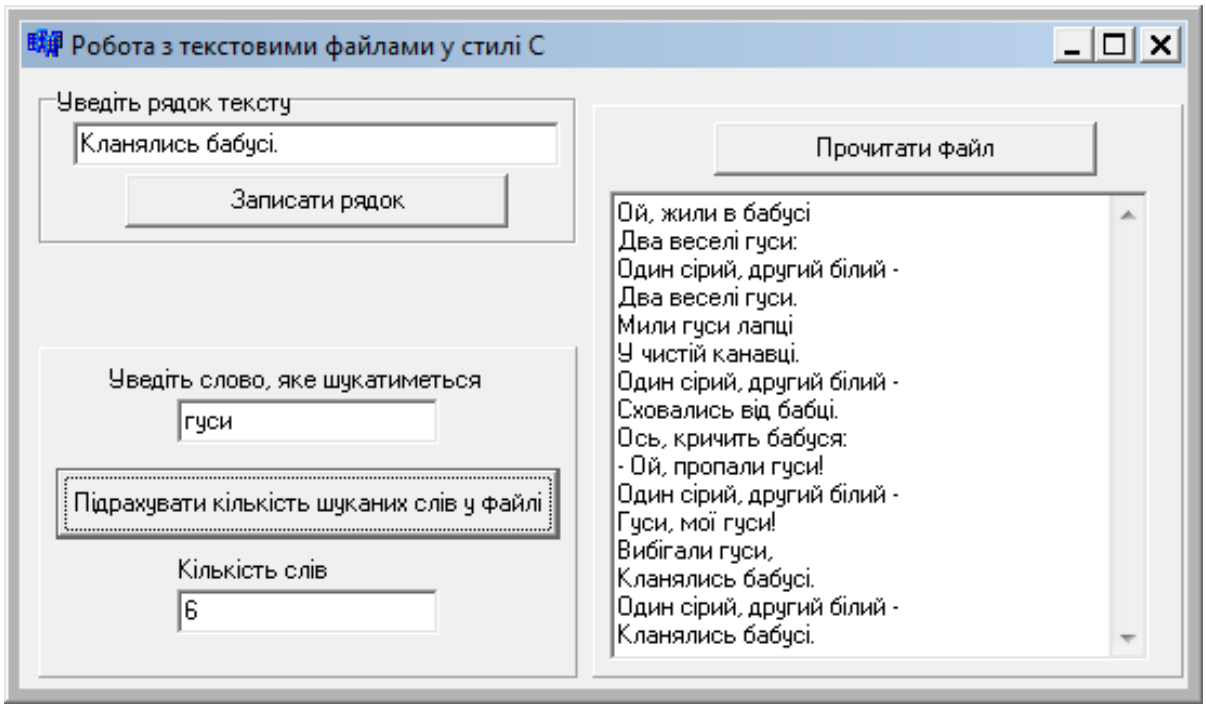
```
#include <stdio.h>
// Глобальне оголошення файлової змінної і визначення імені файла
FILE *f;
char s[]="proba.txt";
```

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{ // Перевіряється  наявність файла на диску: якщо файл є,
  // то він відкривається для зчитування і записування, а якщо немає –
  // то створюється для зчитування і записування.
  if(FileExists(s)) f = fopen(s,"rt+");
    else f = fopen(s,"wt+");
  fclose(f);
}
// Кнопка "Дописати до файла"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ if((f=fopen(s,"at"))==NULL)
  { ShowMessage("Файл не вдається відкрити"); return;  }
  char nazva[15];
  strcpy(nazva, Edit1->Text.c_str());
  float price = StrToFloat(Edit2->Text);
  int kol = StrToInt(Edit3->Text);
  // Дописування даних здійснюється в кінець файла
  fprintf(f, "%s %6.2f %i\n", nazva, price, kol);
  fclose(f);
  Edit1->Clear(); Edit2->Clear(); Edit3->Clear();
  Edit1->SetFocus();
}
// Кнопка "Переглянути файл"
void __fastcall TForm1::Button2Click(TObject *Sender)
 { float price; int kol; char nazva[15];
  if((f=fopen(s,"rt+"))==NULL)
  { ShowMessage("Файл не вдається відкрити");
    return;
  }
  Memo1->Clear();
  // Контроль досягання кінця файла здійснюється функцією feof(f)
  while (!feof(f))
  { // Порядко́ве зчитування з файла
    fscanf(f,"%s %f %i\n",&nazva,&price,&kol);
    AnsiString str=nazva;
    for(int i=str.Length();i<=15; i++)str+=" ";
    // Виведення даних здійснюється за форматом
    Memo1->Lines->Add(str+" "+FloatToStrF(price,ffFixed,6,2)+
                          " "+IntToStr(kol));
  }
  fclose(f);
 }
// Кнопка "Товари, кількість яких є менше за 30"
void __fastcall TForm1::Button3Click(TObject *Sender)
{  if((f=fopen(s,"rt"))==NULL)
    { ShowMessage("Файл не вдається відкрити "); return; }
   Memo2->Clear();
   char nazva[15];
   float price; int kol;
```

```
   while(!feof(f))
    { fscanf(f,"%s %f %i\n",&nazva,&price,&kol);
      if (kol<30)Memo2->Lines->Add(AnsiString(nazva)+"  "
        +FloatToStrF(price,ffFixed,6,2)+"  "+IntToStr(kol));
    }
   fclose(f);
 }
// Кнопка "Визначити назву та розмір файла"
void __fastcall TForm1::Button4Click(TObject *Sender)
{  long length;
   if((f=fopen(s,"rt"))==NULL)
    { ShowMessage("Файл не вдається відкрити"); return; }
   fseek(f, 0, SEEK_END);
   length = ftell(f);
   fclose(f);
   Edit4->Text=s;
   Edit5->Text=IntToStr(length);
}
```

Приклад 12.2 Увести до текстового файла рядки символів. Прочитати дані з файла і з'ясувати, скільки разів зустрічається у файлі слово, яке вводиться з клавіатури.

Форма додатка з результатами роботи матиме вигляд



Текст програми:

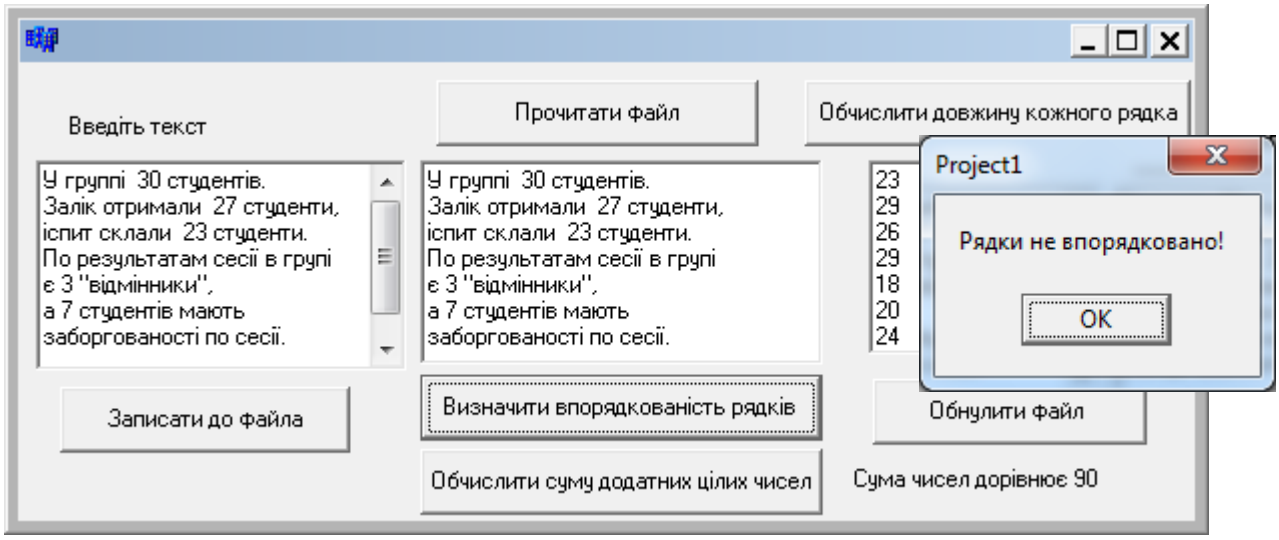
```
#include <stdio.h>
FILE *f;
int flag;
```

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{// Перевірка наявності файла на диску і визначення розміру файла
  if(FileExists("str.txt")) f=fopen("str.txt","rt+");
  else  f=fopen("str.txt","wt+");
  fseek(f,0,2);   flag=ftell(f); // Визначення розміру файла
  fclose(f);
 }
// Кнопка "Дописати рядок"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ f=fopen("str.txt","at");
  char s[80];
  strcpy(s, Edit1->Text.c_str());
  fprintf(f,"%s\n",&s);
  fclose(f);
  flag=1;
}
// Кнопка "Прочитати файл"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ if (flag==0)
  { ShowMessage("Файл є порожній!\nНемає чого читати!");
    return;
  }
  else  Memo1->Lines->LoadFromFile("str.txt");
}
// Кнопка "Підрахувати кількість шуканих слів у файлі"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ char s[80], key[20];  int k=0;
  strcpy(key,Edit2->Text.c_str());//Зчитування шуканого слова з Edit2
  if ((f=fopen("str.txt","rt"))==NULL)
  { ShowMessage("Неможливо прочитати файл");
    return;
  }
  char *str = " ,.:;!?" ;          // Рядок з розділювачами слів
  while (fgets(s,80,f))
  { if(s[strlen(s)-1]=='\n')
      s[strlen(s)-1]= '\0';
    char *p=new char [strlen(s)+1];
    // Виокремлення з рядка окремих слів
    p=strtok(s,str);
    while (p!=NULL)
    { if(!strcmp(p,key)) k++;
      p=strtok(NULL,str);
    }
  }
  Edit3->Text=IntToStr(k);
  fclose(f);
}
```

Приклад 12.3 Увести рядки до компонента Memo і записати їх до файла. прочитати дані з файла і відобразити їх в іншому Memo. Визначити:

- ✓ довжину кожного рядка;
- ✓ впорядкованість за зростанням довжин рядків (вивести відповідне повідомлення);
- ✓ суму всіх додатних цілих чисел, що зустрічаються;
- ✓ надати можливість обнулити файл (вилучити вміст файла).

Форма додатка з результатами роботи матиме вигляд



Текст програми:

```
#include <stdio.h>
FILE *f;   char fn[]="pr.txt";
// Кнопка "Записати до файла"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ f=fopen(fn,"wt");
  if(f==0) { ShowMessage("Не відкритий!"); return; }
  char s[40];  int n=Memo1->Lines->Count;
  for(int i=0;i<n;i++)
  { if(Memo1->Lines->Strings[i]=="") continue;
    strcpy(s,Memo1->Lines->Strings[i].c_str());
    strcat(s,"\n");
    fputs(s,f);
  }
  fclose(f);
}
// Кнопка "Прочитати файл"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ char s[40];    Memo2->Clear();
  f=fopen(fn,"rt+");
  if(f==0) {ShowMessage("Неможливо прочитати файл"); return;}
  while(fgets(s,40,f))
  { if(s[strlen(s)-1]=='\n') s[strlen(s)-1]='\0';
    Memo2->Lines->Add((AnsiString) s);
  }
```

```
  fclose(f);
}
// Кнопка "Обчислити довжину кожного рядка"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ char s[40];
  int dl;
  Memo3->Clear();
  f=fopen(fn,"rt+");
  if(f==0) {ShowMessage("Неможливо прочитати файл");return;}
  while (fgets(s,40,f))
  { if(s[strlen(s)-1]=='\n') s[strlen(s)-1]='\0';
    dl=strlen(s);
    Memo3->Lines->Add(IntToStr(dl));
  }
  fclose(f);
}
// Кнопка "Визначити впорядкованість рядків"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ char s[40];
  int dl1,dl2;
  f=fopen(fn,"rt+");
  if(f==0) {ShowMessage("Неможливо прочитати файл");return;}
  fgets(s,40,f);
  if(ftell(f)>0)
  { if(s[strlen(s)-1]=='\n') s[strlen(s)-1]='\0';
    dl1=strlen(s);
    int flag=0;
    while (fgets(s,40,f))
    { if (s[strlen(s)-1]=='\n') s[strlen(s)-1]='\0';
      dl2=strlen(s);
      if (dl1>dl2){flag=1; break;}
      dl1=dl2;
    }
    if(flag==0)ShowMessage("Рядки впорядковано за зростанням!");
    else ShowMessage("Рядки не впорядковано!");
  }
  fclose(f);
}
// Кнопка "Обчислити суму додатних цілих чисел"
void __fastcall TForm1::Button5Click(TObject *Sender)
{ char s[40];
  if ((f=fopen(fn,"rt"))==NULL)
  { ShowMessage("Неможливо прочитати файл"); return; }
  int num, sum=0;
  char *str=" ,.:;!?" ;
  while(fgets(s,40,f))
  { if(s[strlen(s)-1]=='\n')  s[strlen(s)-1]= '\0';
    char *p=new char [strlen(s)+1];
    p=strtok(s,str);
```

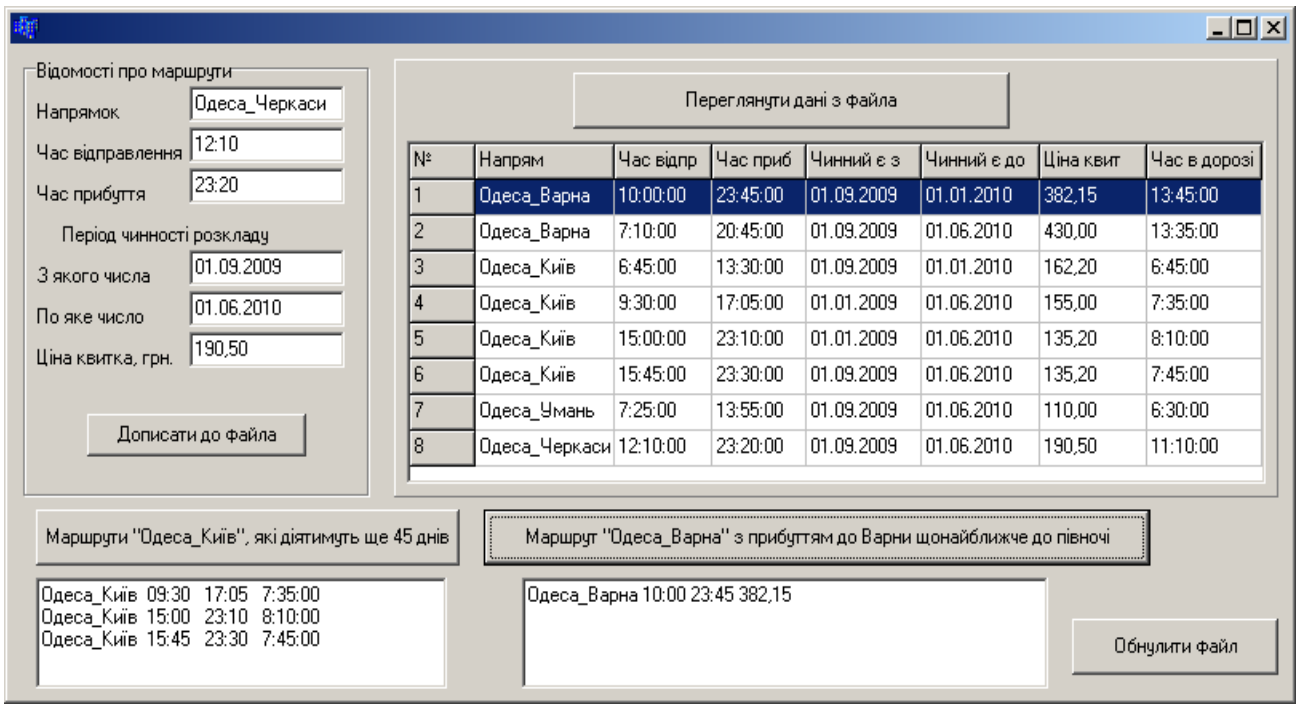
```
      while(p!=NULL)
      { num=atoi(p);
        // Функція  atoi() повертає число, якщо рядок p складається з цифр, або 0
        sum+=num;
        p=strtok(NULL,str);
    } }
    Label2->Caption="Сума чисел дорівнює "+IntToStr(sum);
    fclose(f);
}
// Кнопка "Обнулити файл"
void __fastcall TForm1::Button6Click(TObject *Sender)
{ f=fopen(fn,"wt");
  if(f==0) {ShowMessage("Не відкритий!");return;}
  fclose(f);
}
```

Приклад 12.4 Увести дані про автобусні маршрути по автостанції: напрямок маршруту, час відправлення та прибуття, період чинності даного розкладу за напрямками, ціна квитка. Записати дані до текстового файла і виконати такі завдання:

- ✓ вивести всі дані з інформацією про тривалість подорожі;
- ✓ вивести відомості про маршрути "Одеса – Київ", якщо цей розклад буде чинним ще 45 днів від поточного дня;
- ✓ віднайти маршрут "Одеса – Варна", для якого можливе прибуття до Варни після 17:00 і щонайближче до півночі.

Примітка. Вводити дані щодо напрямку маршруту треба без пропусків або сполучати за допомогою символу "підкреслення".

Можливий варіант робочої форми може мати вигляд



Текст програми:

```
#include <stdio.h>
// Оголосимо глобальні змінні
FILE *f;
char *fn="a.txt";
char npr[50], totp[9], tpri[9], dot[11], dpri[11];
float cena;
void __fastcall TForm1::FormCreate(TObject *Sender)
{ SG1->RowCount=2;
  SG1->Cells[0][0]="№";
  SG1->Cells[1][0]="Напрям";
  SG1->Cells[2][0]="Час відпр";
  SG1->Cells[3][0]="Час приб";
  SG1->Cells[4][0]="Чинний є з" ;
  SG1->Cells[5][0]="Чинний є до";
  SG1->Cells[6][0]="Ціна квит";
  SG1->Cells[7][0]="Час в дорозі";
}
// Кнопка "Дописати до файла"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ f=fopen(fn,"at+");
  if (f==0) { ShowMessage("Не відкрито!"); return; }
  strcpy(npr,Edit1->Text.c_str());
  strcpy(totp,Edit2->Text.c_str());
  strcpy(tpri,Edit3->Text.c_str());
  strcpy(dot,Edit4->Text.c_str());
  strcpy(dpri,Edit5->Text.c_str());
  cena=StrToFloat(Edit6->Text);
  fprintf(f,"%s %s %s %s %s %5.2f\n",
            npr,totp,tpri,dot,dpri,cena);
  fclose(f);
}
// Кнопка "Переглянути дані з файла"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ f=fopen(fn,"rt+");
  if(f==0){ShowMessage("Файл не відкритий!");return;}
  int i=0;      // Порядковий номер маршруту
  while(fscanf(f,"%s %s %s %s %s %f",
                &npr,&totp,&tpri,&dot,&dpri,&cena)>0)
  { i++;
    SG1->Cells[0][i]=IntToStr(i);
    SG1->Cells[1][i]=npr;
    SG1->Cells[2][i]=TTime(totp);
    SG1->Cells[3][i]=TTime(tpri);
    SG1->Cells[4][i]=TDate(dot);
    SG1->Cells[5][i]=TDate(dpri);
    SG1->Cells[6][i]= FormatFloat("0.00",cena);
    SG1->Cells[7][i]=TTime(tpri)-TTime(totp);
```

```
    SG1->RowCount=i+1;
  }
  fclose(f);
}
// Кнопка "Маршрути "Одеса_Київ", які діятимуть ще 45 днів"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ String napr;
  f=fopen(fn,"rt+");
  if(f==0){ShowMessage("Файл не відкрито!");return;}
  Memo1->Clear();
  while(fscanf(f,"%s %s %s %s %s %f",
              &npr,&totp,&tpri,&dot,&dpri,&cena)>0)
  { napr=&npr[0];
    if ((napr=="Одеса_Київ")&& (TDate(Date()+45)<TDate(dpri)))
      Memo1->Lines->Add(AnsiString(npr)+"  "+totp+"  "+
                  tpri+"   "+(TTime(tpri) - TTime(totp)));
  }
  fclose(f);
}
//Кнопка "Маршрут "Одеса_Варна" з прибуттям до Варни щонайближче до півночі"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ Memo2->Clear();
  // Обмеження часу прибуття становить 17:00.
  TTime Ttmin=TTime("17:00");
  AnsiString s;
  f=fopen(fn,"rt+");
  if(f==0)
  { ShowMessage("Файл не відкрито!"); return;}
  while(fscanf(f,"%s %s %s %s %s %f",
          &npr,&totp,&tpri,&dot,&dpri,&cena)>0)
  { if(AnsiString(npr)=="Одеса_Варна")
     if(TTime(tpri)>(Ttmin))
      s = AnsiString(npr)+" "+ AnsiString(totp)+" "+
          AnsiString(tpri)+" "+FormatFloat("0.00",cena);
    Ttmin=TTime(tpri);
  }
  Memo2->Lines->Add(s);
  fclose(f);
}
// Кнопка "Обнулити файл"
void __fastcall TForm1::Button5Click(TObject *Sender)
{ f=fopen(fn,"wt");
  if(f==0)
    { ShowMessage("Файл не відкрито!");
      return;
    }
  fclose(f);
}
```

Введення даних до файла з числового масиву

Для записування елементів числового масиву до файла існує дві можливості.

1) Перетворити елементи масиву на С-рядки, які записати до файла за допомогою функції `fputs()`. Наприклад, записування до файла цілочисельного масиву з 10-ти елементів:

```
char s[80];
for(int i=0; i<10; i++)
{ itoa(a[i],s,10);      // Перетворення елемента на С-рядок,
  strcat(s, "\n");      // долучення символу переведення на новий рядок
  fputs(s, f);          // і записування елемента до файла.
}
```

У цьому прикладі кожен елемент перетворюється на рядок окремо від інших за допомогою функції `atoi()` (докладніше про цю функції див. розд. 7) і записується до файла. В результаті всі числа масиву буде записано до файла у стовпчик. За потреби записувати елементи в один рядок через пробіл слід змінити тіло циклу в такий спосіб:

```
for(int i=0; i<10; i++)
 { itoa(a[i], s, 10);        // Перетворюємо елемент на С-рядок,
   strcat(s, " ");           // додаємо пробіл після чергового числа
   fputs(s, f);              // і записуємо число з пробілом до файла.
 }
fputs("\n", f);
```

Якщо кількість елементів є не надто велика, можна спочатку створити рядок, до якого записати через пробіл всі елементи, а тоді записати цей рядок до файла:

```
char s[80]="", s1[10];
for(int i=0; i<10; i++) // До рядка долучається черговий елемент
 { itoa(a[i], s1, 10); strcat(s, s1); } // у символьній формі
strcat(s, "\n");
fputs(s, f);
```

2) Записати елемент до файла за допомогою функції форматованого введення даних `fprintf()`:

```
for(int i=0; i<10; i++)  fprintf(ft,"%6.2f\n", a[i]);
```

Також матрицю до файла можна записати у такі способи.

1) Кожен елемент матриці перетворити до С-рядка і записати до файла окремо, після завершення кожного рядка матриці приєднати символ <Enter>:

```
for(int i=0; i<10; i++)
{ for(int j=0; j<10; j++)
  { itoa(a[i][j],s,10);// Перетворення числа на С-рядок,
    strcat(s,"\t");     // долучення табуляції після чергового числа
    fputs(s,f);  }      // і записування числа з пробілом до файла
  fputs("\n",f); //Записування символу переведення курсора у кінець рядка
}
```

2) Сформувати С-рядок з елементів кожного рядка матриці й записати його до файла:

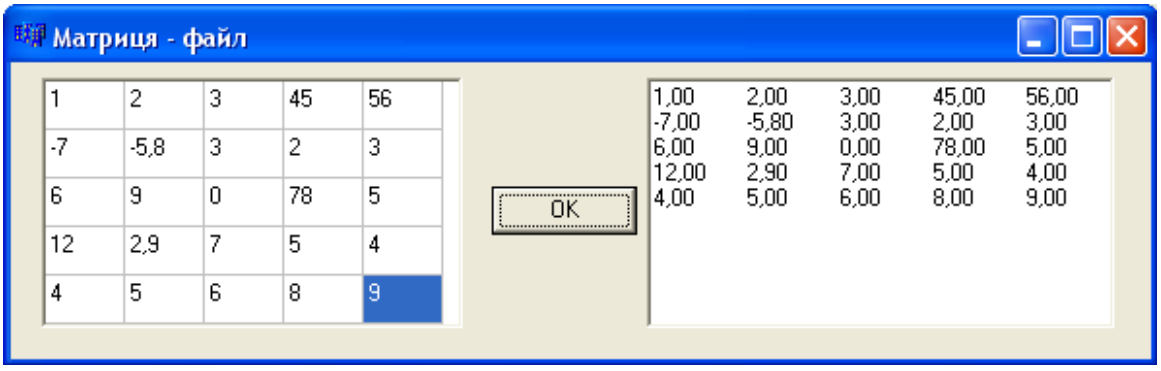
```
char s[80]="";
for(int i=0; i<10; i++)
{ for(int j=0; j<10; j++)
  { itoa(a[i][j],s,10);// Перетворення елемента на С-рядок,
    strcat(s, "\t");    // долучення табуляції після чергового числа
  }
  strcat(s, "\n");// і символу переведення позиції на кінець рядка
  fputs(s, f);     // Записування рядка матриці до файла
}
```

3) Кожен елемент матриці записати до файла у числовій формі за допомогою функції `fprintf()`:

```
for(int i=0; i<10; i++)// Записування елемента
{ for(int j=0; j<10; j++) fprintf(ft,"%6.2f\n",a[i][j]);
  fputs("\n", f);       // і символу переведення курсора у кінець рядка
}
```

Зверніть увагу на те, що функція `itoa()` перетворює ціле число на рядок. Для перетворення дійсних чисел на рядок можна перетворити число спочатку на `AnsiString` за допомогою однієї з відповідних функцій перетворювання, а тоді вже перетворити його на С-рядок.

Приклад 12.5 Заповнити матрицю 5×5 випадковими числами і записати цю матрицю до текстового файла.



Текст програми:

```
#include <stdio.h>
float a[5][5];
FILE* f;
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i,j,c;  char s[100];
  randomize();
  for(i=0; i<5; i++)
  for(j=0; j<5; j++)
  { StringGrid1->Cells[j][i]=((1000-random(2000))/10.);
    a[i][j]=StrToFloat(StringGrid1->Cells[j][i]);
  }
```

```
  if((f=fopen("my_text.txt", "wt"))==0)
  { ShowMessage("Файл не відкрито "); return; }
  for(int i=0; i<5; i++)
  { for(int j=0; j<5; j++)
    { // Перетворення елемента спочатку на AnsiString, потім на С-рядок.
      strcpy(s, FloatToStrF(a[i][j], ffFixed,6,2).c_str());
      strcat(s,"\t");      // Долучення табуляції після чергового елемента,
      fputs(s, f);         // записування числа з пробілом до файла
    }
    fputs("\n", f);        // і символу переведення курсора у кінець рядка
  }
  fclose(f);
  Memo1->Lines->LoadFromFile("my_text.txt");
}
```

Для виведення одновимірного масиву з текстового файла слід зчитати рядок і розбити його на числа, наприклад, за допомогою функції `strtok()`:

```
char s[80];
char *D=" ";          // Пробіл – розділювач поміж числами у рядку.
char *p=new char [80]; // Рядок для числа у символьній формі.
fgets(s,80,f);        // Зчитування рядка,
p=strtok(s,D);        // і виокремлення першого числа.
while (p)             // Допоки рядок не є порожній,
{ a[i]=atof(p);       // перетворюється дійсне число з символьної форми,
  i++;                // збільшується індекс елемента масиву
  p=strtok(NULL,D);   //  і виокремлюється наступне число.
}
```

Приклад 12.6 Заповнити матрицю 3×5 числами з текстового файла і обчислити суму елементів матриці.

Розв'язок. У Блокноті слід створити текстовий файл з ім'ям `matr.txt`, в якому записати елементи матриці. Форма з результатами роботи матиме вигляд

Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float m[3][5]; FILE *f;
  char *s=new char[50];
  int i=0, j; float sum=0;
  char *p=new char [50];
  if((f=fopen("matr.txt", "rt"))==0)
```

```
  { ShowMessage("Файл не відкрито "); return; }
  while(fgets(s,50,f))     // У циклі зчитуються рядки з файла.
  { j=0;                   // Розпочинаючи з нульового елемента,
    p=strtok(s," ");       // виокремлюється перше число
    while(p)               // і, допоки у рядку є числа,
    { m[i][j]=atof(p);     // перетворюється елемент з символьної форми,
      j++;                 // збільшується індекс стовпчика
      p=strtok(NULL," "); // і виокремлюється наступне число
    }
   i++;    // Збільшення індекса рядка (перехід до нового рядка матриці)
  }
  fclose(f);
  for(int i1=0; i1<i; i1++)
  for(int j1=0; j1<j; j1++)
  { SG2->Cells[j1][i1] = FormatFloat("0.0",m[i1][j1]);
    sum+=m[i1][j1];
  }
  Edit1->Text=FormatFloat("0.0", sum);
}
```

12.2.3 Робота з текстовими файловими потоками у стилі C++

У C++ стандартна бібліотека містить три класи потоків для роботи з файлами:

ifstream	вхідні файлові потоки (для зчитування)
ofstream	вихідні файлові потоки (для записування)
fstream	двонапрямлені файлові потоки (для зчитування та записування)

При роботі з файлами цих класів треба долучати до програми заголовний файл <fstream.h>.

Об'єкти файлових потоків створюються за допомогою конструкторів відповідних класів, наприклад:

```
// Створення (відкриття) вихідного потоку (записування)
ofstream outfile("Test.dat");
// Створення (відкриття) вхідного потоку (зчитування)
ifstream fin ("Test.dat");
// Створення (відкриття) введення-виведення (записування і зчитування)
fstream f_in_out("Test.dat");
```

До створених у такий спосіб потоків можна застосовувати операції "помістити в потік" (<<) і "взяти з потоку" (>>). Перевага цих операцій, які працюють з текстовими файлами, порівняно з розглянутими у попередніх розділах функціями, є простота використання і автоматичне розпізнавання типів даних. Розглянемо, наприклад, такий код:

```
int  i=1, j=25, il, j1; double a=25e6, al;
char  s[40], sl[40];
strcpy(s, "Іванов");
ofstream fout ("Test.dat"); // Створення файла як вихідного потоку
```

```
if(!fout){ShowMessage("Файл не вдається створити"); return;}
fout << i << ' ' << j << ' ' << a << ' ' << s <<endl;
fout.close();                 // Закриття файла
ifstream fin("Test.dat");     // Відкриття файла як вхідного потоку
if(!fin) {ShowMessage("Файл не вдається відкрити"); return;}
fin >> i1 >> j1 >> a1 >> s1;
fin.close();                  // Закриття файла
```

У цьому коді створюється файл з ім'ям `Test.dat`, і до нього записуються у текстовому вигляді два цілі числа – `i` та `j`, дійсне число `a` й рядок `s`, який містить одне слово, після чого маніпулятором потоку `endl` здійснюється перехід до нового рядка. Причому записування всіх цих даних здійснюється одним оператором, що містить зчеплені операції "помістити в потік". Якщо порівняти це з аналогічними кодами, наведеними у попередніх підрозділах, то можна переконатися в компактності й простоті застосування цієї операції. Після того як файл закриється, в ньому буде записано текст `"1 25 2.5e+07 Іванов"`. Наступні команди створюють вхідний потік, пов'язаний з цим файлом, і за допомогою операції "узяти з потоку" читають дані.

Як було зазначено раніш, файл треба описати. У C++ це можна здійснити в такий спосіб:

```
ofstream filename("C:\Test.dat", ios::out);
```

Тут у подвійних лапках зазначено ім'я файла на диску `C`, а `filename` – ім'я файлової змінної, тобто ім'я, за допомогою якого здійснюватиметься відкривання файла у програмі. Якщо файл не існує, то його буде створено. Далі, після коми, слід зазначити режим відкривання файла (табл. 12.3).

Таблиця 12.3

Режими відкривання файлів

Режим	Значення режиму
`ios::in`	Відкрити файл для зчитування
`ios::out`	Відкрити файл для записування
`ios::trunc`	Вилучити дані з файла
`ios::nocreate`	Якщо файл не існує, відкривання файла не відбудеться
`ios::ate`	Після відкриття перемістити позицію зчитування-записування на кінець файла
`ios::app`	Відкрити для записування даних в кінець файла
`ios::binary`	Відкрити файл, вважаючи його за бінарний (не текстовий)

Режими `ios::ate` та `ios::app` є схожими, оскільки переміщують позицію зчитування-записування (файловий вказівник) на кінець файла. Відмінність поміж ними полягає в тому, що режим `ios::app` дозволяє дописувати дані лише в кінець файла, в той час як режим `ios::ate` просто переміщує вказівник на кінець файла за аналогією роботи С-функції `fseek(f,0,SEEK_END)`.

Імена констант відкриття є абревіатурами виконуваних дій: `app` – append (долучити), `ate` – at end (на кінець), `trunc` – trancate (урізати, відкинути) тощо.

Режими відкриття файла за суттю є бітовими масками, а тому можна задавати два та більше режимів, поєднуючи їх операцією логічного “АБО”. Наприклад, щоб відкрити бінарний файл з ім’ям `Test.dat` як для зчитування, так і для записування, слід записати функцію

```
fstream filename("Test.dat",ios::in|ios::out|ios::binary);
```

Тут символ “|” поєднує два режими `ios::in` та `ios::out`.

В табл. 12.4 наведемо відповідність поміж режимами відкривання в C++ та C.

Таблиця 12.4

Відповідність режимів відкривання файлів в C++ і C

Режим C++	Режим C	Значення режиму
`ios::in`	`"r"`	Відкрити файл для зчитування. Позиція встановиться на початок файла
`ios::out`	`"w"`	Відкрити файл для записування. Позиція встановиться на початок файла. Якщо файл існує, записування здійснюватиметься поверх даних, тобто попередньо відбудеться урізання довжини файла до нуля
`ios::out | ios::trunc`		
`ios::out | ios::app`	`"a"`	Відкрити файл для записування даних в кінець файла
`ios::in | ios::out`	`"r+"`	Відкрити файл для зчитування-записування даних, починаючи з довільної позиції, тобто позиція може змінюватись
`ios::in | ios::out | ios::trunc`	`"w+"`	Відкрити файл для зчитування-записування, урізавши довжину файла до нуля для існуючого файла

Щоб перевірити правильність відкриття файла, можна скористатися умовним оператором

```
if(!filename)
 { ShowMessage("Неможливо відкрити файл!"); return; }
```

Завершення роботи з файлом здійснюється за допомогою функції

```
filename.close();
```

12.2.4 Послідовне записування до файла і зчитування з файла

Для записування певної інформації до файла використовується операція “помістити в потік” (за аналогією з `cout<<`):

`filename` << *блок1* << *блок2* << ... << *блокN*;

Для зчитування з файла певної інформації використовується операція “взяти з потоку” (за аналогією з `cin>>`)

`filename` >> *блок1* >> *блок2* >> ... >> *блокN*;

Однак при читанні з файла за допомогою потоків зчитування здійснюється до пробілу чи символу нового рядка. Тому це зчитування придатне лише для

чисел та окремих слів. Для зчитування цілого рядка використовується функція `getline()`, яка має прототип

```
getline (char*, int, char='\n');
```

Наприклад, оператор

```
fin.getline(s, n);
```

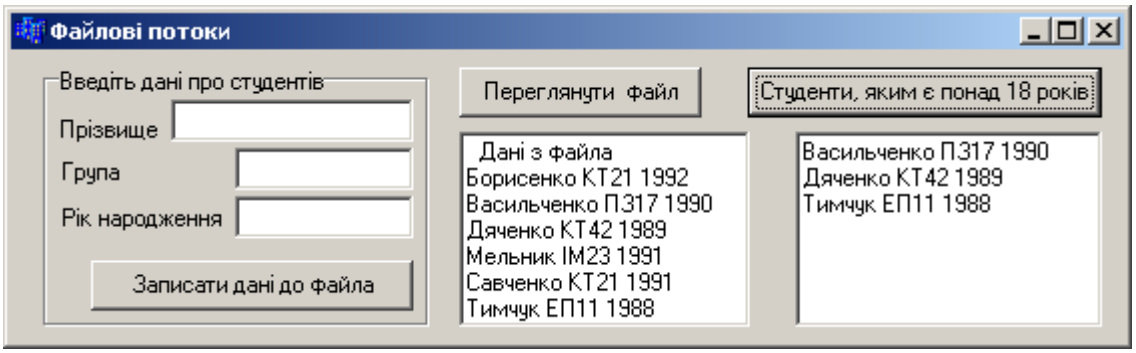
читає з потоку не більше за `n-1` символів й записує їх до змінної `s`.

Для роботи з файловим потоком слід до програми долучити бібліотеку `<fstream.h>`.

Розглянемо приклад, в якому показано створення файла, послідовне записування та зчитування даних.

Приклад 12.7 Увести інформацію про студентів: прізвище, найменування групи, рік народження. Переглянути весь список і вивести дані про студентів, яким на даний момент є понад 18 років.

Форма з результатами роботи матиме вигляд



Текст програми:

```
#include <fstream.h>
#include <DateUtils.hpp>  // Для функції визначення року YearOf()
// Кнопка "Записати дані до файла".
void __fastcall TForm1::Button1Click(TObject *Sender)
{ char PIB[50], gr[10];   int god;
  // Відкриття файла для записування даних у кінець файла
  ofstream output("dan.txt", ios::app);
  if(!output)  // Перевірка правильності відкриття файла для записування
  { ShowMessage("Файл не створено або не відкрито!"); return; }
  strcpy(PIB,Edit1->Text.c_str());
  strcpy(gr,Edit2->Text.c_str());
  god=StrToInt(Edit3->Text);
  output<<PIB<<' '<<gr<<' '<<god<<endl;  // Записування до файла
  output.close();                          // Закриття файла
  Edit1->Clear(); Edit2->Clear(); Edit3->Clear();
  Edit1->SetFocus();
}
// Кнопка "Переглянути файл"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ char *s=new char[80];
  Memo1->Lines->Add("\t Дані з файла");
  ifstream f("dan.txt");   // Відкриття файла для зчитування
```

```
  if(!f) { ShowMessage("Немає файла"); return;}
  while(f.getline(s,80)) Memo1->Lines->Add(AnsiString(s));
  f.close(); // Закриття файла
}
// Кнопка "Студенти, яким є понад 18 років"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ char PIB[50], gr[10];   int god;
  Memo2->Clear();
  fstream f("dan.txt",ios::in); // Відкриття файла для зчитування
  if(!f) { ShowMessage("Немає файла"); return;}
  while(f >> PIB >> gr >> god)
   if(YearOf(Date())-god > 18)
     Memo2->Lines->Add(String(PIB)+" "+String(gr)+" "+
                                                IntToStr(god));
  f.close();
}
```

12.2.5 Довільне записування до файла і зчитування з файла

Розглянемо варіант довільного записування і зчитування даних. Це означає, що записування даних до файла і зчитування цих даних здійснюватимуться з довільної зазначеної позиції файла. Позиціонування (встановлення курсора на певну позицію) виконується за допомогою методу `seekp()` класу `ifstream` при записуванні даних до файла, і за допомогою методу `seekg()` класу `ofstream` – при зчитуванні даних з файла. Ці методи мають два аргументи: перший зазначає на скільки байтів відносно позиції, зазначеної у другому аргументі, слід зрушитися. Існують такі константні значення позицій:

`ios::beg` – початок потоку (встановлюється за замовчуванням);

`ios::end` – кінець потоку;

`ios::cur` – поточна позиція потоку.

Наприклад:

`f.seekg(n);` – позиціонування на `n`-й байт від початку файла;

`f.seekg(n, ios::cur);` – позиціонування на `n`-й байт уперед від поточної позиції;

`f.seekp(k, ios::end);` – позиціонування на `k`-тий байт від кінця файла;

`f.seekp(0, ios::end);` – позиціонування на кінець файла.

Методи `tellg()` та `tellp()` повертають поточну позицію файла для потоків введення та виведення відповідно.

Для створювання текстових файлів довільного доступу існує багато способів. Мова C++ не накладає вимог щодо вмісту текстових файлів, але записані у файлі рядки мають бути однакової фіксованої довжини. Це надає можливість легко визначати точне місцезнаходження будь-якого рядка відносно початку файла. У такому файлі дані може бути записано в будь-яке місце та змінено без перезаписування всього файла. Для організації такого записування використовується маніпулятор `setw(num)`, який задає довжину поля виведення з `num` по-

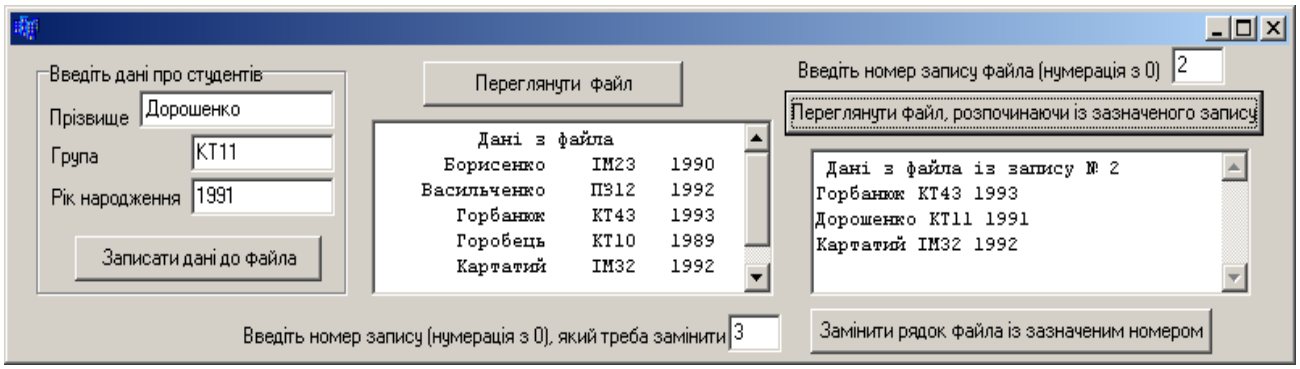
зицій. Для його використання слід долучити заголовний файл <iomanip.h>. Приклад його використання:

```
fout<<setw(15)<<name<<setw(20)<<surname<<setw(7)<<year<<endl;
```

Приклад 12.8 Записати до файла дані про студентів: прізвище, найменування групи, рік народження. Запрограмувати можливість змінювати у файлі той рядок (запис), номер якого зазначить користувач у текстовому полі (зауважимо, що рядки нумеруються з 0). Надати можливість переглядання файла, розпочинаючи з зазначеного запису, – номер запису теж задає користувач.

Розв'язок. Це завдання є подібне до прикладу 12.7, але для його реалізації слід інакше організувати записування даних до файла. Зробимо це із використанням маніпулятора setw, який задає ширину виведення змінної у cout, (аналогічно до табуляції). Для використання маніпуляторів треба долучити до програми заголовний файл iomanip.h.

Форма проекту з результатами роботи матиме вигляд



Текст програми:

```
#include <iomanip.h>
#include <fstream.h>
// Кнопка "Записати дані до файла"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ char PIB[50], gr[10];   int god;
  // Відкриття файла для записування даних у кінець файла
  ofstream output("dan.txt", ios::app);
  if(!output) // Перевірка правильності відкриття файла для записування
  { ShowMessage("Файл не створено або не відкрито!"); return; }
  strcpy(PIB,Edit1->Text.c_str());
  strcpy(gr,Edit2->Text.c_str());
  god=StrToInt(Edit3->Text);
  // Записування до файла з використанням маніпулятора setw
  output<<setw(15)<<PIB<<setw(8)<<gr<<setw(7)<<god<<endl;
  output.close();       // Закриття файла
  Edit1->Clear(); Edit2->Clear(); Edit3->Clear();
  Edit1->SetFocus();
}
// Кнопка "Переглянути файл"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ char *s=new char[80];
```

```
  Memo1->Clear();
  Memo1->Lines->Add("\t Дані з файла");
  ifstream f("dan.txt");   // Відкриття файла для зчитування
  if(!f) { ShowMessage("Немає файла"); return;}
  while(f.getline(s,80)) Memo1->Lines->Add(AnsiString(s));
  f.close();           // Закриття файла
}
// Кнопка "Замінити рядок файла із зазначеним номером
void __fastcall TForm1::Button3Click(TObject *Sender)
{ char PIB[50], gr[10];  int god;
  ofstream output("dan.txt",ios::in|ios::out);
  if(!output)
  { ShowMessage("Файл не створено або не відкрито!"); return; }
  int k=StrToInt(Edit4->Text);
  // Уведення нових значень k-го рядка.
  strcpy(PIB,Edit1->Text.c_str());
  strcpy(gr,Edit2->Text.c_str());
  god=StrToInt(Edit3->Text);
  output.seekp(k*32);        // Позиціонування у k-тий рядок файла
  // Записування з використанням маніпулятора setw
  output<<setw(15)<<PIB<<setw(8)<<gr<<setw(7)<<god<<endl;
  output.close();
}
// Кнопка "Переглянути файл, розпочинаючи із зазначеного запису"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ int k=StrToInt(Edit5->Text);      // Введення номера запису
  char PIB[50], gr[10];  int god;
  Memo2->Clear();
  Memo2->Lines->Add(" Дані з файла із запису № "+IntToStr(k));
  ifstream fin("dan.txt");
  if(!fin) { ShowMessage("Немає файла"); return;}
  fin.seekg(k*32);           // Позиціонування у k-тий рядок файла
  while(fin>>PIB>>gr>>god)   // Читаються дані з файла
    Memo2->Lines->Add(String(PIB)+" "+String(gr)+" "+
                      IntToStr(god));
  fin.close();
}
```

12.2.6 Опрацювання текстових файлів за допомогою дескрипторів

При створюванні програм для Windows має сенс використовувати стандартні функції цієї системи. Для роботи з ними не треба створювати спеціальні змінні файлового типу, оскільки в системі Windows кожен файл має власний ***дескриптор*** – унікальний ідентифікатор цілого типу. Англійською він зветься **`Handle`** і під такою назвою є присутній в прототипах багатьох функцій.

Основні функції C++ Builder, які використовуються для опрацювання файлів через дескриптори, наведено у табл. 12.5.

Таблиця 12.5

Функції опрацювання файлів через дескриптори

Формат	Опис функції
`int` **`FileCreate`** `(AnsiString FileName);`	Створює файл з ім'ям `FileName` і повертає дескриптор створеного файла, який можна в подальшому використовувати для роботи з цим файлом. Якщо файл не вдалося створити, функція повертає значення –1
`int` **`FileOpen`** `(AnsiString FileName, unsigned Mode);`	Відкриває файл з ім'ям `FileName` у режимі `Mode` і повертає дескриптор відкритого файла
`void` **`FileClose`**`(int Handle);`	Закриває файл з дескриптором `Handle`
`int` **`FileRead`**`(int Handle, void *Buffer, unsigned Count);`	Читає з файла з дескриптором `Handle` данні з `Buffer` розміром `Count` байтів і повертає значення прочитаних байтів; у разі помилки повертає –1
`int` **`FileWrite`**`(int Handle, void *Buffer, unsigned Count);`	Записує до файла з дескриптором `Handle` данні з `Buffer` розміром `Count` байтів і повертає значення записаних байтів; у разі помилки повертає –1
`int` **`FileSeek`**`(int Handle, int Offset, int Origin);`	Переміщує поточну позицію файла на `Offset` байтів відносно позиції визначеної параметром `Origin`, можливі значення якого розглянуто нижче
`bool` **`DeleteFile`** `(AnsiString FileName);`	Видаляє файл з ім'ям `FileName`. Якщо файл не можливо видалити чи він не існує, функція повертає значення `false`, інакше – `true`
`bool` **`RenameFile`** `(AnsiString OldName, AnsiString NewName);`	Перейменовує файл, тобто змінює ім'я `OldName` на `NewName`. Якщо перейменувати файл не вдалося, функція повертає значення `false`, інакше – `true`

Розглянемо ці функції більш докладно на прикладах.

Файл створюється за допомогою функції `FileCreate()`, наприклад:

```
int f = FileCreate("info.txt");
```

Режими створювання і відкривання файла у Windows відрізняються. Функція `FileCreate()` повертає ідентифікатор файла (ціле додатне число) або значення –1, якщо створити файл не вдалось. Значення результату –1 свідчить про помилку для більшості функцій Windows.

Відкривання файла виконується функцією

```
int FileOpen(const AnsiString FileName, unsigned Mode);
```

Параметр `FileName` містить ім'я файла, можливо, із зазначенням місця розташування на диску цього файла.

Режим відкривання визначається параметром `Mode`, який може набувати одне з таких константних значень:

- ✓ `fmOpenRead` – відкрити лише для зчитування з файла;
- ✓ `fmOpenWrite` – відкрити лише для записування у файл;
- ✓ `fmOpenReadWrite` – відкрити і для зчитування і для записування (режим редагування файла);
- ✓ `fmCreate` – за умови наявності файла, відкрити його для записування, інакше – створити новий файл. На відміну від інших констант, які означено в модулі <`SysUtils`>, цю константу означено у <`classes.h`>.

Наприклад, команда

```
int f = FileOpen("info.txt", fmOpenWrite);
```

відкриє файл з ім'ям `info.txt` у режимі записування даних.

Закрити цей файл можна за допомогою функції

```
FileClose(f);
```

Для переміщення поточної позиції файла застосовують функцію

```
int FileSeek(int Handle, int Offset, int Origin);
```

Тут параметр `Offset` задає зсув у байтах відносно позиції визначеної параметром `Origin`.

Значення параметра `Origin`	**Спосіб відліку**
0	від початку файла
1	від поточної позиції
2	від кінця файла

Наприклад, функція

```
FileSeek(f,0,0);
```

встановить поточну позицію на початок файла, а функція

```
FileSeek(f,1,1);
```

здійснить переміщення позиції на наступний символ (байт). Застосовувати такі переміщення вказівника слід обережно, щоб не вийти за межі файла.

Розглянемо приклад відкриття файла з ім'ям `"test.txt"`, переміщення позиції на кінець файла і *дописування* рядка `st`:

```
int f;              // Оголошення дескриптора файла
AnsiString st="Цей рядок буде дописано в кінець файла";
f=FileOpen("test.txt", fmOpenWrite);
// Відкриття файла для зчитування і записування
FileSeek(f,0,2);  // Встановлення позиції на кінець файла
FileWrite(f, st.c_str(),st.Length());
// Записування рядка st довжиною st.Length() до файла
FileClose(f);       // Закриття файла
```

Відкрити файл *для зчитування* можна в такий спосіб:

```
f = FileOpen("test.txt", fmOpenReadWrite);
```

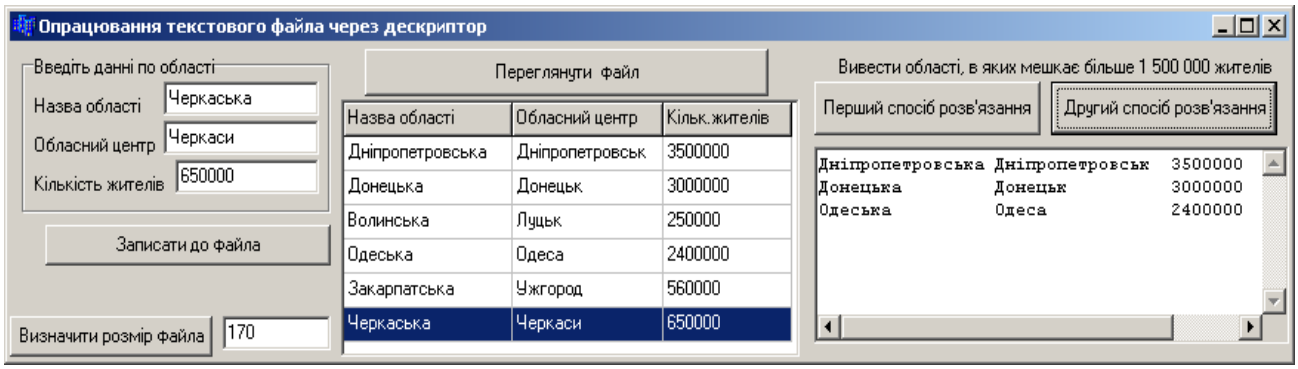
```
char buf[80];    // Рядок (буфер)
char *s = buf;   // і вказівник на цей рядок
int n = FileRead(f, s, 80);
FileClose(f);
```

Розглянемо приклад створення і опрацювання текстового файла через дескриптори.

Приклад 12.9 Створити файл з інформацією про чисельність населення областей України: назва області, обласний центр, кількість мешканців області. Вивести області, в яких кількість жителів перевищує 1 500 000 людей.

Розв'язок. При створюванні текстового файла усі дані, які зчитуватимуться з компонентів `Edit`, записуватимуться до файла через пробіл у вигляді рядків. Для подальшого опрацювання слів цього рядка вважатимемо, що кожен рядок завершується символами `"\r\n"`. Словом вважатимемо частину рядка, відокремлену пробілами. При опрацюванні файла, а саме, при зчитуванні даних, використовуватимемо функцію `FileRead()`, другим параметром якої є змінна типу `*char`, в яку записуватимуться данні, а третім – розмір у байтах зчитуваних даних.

Наведемо програмну реалізацію поставленого завдання двома способами, в яких проілюструємо різні підходи опрацювання рядків текстового файла. У першому способі для зчитування рядків з файла створено окрему функцію `GetLine()`, яка читає рядки посимвольно. Другий спосіб показує засоби зчитування рядка по словах у спеціально для цього створеній функції `GetString()`.



Текст програми:

```
int f;                          // Дескриптор оголошено глобально
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)      // Конструктор форми
{ SG1->Cells[0][0]="Назва області";
  SG1->Cells[1][0]="Обласний центр";
  SG1->Cells[2][0]="Кільк.жителів";
  SG1->ColWidths[0]=80;
  SG1->ColWidths[1]=70;
  SG1->ColWidths[2]=70;
} // Тут  SG1 – нове значення властивості  Name компонента StringGrid1
```

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{ if( FileExists("obl.txt") )// Перевірка наявності файла
    f = FileOpen("obl.txt",fmOpenWrite);// і відкриття його.
  else                              // Якщо файл є відсутній,
    f = FileCreate("obl.txt");// створити його.
  FileClose(f);
}
// Кнопка "Записати до файла".
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString st;
 if(Edit1->Text=="" || Edit2->Text=="" || Edit3->Text=="")
 { ShowMessage("Не всі данні введено!");
   Edit1->SetFocus();
   return;
 }
 st=Edit1->Text+" "+Edit2->Text+" "+
                          StrToInt(Edit3->Text)+ "\r\n";
 // Відкриття файла для зчитування і записування.
 f=FileOpen("obl.txt",fmOpenWrite);
 if(f != -1)          // Якщо помилок при відкритті файла не було,
 { FileSeek(f,0,2);   // позиція переміститься на кінець файла.
   FileWrite(f,st.c_str(),st.Length());
   FileClose(f);
 }
 else       // У разі помилки доступу до файла, виведеться повідомлення.
 { ShowMessage("Помилка доступу до файла"); return; }
 Edit1->Clear();  Edit2->Clear();  Edit3->Clear();
 Edit1->SetFocus();
}
// Для зчитування рядків з файла створено окрему функцію GetLine(),
// яка читає рядки символів від поточної позиції до символу "новий рядок"
// і повертає значення кількість прочитаних символів.
int GetLine(int f, AnsiString *st)
{ unsigned char buf[256]; // Оголошення рядка (буфера),
  unsigned char *p = buf; // вказівника на рядок,
  int n;                  // цілих змінних для кількості прочитаних байтів
  int len = 0;            // і довжини рядка.
  // У циклі читаються данні типу  char по одному байту, тобто посимвольно.
  while(n != 0)       // Допоки є що читати,
  { if(*p == '\r')    // перевіряється чи є зчитаний символ "кінцем" рядка.
    { n = FileRead(f, p, 1); // Прочитати символ '\n'.
      break;
    }
    len++;
    p++;
    n = FileRead(f, p, 1);
  }
  *p = '\0';            // Записування у рядок символу кінця рядка
```

```
  if(len != 0) st->printf("%s", buf);
  return len;
}
// Кнопка "Переглянути файл"
//  Відбувається порядкове неформатоване зчитування даних з файла і виведення до
//  SG1 за допомогою властивості DelimitedText, яка розбиває рядок на окремі лексеми
void __fastcall TForm1::Button2Click(TObject *Sender)
{ AnsiString st;
  bool d=true;
  f=FileOpen("obl.txt", fmOpenReadWrite);
  if(f == -1)
   { ShowMessage("Файл не відкрито"); return; }
  while(GetLine(f,&st) != 0)
   { if(d)
    { SG1->Rows[SG1->Row]->DelimitedText=st; d=false; }
    else
    { SG1->RowCount++;
      SG1->Row=SG1->RowCount-1;
      SG1->Rows[SG1->Row]->DelimitedText=st;
   }}
  FileClose(f);
}
// Кнопка "Визначити розмір файла"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ f=FileOpen("obl.txt",fmOpenReadWrite);
  if(f==-1)// У разі помилки доступу до файла, вивести відповідне повідомлення
   { ShowMessage("Файл не відкрито"); return; }
  int k=FileSeek(f,0,2);   // Переведення  позиції на кінець файла
  Edit4->Text=IntToStr(k);
  FileClose(f);
}
// Вивести області, в яких мешкає більше 1 500 000 жителів
// Кнопка "Перший спосіб розв'язання"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ f=FileOpen("obl.txt", fmOpenReadWrite);
  if(f == -1) { ShowMessage("Файл не відкрито"); return; }
  Memo1->Clear();
  char *buf, *stw, *p;     AnsiString st, bt;
  while(GetLine(f,&st) != 0)
  { stw=st.c_str();
    bt=AnsiString(stw);
    p=strpbrk(stw,"0123456789"); /* strpbrk() відшукує місце першого
        входження в рядок stw будь-якого символу з другого рядка і повертає частину
        рядка s1, починаючи з цього символу і до кінця рядка, тобто в p запишеться
        кінцівка рядка зі значенням кількості мешканців. */
    if((StrToInt(p)>=1500000)) Memo1->Lines->Add(bt);
  }
  FileClose(f);
}
```

```
// Для другого способу створимо функцію GetString() зчитування рядка st по словах,
// яка повертає кількість прочитаних символів,
// а параметр flag дозволяє визначити кінець рядка.
int GetString(int f, AnsiString *st, int &flag)
{ unsigned char buf[256]; // Оголошення рядка (буфера),
  unsigned char *p = buf; // вказівника на рядок,
  int n;                  // цілих змінних для кількості прочитаних байтів
  int len = 0;            // і довжини рядка.
  do n = FileRead(f, p, 1); while((n != 0) && (*p == ' '));
  while((n != 0) && (*p != ' '))
  { if(*p == '\r')
    { n=FileRead(f, p, 1); // Прочитати символ '\n'
      flag=1;
      break;
    }
    len++;    p++;
    n=FileRead(f, p, 1);
  }
  *p = '\0';
  if(len != 0) st->printf("%s", buf);
  return len;
}
// Кнопка "Другий спосіб розв'язання"
void __fastcall TForm1::Button5Click(TObject *Sender)
{ Memo1->Clear();
  AnsiString st, buf, strz="";
  int ls, flag=0;
  f=FileOpen("obl.txt", fmOpenReadWrite);
  if(f==-1) { ShowMessage("Файл не відкрито"); return; }
  do
  { ls=GetString(f,&st,flag);
    if(ls!=0)
    { buf=st;
      for(int i=st.Length();i<=15;i++) st=st+" ";
      strz += st+" ";
    }
    bool t=false;
    for(int i=1;i<=buf.Length();i++)
     if(!(buf[i]>='0'&& buf[i]<='9'))
      { t=true; break; }
    if(!t&&(StrToInt(buf)>=1500000))
      Memo1->Lines->Add(strz);
    if(flag==1)
      { strz="";
        flag=0;
      }
  } while(ls!=0);
  FileClose(f);
}
```

12.3 Бінарні файли

У *бінарному* (двійковому) файлі число, на відміну від текстового, зберігається у внутрішньому його поданні. У двійковому форматі можна зберігати не лише числа, а й рядки та цілі інформаційні структури. Причому останні зберігати зручніше, завдяки тому що відсутня потреба явно зазначати кожен елемент структури, а зберігається вся структура як цілковита одиниця даних. Хоча цю інформацію не можна прочитати як текст, вона зберігається більш компактно і точно. Тому, що саме і в якій послідовності розміщено в бінарному файлі, має бути відомо програмі.

12.3.1 Робота з бінарними файлами у стилі C

З двійковими файлами можна виконувати ті ж самі дії, що і з текстовими. Для відкривання бінарного файла використовується та сама команда `fopen()`, лише у другому параметрі (режимі відкривання файла) замість літери "`t`" треба записати літеру "`b`". Наприклад, бінарний файл з ім'ям `tmp.dat` можна відкрити для зчитування такою командою:

```
f = fopen("tmp.dat", "rb");
```

де `f` – вказівник типу `FILE*`.

Записування і зчитування у двійкових файлах найчастіше здійснюються за допомогою відповідно функцій `fwrite()` та `fread()`.

Прототип функції зчитування `fread()`:

```
size_t fread
( char *buffer,          // Масив для зчитування даних,
  size_t elemSize,       // розмір одного елемента,
  size_t numElems,       // кількість елементів для зчитування
  FILE *f                // і вказівник потоку
);
```

Тут `size_t` означений як беззнаковий цілий тип у системних заголовних файлах. Функція намагається прочитати `numElems` елементів з файла, який задається вказівником `f`, розмір кожного елементу становить `elemSize`. Функція повертає реальну кількість прочитаних елементів, яка може бути менше за `numElems`, у разі завершення файла чи то помилки зчитування.

Приклад використання функції `fread()`:

```
FILE *f;
double buff[100];
size_t res;
f = fopen ("tmp.dat", "rb");      // Відкриття файла
if(f == 0)
 {  ShowMessage ("Не можу відкрити файл для зчитування");
    exit (1);                     // Завершення роботи з кодом 1
 }
// Зчитування 100 дійсних чисел з файла
res = fread(buff, sizeof(double), 100, f);
// res дорівнює реальній кількості прочитаних чисел
```

У даному прикладі файл tmp.dat відкривається для зчитування як бінарний, з нього читаються 100 дійсних чисел розміром 8 байтів кожне. Функція fread() повертає реальну кількість прочитаних чисел, яка є менше чи то дорівнює 100.

Функція fread() читає інформацію у вигляді потоку байтів і в незмінному вигляді розміщує її в пам'яті. Слід розрізнювати текстове подавання чисел і їхнє бінарне подавання. У наведеному вище прикладі числа у файлі має бути записано у *бінарній формі*, а не у вигляді тексту. Для текстового введення чисел слід використовувати функції введення за форматом, які було розглянуто вище.

Функція бінарного записування до файла fwrite() є аналогічна до функції зчитування fread(). Вона має такий прототип:

```
size_t fwrite
( char *buffer,      // Масив записуваних даних,
  size_t elemSize,   // розмір одного елемента,
  size_t numElems,   // кількість записуваних елементів
  FILE *f            // і вказівник потоку
);
```

Функція повертає кількість реально записаних елементів, яка може бути менше за numElems, якщо при записуванні виникла помилка: приміром, не вистачило вільного простору на диску.

Приклад використання функції fwrite():

```
FILE *f;
double buff[100];
size_t res;
. . .                              // Введення елементів масиву buff[100]
f = fopen("tmp.dat", "wb");        // Створення бінарного файла
if(f == 0)
  { ShowMessage ("Не можу відкрити файл для запису");
    exit(1);                       // Завершення роботи програми з кодом 1
  }
// Записування 100 дійсних чисел до файла
res = fwrite(buff, sizeof(double), 100, f);
// У разі успішного записування  res = 100
```

До обох функцій передається вказівник на дані, які виводяться чи вводяться, параметр elemSize задає розмір передаваних даних в байтах, а параметр numElems визначає кількість таких даних.

Так само, як і в текстових, у бінарних файлах можна визначати позицію зчитування-записування і переміщувати її у довільне місце файла командами відповідно ftell() та fseek(). Можливість переміщувати позицію є корисна для файлів, які складаються з однорідних записів однакового розміру. Приміром, якщо у файлі записано лише дійсні числа типу double, то для того, щоб прочитати i-те число, слід виконати оператори:

```
fseek(F, sizeof(double)*(i-1), 0);
fread(&a, sizeof(double), 1, F);
```

У такий спосіб можна читати які завгодно записи в будь-якій послідовності. За допомогою переміщування позиції можна редагувати записи у файлі. Нехай, приміром, треба змінити у файлі одне з чисел, помноживши його на 10. Це можна зробити, якщо відкрити файл у режимі зчитування і записування (наприклад у форматі "rb+"), переміститись у відповідну позицію і виконати оператори

```
fread(&z, sizeof(double), 1, F);
z *= 10;
fseek(F, -sizeof(double), 1);
fwrite(&z, sizeof(double), 1, F);
```

Перший з цих операторів читає число з файла у дійсну змінну z типу double, другий – помножує її на 10. Третій оператор повертає позицію на один запис назад, оскільки після виконання fread() позиція просунулась уперед. Останній оператор пише в ту позицію, з якої було прочитано число, нове значення.

Те ж саме завдання можна розв'язати в інший спосіб:

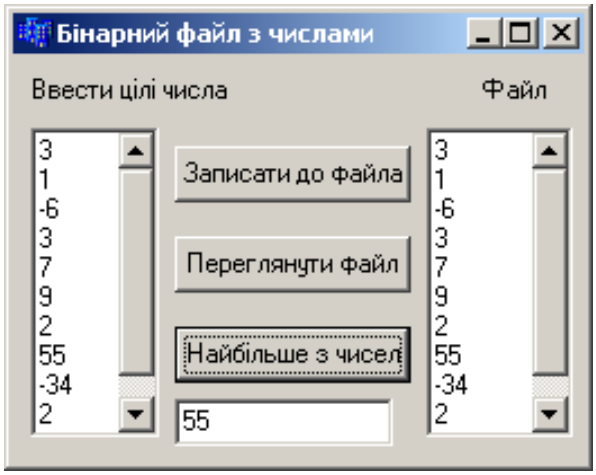
```
long int pos = ftell(F);        // Запам'ятовування позиці.
fread(&z,sizeof(double),1,F);// Зчитування числа з файла до змінної
z *= 10;
fseek(F,pos,0);                 // Відновлення позиції
fwrite(&z,sizeof(double),1,F);//Записування числа зі змінної z до файл
```

Тут функція ftell() визначає поточну позицію файла, з якої читається число, а функція fseek() відновлює цю позицію перед записуванням зміненого числа.

Як було зазначено вище, за допомогою двійкових файлів можна записувати й зчитувати не лише числа та рядки, а й структури (див. приклади 12.12 та 12.14).

Приклад 12.11 Створити бінарний файл, який містить цілі числа, і віднайти серед них найбільше число.

Форма проекту з результатами роботи матиме вигляд

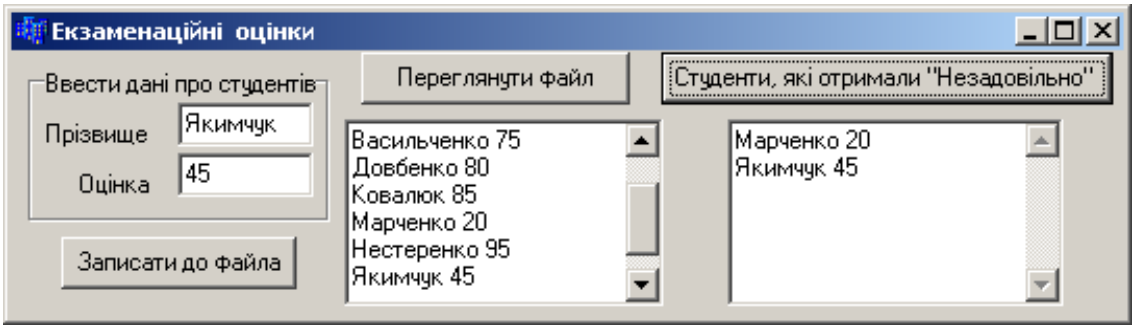


Текст програми:

```
#include <stdio.h>
FILE* f;
// Кнопка "Записати до файла".
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int z, i, N=Memo1->Lines->Count;
  if((f=fopen("num.dat", "wb"))==0) // Відкриття файла для створення
   { ShowMessage("Не вдається створити файл"); return;}
  for(i=0; i<N; i++)
   { z=StrToInt(Memo1->Lines->Strings[i]);//Читання числа z з Memo1
     fwrite(&z, sizeof(int),1,f);   // Записування числа z до файла
   }
  fclose(f);  }
// Кнопка "Переглянути файл"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ int z;   Memo2->Clear();
  if((f=fopen("num.dat","rb"))==0) // Відкриття файла для зчитування
   { ShowMessage("Не вдається відкрити файл"); return; }
  while(fread(&z, sizeof(int),1,f)) // Зчитування числа
    Memo2->Lines->Add(IntToStr(z));
  fclose(f);
}
// Кнопка "Найбільше з чисел"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ int z, max;
  if((f=fopen("num.dat","rb"))==0)// Відкриття файла для зчитування
   { ShowMessage("Не вдається відкрити файл"); return; }
  fread(&z, sizeof(int),1,f);        // Зчитування першого числа
  max= z;                            // і запам'ятовування  його в max
  while(fread(&z, sizeof(int),1,f)) // Зчитування числа
   if(max<x)  max=z;//Якщо число більше за max, воно запам'ятовується в max.
  fclose(f);
  Edit1->Text=IntToStr(max);
}
```

Приклад 12.12 Увести відомості про екзаменаційні оцінки студентів: прізвище та оцінка. Вивести відомості про студентів, які отримали "незадовільно" (менше за 60 балів).

Форма проекту з результатами роботи матиме вигляд



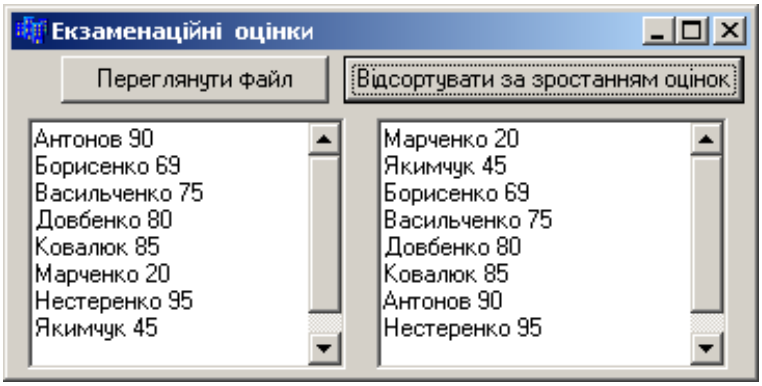
Текст програми:

```
#include <stdio.h>
FILE* f;
struct ved
{ char priz[20];
  int oc;
} v;
// Кнопка "Переглянути файл"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ Memo1->Clear();
  //Відкриття файла для зчитування
  if((f=fopen("ocinki.dat", "rb"))==0)
   { ShowMessage("Файл не відкрито"); return; }
  while(fread(&v, sizeof(ved),1,f)) //Зчитування з файла в змінну v
    Memo1->Lines->Add(AnsiString(v.priz)+" "+IntToStr(v.oc));
  fclose(f);
}
// Студенти, які отримали "Незадовільно"
void __fastcall TForm1::Button2Click(TObject *Sender)
{  Memo2->Clear();
  if((f=fopen("ocinki.dat", "rb"))==0)
    { ShowMessage("Файл не відкрито"); return; }
  while (fread(&v, sizeof(ved),1,f))
  { if(v.oc < 60)          //Якщо оцінка є менше за 60
      Memo2->Lines->Add(AnsiString(v.priz)+" "+IntToStr(v.oc));
  }
  fclose(f);
}
```

Приклад 12.13 Відсортувати дані файла з відомостями про екзаменаційні оцінки (який було створено у попередньому прикладі) за зростанням оцінок.

Розв'язок. Вважаємо, що файл `ocinki.dat` вже створено і розміщено у теці з проектом. Оскільки при сортуванні файла треба кілька разів переходити до початку файла, а всі операції з файлом виконуються набагато повільніше, аніж з масивом, є сенс записати всі дані з файла до динамічного масиву, відсортувати його, а потім повернути дані з масиву до файла.

Форма проекту з результатами роботи матиме вигляд



Текст програми:

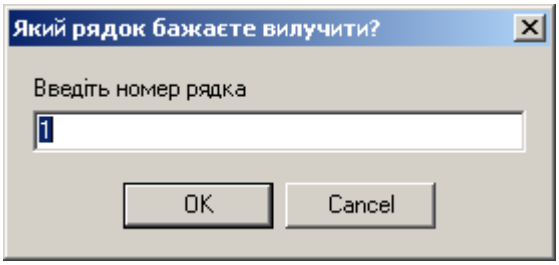
```
#include <stdio.h>
FILE* f;
struct ved
{ char priz[20];
  int oc;
} v;
//Кнопка "Переглянути файл"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ Memo1->Clear();
  if((f=fopen("ocinki.dat", "rb"))==0)
   { ShowMessage("Файл не відкрито"); return; }
  while (fread(&v, sizeof(ved),1,f))
    Memo1->Lines->Add(AnsiString(v.priz)+" "+IntToStr(v.oc));
  fclose(f);
}
//Кнопка "Відсортувати за зростанням оцінок"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ int K=0, i=0, j;   // K – кількість записів у файлі
  bool pr;           // Ознака – чи відсортовано масив
  if((f=fopen("ocinki.dat", "rb"))==0)
   { ShowMessage("Файл не відкрито"); return; }
  while(fread(&v, sizeof(ved),1,f))
   K++;              // Обчислення кількості записів у файлі
  fseek(f,0,0);      // Переміщення позиції на початок файла
  ved *a=new ved[K]; // Оголошення динамічного масиву
  while(fread(&v, sizeof(ved),1,f))
   { a[i]=v; i++; }  // Записування даних з файла до масиву
  fclose(f);
  do
   { pr=true;        // Припустімо, що масив є відсортовано
     for(i=0; i<K-1; i++)
      if(a[i].oc>a[i+1].oc)// Якщо сусідні елементи розташовано
        { ved tmp=a[i];     // у неналежному порядку,
          a[i]=a[i+1];      // переставляються елементи
          a[i+1]=tmp;
          pr=false;         // і "вимикається" ознака упорядкованості.
        }
   } while(!pr);
  if((f=fopen("ocinki.dat", "wb"))==0) // Створення файла
   { ShowMessage("Файл не відкрито"); return; }
  for(i=0; i<K; i++)                  // У циклі всі елементи масиву
   { fwrite(&a[i],sizeof(ved),1,f);// записуються до файла
     Memo2->Lines->Add(AnsiString(a[i].priz)+" "+
              IntToStr(a[i].oc));  // і виводяться на форму.
   }
 fclose(f);
}
```

Приклад 12.14 Створити бінарний файл, що містить інформацію про товари: найменування товару, ціна і кількість. Відібрати товари, кількість яких є менше за 30 одиниць. Відредагувати один із записів. Вилучити один із записів. Номер запису для редагування й вилучення ввести з клавіатури.

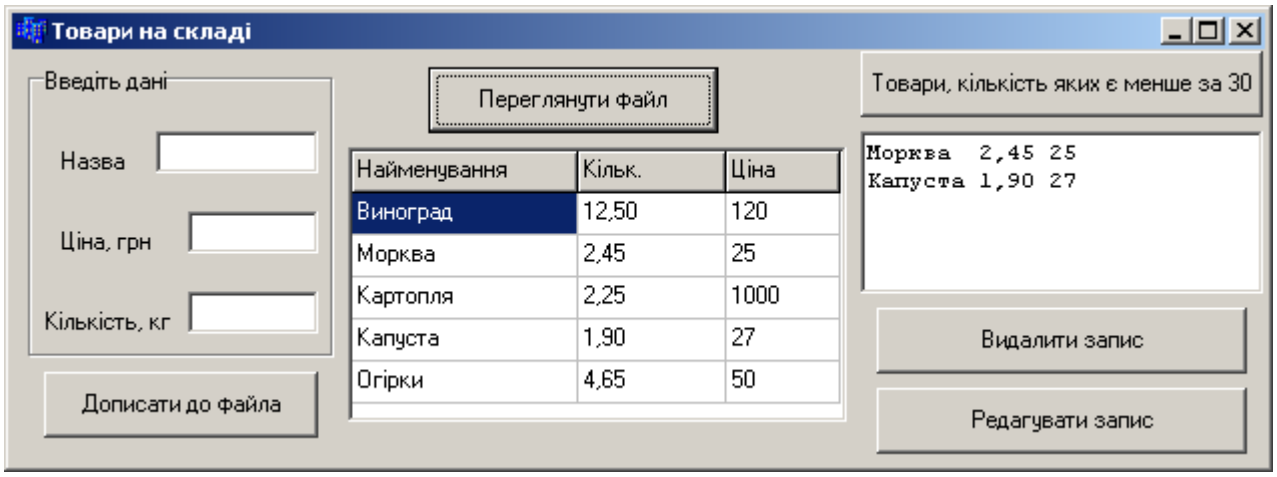
Розв'язок. Окрім зазначених алгоритмів опрацювання бінарного файла у цій програмі наведено засоби редагування і вилучення певного запису з файла. Номер такого запису вводиться за допомогою вікна введення `InputBox()`, синтаксис та приклад роботи якого розглядались у прикладі 6.7. Вікно введення, створене у цій програмі командою

```
AnsiString snom=InputBox("Який рядок бажаєте вилучити?",
          "Введіть номер рядка", "1");
```

матиме вигляд:



Вікно форми проекту з результатами роботи програми:



Текст програми:

```
int n, flag;
// При дописуванні даних у кінець файла  flag становитиме 0,
// а при редагуванні даних flag буде мати значення 1;
// n визначатиме номер запису у файлі, причому номер рахуватиметься за кількістю
// рядків, відбитих у компоненті StringGrid, хоча у файлі відлік розпочинається з 0
FILE *f;
char s[]="товар.dat"; // Ім'я файла
struct tovar
 { char N[20];    float price;   int kol; };
tovar t;
```

```
// Кнопка "Дописати до файла", яка може змінювати свій надпис
// на "Зберегти зміни".
void __fastcall TForm1::Button1Click(TObject *Sender)
{ if(flag==0)                        // У режимі "Дописати до файла"
  {if((f=fopen(s,"ab+"))==NULL) // файл відкривається для дописування
    { ShowMessage("Файла немає!"); return;} // даних у кінець файла
   strcpy(t.N,Edit1->Text.c_str( ));
   t.price=StrToFloat(Edit2->Text);
   t.kol=StrToInt(Edit3->Text);
   fwrite(&t,sizeof(tovar),1,f); // Записування нових даних
   fclose(f);
  }
  else                               // У режимі "Зберегти зміни"
  {if((f=fopen(s,"rb+"))==NULL) // після відкриття позицію розташовано
    { ShowMessage("Файла немає!"); return;} // на початку файла
   strcpy(t.N,Edit1->Text.c_str( ));
   t.price=StrToFloat(Edit2->Text);
   t.kol=StrToInt(Edit3->Text);
   fseek(f,(n-1)*sizeof(tovar),0);// Переміщення на запис з номером (n-1)
   fwrite(&t,sizeof(tovar),1,f);
   Button1->Caption="Дописати до файла";
   fseek(f,0,0);                    // Переміщення на початок файла
   int i=2;
   while(fread(&t,sizeof(tovar),1,f))
    { SG1->RowCount=i;
      SG1->Cells[0][i-1]=AnsiString(t.N);
      SG1->Cells[1][i-1]=FormatFloat("0.00",t.price);
      SG1->Cells[2][i-1]=IntToStr(t.kol);
      i++;
    }
   fclose(f);
   flag=0;                      // Вимкнення режиму "Зберегти зміни"
  }
  Edit1->Clear();  Edit2->Clear();  Edit3->Clear();
  Edit1->SetFocus();
}
// Кнопка "Переглянути файл"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ if((f=fopen(s,"rb"))==NULL)
   { ShowMessage("Файла немає!");  return; }
  SG1->Cells[0][0]="Найменування";
  SG1->Cells[1][0]="Кільк.";
  SG1->Cells[2][0]="Ціна";
  int i=2;
  while(fread(&t,sizeof(tovar),1,f))
  { SG1->RowCount=i;
    SG1->Cells[0][i-1]=AnsiString(t.N);
    SG1->Cells[1][i-1]=FormatFloat("0.00",t.price);
    SG1->Cells[2][i-1]=IntToStr(t.kol);
```

```
    i++;
  }
  fclose(f);
}
// Кнопка "Товари, кількість яких є менше за 30"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ if((f=fopen(s,"rb+"))==NULL)
   { ShowMessage("Файла немає!");  return; }
  Memo1->Lines->Clear( );
  while(fread(&t,sizeof(tovar),1,f))
    if(t.kol<30) Memo1->Lines->Add(AnsiString(t.N)+" "+
        FormatFloat("0.00",t.price)+" "+IntToStr(t.kol));
  fclose(f);
}
// Кнопка "Видалити запис"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ if ((f=fopen(s,"rb"))==NULL)
   { ShowMessage("Файла немає!"); return; }
  FILE* tmp;    // Оголошення вказівника на тимчасовий файл
  if((tmp=fopen("tmp.dat","wb+"))==NULL) //і створення його.
   { ShowMessage("Файла немає!"); return; }
  AnsiString snom=InputBox("Який рядок бажаєте вилучити?",
              "Введіть номер рядка", "1");
  if(snom=="") { ShowMessage("Ви не ввели дані"); return; }
  int kk, k=StrToInt(snom);
  while((fread(&t,sizeof(tovar),1,f)))
   { kk=ftell(f)/sizeof(tovar); // Функція ftell(f) визначає кількість
     // прочитаних байтів, а в змінній kk обчислюється поточний номер запису
     if(k==kk)  continue; // Пропуск зазначеного запису
     fwrite(&t,sizeof(tovar),1,tmp);
   }
  fclose(f);
  fseek(tmp,0,0);     // Перехід на початок тимчасового файла
  f=fopen(s, "wb+"); // і нове створення файла для зчитування і записування
  while((fread(&t,sizeof(tovar),1,tmp)))
   fwrite(&t,sizeof(tovar),1,f);
  fclose(tmp);   fclose(f);
  remove("tmp.dat");     // Видалення допоміжного файла
  Button2Click(NULL);    // Переглядання даних після видалення
}
// Кнопка " Редагувати запис"
void __fastcall TForm1::Button5Click(TObject *Sender)
{ if((f=fopen(s,"rb+"))==NULL)
   { ShowMessage("Файла немає!");  return; }
  AnsiString snom=InputBox("Який рядок бажаєте коригувати?",
    "Введіть номер рядка.\nНові значення вводитимуться \n
     з відповідних компонентів Edit", "");
  if(snom=="") {ShowMessage("Ви не ввели дані"); return;}
  n=StrToInt(snom);
```

```
    fseek(f,(n-1)*sizeof(tovar),0);    // Перехід на (n-1)-й запис,
    fread(&t,sizeof(tovar),1,f);       // зчитування цього запису
    Edit1->Text=AnsiString(t.N);       // і виведення його
    Edit2->Text=FormatFloat("0.00",t.price);//до відповідних
    Edit3->Text=IntToStr(t.kol);       // компонентів для редагування.
    fclose(f);
    flag=1;                  // Ввімкнення режиму "Зберегти зміни"
    Button1->Caption="Зберегти зміни";// Змінення надпису на кнопці
    Button2Click(NULL);     // Переглядання даних після редагування
}
```

12.3.2 Робота з бінарними файловими потоками у стилі C++

Для роботи з бінарним форматом файла використовується прапорець `ios::binary`. Ці файли відкриваються здебільшого в режимі зчитування-записування. Для цього зручними є об'єкти класу `fstream`. Дані до бінарних файлів записуються за допомогою методу `write()` класу `ofstream`, а зчитуються – за допомогою методу `read()` класу `ifstream`.

Наприклад:

```
fout.write((char)*&A, sizeof(A));
fin.read((char)*&A, sizeof(A));
```

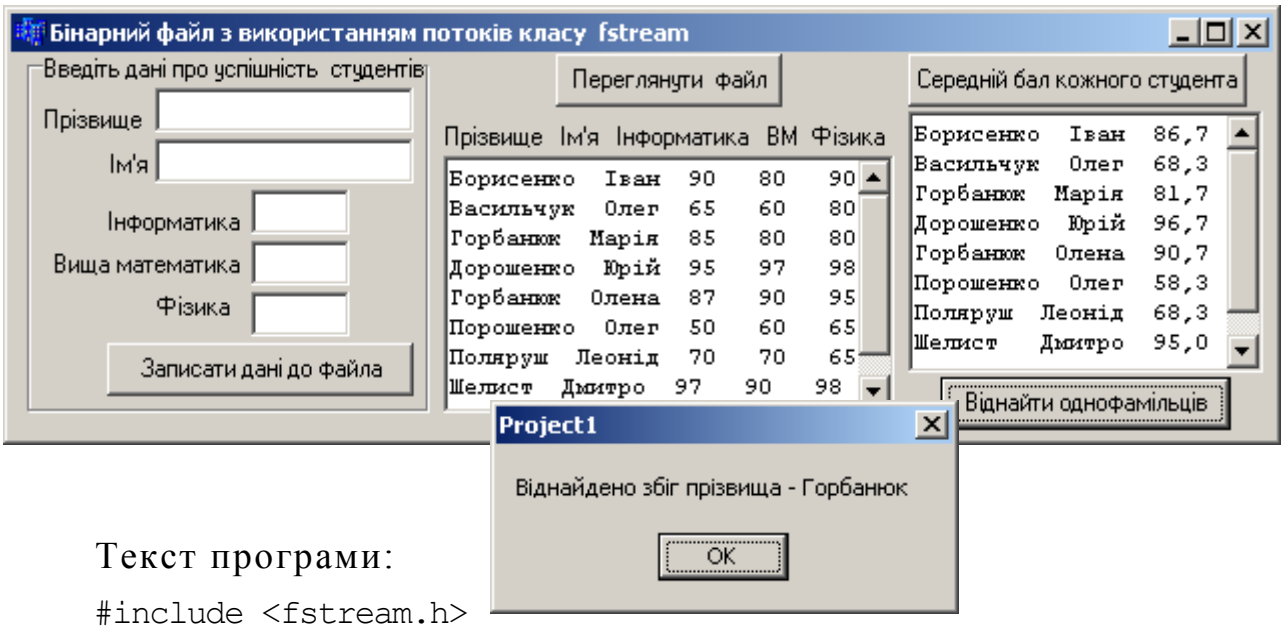
Перший параметр у цих функцій – адреса змінної `A`. Ці функції працюють з байтами (типу `char`). Для них немає жодного значення, в який спосіб організовано й що собою являють дані, – вони просто переносять їх побайтово з буфера до файла та навпаки. Другий параметр – довжина змінної `A` у байтах. Тому перший оператор обчислює значення виразу `sizeof(A)` й копіює до файла, пов'язаного з об'єктом `fout`, обчислену кількість байтів, розпочинаючи з вказаної адреси. Другий оператор копіює кількість байтів `sizeof(A)` з файла `fin` до змінної `A`.

У бінарному файлі також існує можливість довільного доступу до файла за допомогою методів `seekp()` та `seekg()`.

При повторному відкритті файла з використанням тієї ж самої файлової змінної потік слід очищувати за допомогою методу `clear()`, наприклад: `f.clear()`.

Приклад 12.15 Створити файл з інформацією про успішність студентів: прізвище, ім'я, оцінки з інформатики, математики та фізики. Вивести на екран середній бал кожного студента. Виявити, чи є серед студентів збіг прізвищ (однофамільці).

Вікно форми проекту з результатами роботи програми:



Текст програми:

```
#include <fstream.h>
fstream f;
char *filename=new char[30];
struct student
 { char name[30], surname[20];
   int inf, mat, fiz;  };
student A;
void __fastcall TForm1::FormCreate(TObject *Sender)
{ AnsiString fn=InputBox("Відкриття файла",
                  "Введіть ім'я файла","stud.dat");
  if(fn=="") { ShowMessage("Ви не ввели дані"); return; }
  strcpy(filename,fn.c_str());
  // Створення чи відкриття файла
  f.open(filename,ios::out|ios::app|ios::binary);
  f.clear();   // Очищення потоку
  if(!f) { ShowMessage("Помилка відкриття файла"); return; }
  f.close();
}
// Кнопка "Записати дані до файла"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ f.open(filename,ios::out|ios::app|ios::binary);
  f.clear();
  if(!f) { ShowMessage("Помилка відкриття файла "); return; }
  f.seekp(0,ios::end); // Позиціонування на кінець файла
  strcpy(A.name, Edit1->Text.c_str());
  strcpy(A.surname, Edit2->Text.c_str());
  A.inf=StrToInt(Edit3->Text);
  A.mat=StrToInt(Edit4->Text);
  A.fiz=StrToInt(Edit5->Text);
  f.write((char*)&A, sizeof(A));   // Записування даних до файла
  f.close();
}
```

```
// Кнопка "Переглянути файл"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Memo1->Clear();
  f.open(filename,ios::out|ios::in|ios::binary);
  f.clear();
  if(!f)
   { ShowMessage("Помилка відкриття файла "); return; }
  while(f.read((char*)&A, sizeof(A)))      // Зчитування з файла
   Memo1->Lines->Add(AnsiString(A.name)+"  "+
      AnsiString(A.surname)+"  "+   IntToStr(A.inf)+"   "+
      IntToStr(A.mat)+"   "+IntToStr(A.fiz));
  f.close();
}
// Кнопка "Середній бал кожного студента"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ f.open(filename,ios::out|ios::in|ios::binary);
  f.clear();
  Memo2->Clear();
  if(!f) { ShowMessage("Помилка відкриття файла"); return; }
  double sr;
  while(f.read((char*)&A, sizeof(A)))
  { sr=(A.inf + A.mat + A.fiz)/3.;     // Обчислення середнього балу
    Memo2->Lines->Add(AnsiString(A.name) + "  " +
      AnsiString(A.surname)+"  "+FormatFloat("0.0",sr));
  }
  f.close();
}
// Кнопка "Віднайти однофамільців"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ f.open(filename,ios::out|ios::in|ios::binary);
  f.clear();
  if(!f) { ShowMessage("Помилка відкриття файла"); return; }
  student A1;  long int k=0;
  while(f.read((char*)&A, sizeof(A)))
  { while(f.read((char*)&A1, sizeof(A1)))
     if(strcmp(A.name,A1.name)==0)
      { ShowMessage("Віднайдено збіг прізвища - " +
                    AnsiString(A.name));
        return;
      }
    f.clear();
    k++;
    // Позиціонування на структуру файла з номером k
    f.seekg(k*sizeof(A),ios::beg);
  }
  ShowMessage("Збігу прізвищ немає");
  f.close();
}
```

12.3.3 Опрацювання бінарних файлів за допомогою дескрипторів

Для опрацювання бінарних файлів через дескриптори застосовуються функції, які розглянуто в п. 12.2.6.

Нагадаємо порядок дій для записування даних у файл на такому прикладі:

```
int f;                          // Оголошення дескриптора
f=FileOpen("a.dat",fmOpenWrite);// Відкриття файла для записування
int kb=FileSeek(f, 0, 2);       // Встановлення позиції на кінець файла
                                // і одночасне визначення розміру файла
int k=25;
// Записування до файла значення k цілого типу розміром sizeof(int)
FileWrite(f, &k, sizeof(int));
FileClose(f);                   // Закриття файла
```

Відкриття файла і зчитування даних з нього виконуватиметься в такий спосіб:

```
f=FileOpen("a.dat", fmOpenRead);
FileRead(f, (int*)&k, sizeof(int));
FileClose(f);
```

Відкриття файла для зчитування-записування здійснюватиметься наступною послідовністю команд:

```
f=FileOpen("a.dat",fmOpenReadWrite);
if(f != -1)
/* Далі можна опрацьовувати файл: читати чи записувати данні, встановивши
   потрібну позицію за допомогою функції FileSeek(). */
```

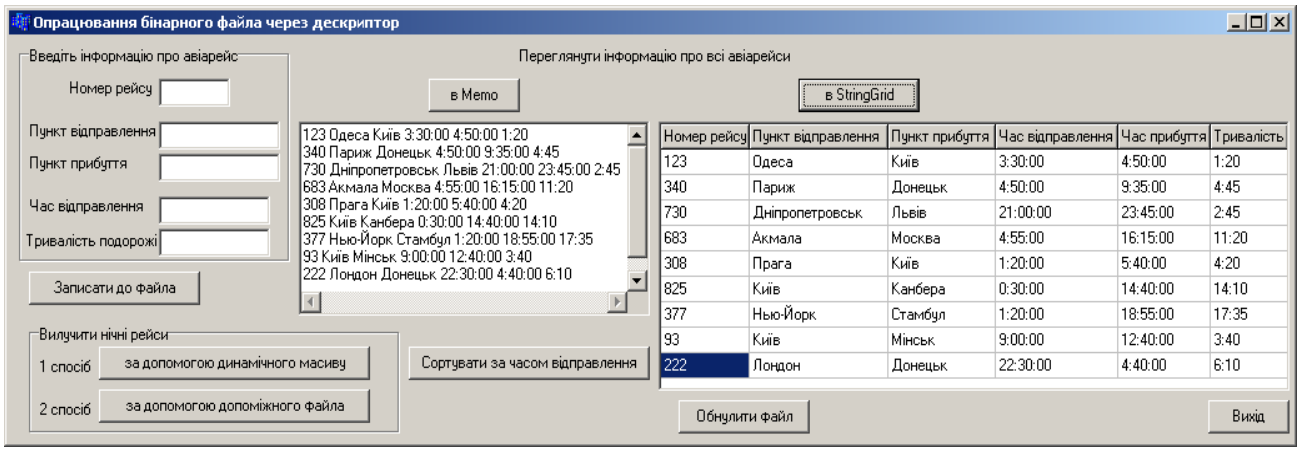
Ще раз зауважимо, що за допомогою дескрипторів можна створювати і опрацьовувати файли з даними будь-якого типу.

Розглянемо приклад роботи з бінарним файлом через дескриптор.

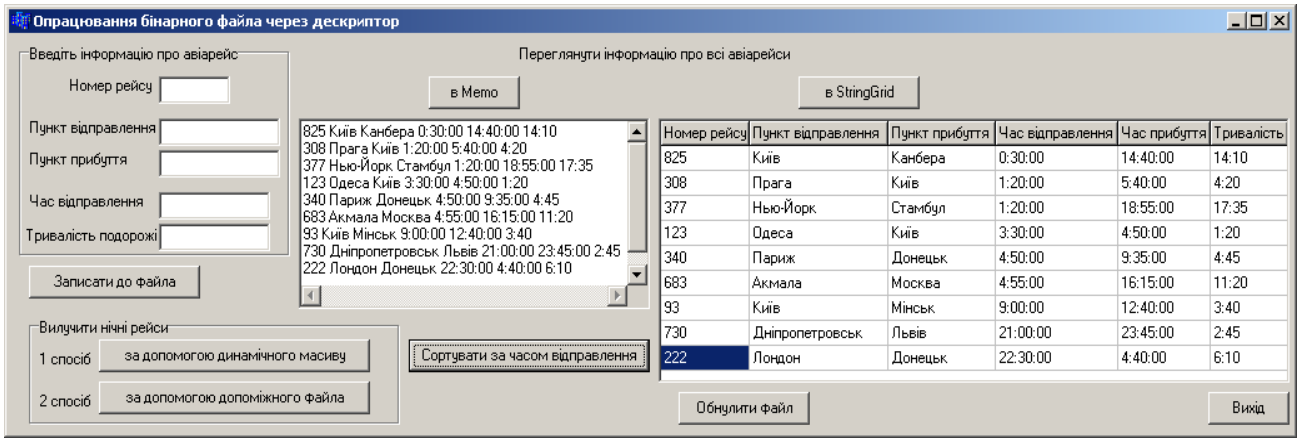
Приклад 12.16 Створити файл з інформацією про авіарейси: номер рейсу, маршрут руху, час відправлення (за шаблоном ЧЧ:ММ), час прибуття в пункт призначення (за шаблоном ЧЧ:ММ), при чому з клавіатури мають вводитися: номер рейсу, маршрут руху, час відправлення і тривалість польоту, а час прибуття в пункт призначення слід обчислити. Передбачити можливість вилучення даних про нічні рейси (з 00:00 по 5:00). Відсортувати данні за часом відправлення літаків.

Розв'язок. У програмі за вибором користувача передбачено можливість виведення даних у компонент `Memo1` чи то у компонент `StringGrid1`, який у програмі перейменовано для зручності на `SG1`. Тут спеціально наведено засоби виведення даних в обидва компоненти, хоча вочевидь, що в цьому прикладі зовні краще сприймається інформація у `StringGrid`.

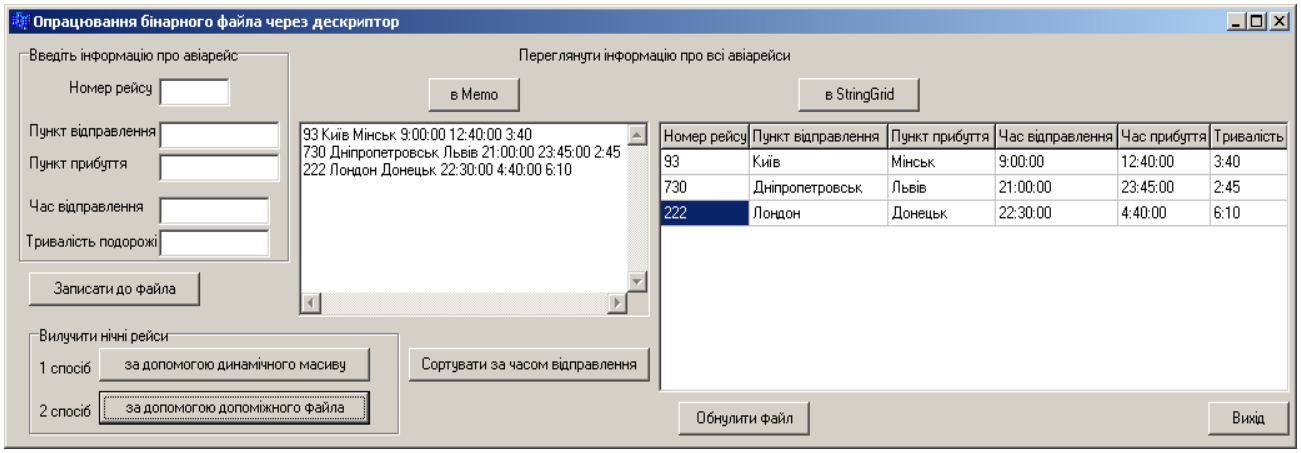
При перегляданні даних обома способами вікно форми матиме вигляд



Форма після сортування авіарейсів за часом відправлення літаків:



Для розв’язування завдання вилучення нічних рейсів запропоновано два способи. У першому – організовано допоміжний динамічний масив, до якого обираються відомості про авіарейси, за винятком нічних. Початковий файл вилучається і створюється заново, після чого до нього записуються відомості з динамічного масиву, а пам’ять від динамічного масиву звільнюється. Порівняно з першим способом, другий має більш компактний програмний код. У ньому організовано допоміжний файл, в який обираються авіарейси, окрім нічних. Після цього початковий файл вилучається, а допоміжний перейменовується на ім’я початкового файла. Результати обох способів є аналогічні і на формі вони мають такий вигляд:



Окрім зазначених алгоритмів опрацювання бінарного файла через дескриптор, у програмі наведено засоби вилучення всіх даних файла, тобто обнуління файла. Перед вилученням даних у програмі передбачено виведення діалогового повідомлення `MessageDlg()`. На відміну від повідомлення `ShowMessage()`, яке виводить лише вікно з відповідним текстом повідомлення та однією командною кнопкою з надписом `OK`, при натисканні на яку вікно повідомлення зникає, діалогове повідомлення `MessageDlg()` може моделювати кількість кнопок із зазначеними надписами і передбачає можливість програмного реагування залежно від того, яку саме кнопку було натиснуто.

Синтаксис функції `MessageDlg()` є такий:

```
int MessageDlg( const AnsiString Msg, TMsgDlgType DlgType,
                TMsgDlgButtons Buttons, int HelpCtx);
```

де `Msg` – текст повідомлення;

`DlgType` задає характер повідомлення, а саме вигляд піктограми ліворуч від тексту повідомлення. Існують такі можливі значення цього параметра: `mtWarning` – зауваження  , `mtError` – помилка  , `mtInformation` – інформаційне повідомлення  , `mtConfirmation` – запит підтвердження  , `mtCustom` – без рисунка;

`Buttons` задає кількість і вигляд кнопок. Можливі значення цього параметра: `mbYesNoCancel`, `mbYesNoAllCancel`, `mbOKCancel`, `mbAbortRetryIgnore`, `mbAbortIgnore`. Тобто кожна кнопка має відповідне константне значення. Після натискання певної кнопки відповідне константне значення повертається функцією як результат виконання;

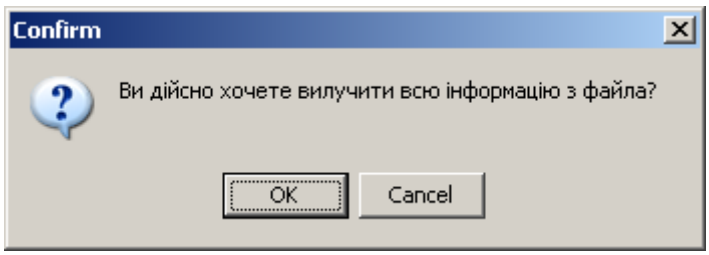
`HelpCtx` визначає екран контекстної довідки, що буде з'являтися за натискання клавіші <F1>. Якщо така довідка не планується, слід задати значення 0.

Значення параметра `Buttons`	Відповідне значення, яке повертається
`mbOK`	`mrOk`
`mbCancel`	`mrCancel`
`mbYes`	`mrYes`
`mbNo`	`mrNo`
`mbAbort`	`mrAbort`
`mbRetry`	`mrRetry`
`mbIgnore`	`mrIgnore`
`mbAll`	`mrAll`
`mbNoToAll`	`mrNoToAll`
`mbYesToAll`	`mrYesToAll`

Діалогове повідомлення, створене у програмі командою

```
int btn = MessageDlg("Ви дійсно хочете вилучити всю
      інформацію з файла?", mtConfirmation, mbOKCancel, 0);
```

матиме вигляд:



Текст програми

```
// Глобальні оголошення
struct avio_struct
{ char  num[5], from[20], to[20], all_time[6];
  TDateTime start_time, last_time;
  AnsiString info( )
  { return AnsiString(num) + " " + (AnsiString)from + " " +
    (AnsiString)to + " " + start_time.TimeString( ) + " " +
     last_time.TimeString( )+" "+(AnsiString)all_time;
}  };
int size = sizeof(avio_struct);    // Розмір структури
avio_struct w;                     // Змінна типу структури
int f;                             // Дескриптор
char s[]="a.dat";                  // Ім'я файла
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)             // Конструктор форми
{ if (!FileExists(s))              // Якщо такого файла не існувало,
    f=FileCreate(s);               // він створюється
  FileClose(f);
  SG1->Cells[0][0]="Номер рейсу";
  SG1->Cells[1][0]="Пункт відправлення";
  SG1->Cells[2][0]="Пункт прибуття";
  SG1->Cells[3][0]="Час відправлення";
  SG1->Cells[4][0]="Час прибуття";
  SG1->Cells[5][0]="Тривалість";
}
// Кнопка "Записати до файла"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString s1,s2;
  strcpy(w.num,Edit1->Text.c_str( ));
  strcpy(w.from,Edit2->Text.c_str( ));
  strcpy(w.to,Edit3->Text.c_str( ));
  w.start_time=Edit4->Text;
  strcpy(w.all_time,Edit5->Text.c_str( ));
  s1=(AnsiString)w.all_time;
  int n=StrToInt(s1.SubString(1,s1.Pos(':')-1));
  if (n>=24) n=n%24;
  // Створення шаблону  ЧЧ:ММ.
  s2=IntToStr(n)+":"+s1.SubString(s1.Pos(':')+1,2);
  w.last_time=w.start_time+s2;
  // Відкриття файла для зчитування-записування
  f=FileOpen(s,fmOpenReadWrite);
```

```
  if(f != -1)                 // Перевірка коректності відкриття файла.
  { FileSeek(f,0,2);          // Позиціонування на кінець файла
    FileWrite(f,&w,size);     // і записування даних
    FileClose(f);
  }
  else          //Якщо сталася помилка при відкритті, виведеться відповідне
  { ShowMessage("Помилка доступу до файла"); // повідомлення
    return;  }
  Edit1->Clear( ); Edit2->Clear( ); Edit3->Clear( );
  Edit4->Clear( ); Edit5->Clear( ); Edit1->SetFocus( );
}
// Переглядання інформації про всі авіарейси в Memo
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Memo1->Clear();
  f=FileOpen(s,fmOpenRead);
  while(FileRead(f,&w,size)) Memo1->Lines->Add(w.info( ));
  FileClose(f);
}
// Переглядання інформації про всі авіарейси в StringGrid
void __fastcall TForm1::Button3Click(TObject *Sender)
{ SG1->RowCount=2;
  AnsiString st;
  bool d=true;
  f=FileOpen(s,fmOpenRead);
  if (f == -1)
   { ShowMessage("Файл не відкрито");
     return;
   }
  while (FileRead(f,&w,size))
   { st=w.info( );
     if (d)
      { SG1->Rows[SG1->Row]->DelimitedText=st;
        d=false;
      }
     else
      { SG1->RowCount++;
        SG1->Row=SG1->RowCount-1;
        SG1->Rows[SG1->Row]->DelimitedText=st;
  }   }
  FileClose(f);
}
// Кнопка "Сортувати за часом відправлення"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ f=FileOpen(s,fmOpenReadWrite);
  int fkol=FileSeek(f,0,2); // Визначення розміру файла в байтах
  int kol=fkol/size; // Обчислення кількості записів у файлі
  avio_struct temp;  // Допоміжна змінна типу структури для сортування
  // Виділення динамічної пам'яті під масив структур
  avio_struct *av=new avio_struct[kol];
```

```
  FileSeek(f,0,0);   // Позиціонування на початок файла
  while (FileRead(f,&w,size))
   *(av+i) = w;      // Збереження запису як елемента масиву
  FileClose(f);
  for(int i=0;i<kol-1;i++)
  for(int j=i+1;j<kol;j++)
   if(av[j].start_time<av[i].start_time) // Сортування даних
    { temp=av[i];
      av[i]=av[j];
      av[j]=temp;   }
  f=FileOpen(s,fmOpenReadWrite);//Відкриття файла для записування
  for (int i=0;i<kol;i++)
   { FileWrite(f,&av[i],size);   //Записування даних до обнуленого файла
     delete av;                  //Звільнення динамічного масиву
   }
  FileClose(f);
  Button2Click(NULL);    // Переглядання даних після видалення в Memo
  Button3Click(NULL);    // і в StringGrid
}
// Вилучення нічних рейсів за допомогою динамічного масиву.
void __fastcall TForm1::Button5Click(TObject *Sender)
{ TDateTime d1=TTime("00:00"), d2=TTime("05:00");
  Memo1->Clear( );
  f=FileOpen(s,fmOpenReadWrite);
  int fkol=FileSeek(f,0,2);     // Розмір файла в байтах
  int kol=fkol/size;            // Кількість записів у файлі
  // Виділення динамічної пам'яті під масив структур відповідного розміру
  avio_struct *av = new avio_struct[kol];
  int kolzap=0;
  FileSeek(f,0,0);
  while(FileRead(f,&w,size))
  { if(w.start_time>=d1 && w.start_time<=d2)
      continue;
    *(av+kolzap) = w;           // Збереження запису в елементі масиву
    kolzap++;
  }
  FileClose(f); DeleteFile(s); // Закриття і видалення файла
  f=FileCreate(s);             // і створення заново
  FileClose(f);
  f=FileOpen(s,fmOpenWrite);   // Відкриття файла для записування
  for(int i=0;i<kolzap;i++)
    FileWrite(f,&av[i],size); // Записування даних до файла
  FileClose(f);
  for(int i=0;i<kolzap;i++)
    delete av;           // Звільнення динамічної пам'яті від масиву.
  Button2Click(NULL);    // Переглядання даних після видалення в Memo
  Button3Click(NULL);    // і в StringGrid.
}
```

```
// Вилучення нічних рейсів за допомогою допоміжного файла
void __fastcall TForm1::Button6Click(TObject *Sender)
{ int tmp=FileCreate("tmp.dat"); // Створення тимчасового файла
  FileClose(tmp);
  f=FileOpen(s,fmOpenRead);//Відкриття початкового файла для зчитування,
  tmp=FileOpen("tmp.dat",fmOpenWrite);//а тимчасового – для записування
  FileSeek(f,0,0);
  while (FileRead(f,&w,size))
  { if(!(w.start_time>=TTime("00:00") &&
         w.start_time<=TTime("05:00")))
     FileWrite(tmp,&w,size);  // Записування даних до тимчасового файла
  }
  FileClose(f);  FileClose(tmp);// Закриття обох файлів,
  DeleteFile(s);                // видалення початкового файла
  RenameFile("tmp.dat", s);     // і перейменування тимчасового файла.
  Button2Click(NULL);     // Переглядання даних після видалення в Memo
  Button3Click(NULL);     // і в StringGrid
}
// Кнопка "Обнулити файл"
void __fastcall TForm1::Button7Click(TObject *Sender)
{ int btn=MessageDlg("Ви дійсно хочете видалити усю інформацію
                     з файла?", mtConfirmation, mbOKCancel, 0);
  if(btn==1)
  { f=FileCreate(s);      // Обнулення файла
    FileClose(f);         // і закриття його
    ShowMessage("Файл пустий!");
} }
// Кнопка "Вихід"
void __fastcall TForm1::Button8Click(TObject *Sender)
{ Close(); }
```

12.4 Робота з графічними файлами у C++ Builder

C++ Builder надає зручні та прості у застосуванні засоби для роботи з графічними файлами растрових зображень (бітових образів у форматі *.bmp, піктограм *.ico чи метафайлів *.wmf (*.emf)). Для цього визначено похідні від базового класу `TGraphic`, який надає мінімальний стандартний інтерфейс для роботи з графікою, об'єктні класи `TBitmap`, `TIcon` та `TMetafile`. З цих класів клас `TBitmap` є єдиним графічним класом з канвою.

Канва (властивість Canvas) графічних компонентів дозволяє програмістові при роботі з графікою не вникати в проблеми стеження за системними ресурсами, а лише визначити властивості та характеристики графічних інструментів для використовуваних компонентів.

Зазвичай робота з зображенням провадиться лише з частиною, яка відкрита для доступу через контейнерний клас `TPicture`. Він є інкапсульований у клас `TImage` і візуальний компонент `TImage`, як їхній основний компонент та має можливість роботи з іконками, `bmp`-файлами і метафайлами. А при долу-

ченні заголовного файла jpeg.hpp можна опрацьовувати файли форматів *.jpeg, *.jpg.

Методи LoadFromFile() та SaveToFile()дозволяють динамічно завантажувати та зберігати файли зображень форматів bmp та ico до компонентів типу TImage:

```
Image1->Picture->LoadFromFile("1.bmp");
Image1->Picture->SaveToFile("2.bmp");
```

або

```
Image1->Picture->LoadFromFile("1.ico");
Image1->Picture->SaveToFile("2.ico");
```

Для того, щоби звертатися до метафайлів формату emf (оновлена версія формату метафайлів фірми Microsoft), слід скористатися властивістю Enhanced, встановивши її в значення true. Значення false властивості Enhanced використовується для більш старої версії метафайлів wmf.

```
Image1->Picture->Metafile->Enhanced=true;
Image1->Picture->LoadFromFile("1.emf");
Image1->Picture->SaveToFile("2.emf");
```

Що стосується візуального компонента TImage, то він призначений для розміщення на формі графічного зображення (бітового образу, піктограми чи метафайла). Більшість властивостей цього компонента є такі самі, як і багатьох інших компонентів (розташування зображення на формі, його розміри тощо). Зосередимо увагу на "нестандартних" властивостях цього компонента:

Picture – контейнер для завантаження файлів зображень чи то при створенні форми чи програмно за допомогою методів LoadFromFile() та SaveToFile();

AutoSize – значення true цієї властивості автоматично змінює (збільшує чи зменшує) розміри контейнера до розмірів зображення;

Stretch – значення true цієї властивості вписує (розтягує чи стискає) зображення у рамки компонента, але оскільки навряд чи реально встановити розміри Image точно пропорційними розміру малюнка, то зображення спотвориться. Тому встановлювати Stretch у true доцільно лише для візерунків, але не для картинок. Властивість Stretch не діє на зображення піктограм ico, які не можуть змінювати свої розміри;

Center – вказує, чи потрібно центрувати зображення у межах компонента. Ігнорується, якщо Autosize = true або якщо Stretch = true і зображення не є піктограмою (ico);

IncrementalDisplay – якщо true, то при завантаженні великих файлів вони будуть відбиватися частинами по мірі завантаження;

Transparent – прозорість фону, яку можна використовувати для накладення зображень одне на одне, щоби верхня картинка не затуляла нижню. Одне з можливих застосувань цієї властивості – накладення на картинку написів, виконаних у вигляді бітової матриці. Кольором прозорості береться колір лівого нижнього кутового пікселя.

Canvas – програмно доступна властивість, яка містить канву для виведення растрового зображення (але не піктограми чи метафайла!).

Для доступу до графічних файлів у C++ Builder існують спеціальні компоненти на закладці Dialogs:

`OpenPictureDialog` – відкрити рисунок. Реалізує спеціальне вікно вибору графічних файлів з можливістю попереднього переглядання рисунків;

`SavePictureDialog` – зберегти рисунок. Реалізує спеціальне вікно зберігання графічних файлів з можливістю попереднього переглядання рисунків.

Усі властивості цих компонентів однакові, тільки їх сенс дещо різний для відкриття і закриття файлів:

`FileName` – основна властивість, в якій повертається у вигляді рядка обраний користувачем файл. Значення цієї властивості можна задати і перед зверненням до діалогу;

`Filter` – фільтр, який задає один, декілька чи усі доступні типи графічних файлів зі списку: *.jpg; *.jpeg; *.bmp; *.ico; *.emf; *.wmf. Один від одного шаблони відокремлюються вертикальними рисками;

`FilterIndex` – визначає номер фільтра, який буде за замовчуванням показаний користувачеві у момент відкриття діалогу. Наприклад, значення `FilterIndex = 1` задає за замовчуванням перший фільтр;

`InitialDir` – визначає початковий каталог, який відкриється у момент початку роботи користувача з діалогом. Якщо значення цієї властивості не задане, то відкриється поточний каталог чи той, який був відкритий при останньому зверненні користувача до відповідного діалогу;

`DefaultExt` – визначає значення розширення файла за замовчуванням. Якщо значення цієї властивості не задане, користувач повинен вказати у діалозі повне ім'я файла з розширенням. Якщо ж задати значення `DefaultExt`, то користувач може писати у діалозі ім'я без розширення. У цьому разі буде прийнято задане розширення;

`Title` – дозволяє задавати заголовок діалогового вікна. Якщо цю властивість не задано, вікно відкривається із заголовком, визначеним операційною системою;

`Options` – визначає умови вибору файла. Множина опцій, які можна встановлювати програмно чи під час проектування, основні з яких:

- ✓ `ofAllowMultiSelect` – дозволяє користувачеві вибирати декілька файлів;
- ✓ `ofCreatePrompt` – у разі, якщо користувач написав ім'я неіснуючого файла, з'явиться зауваження і запит, чи треба створити файл із заданим ім'ям;
- ✓ `ofEnableIncludeNotify` – дозволяє посилати в діалог повідомлення;
- ✓ `ofEnableSizing` – дозволяє користувачеві змінювати розмір діалогового вікна;
- ✓ `ofFileMustExist` – у разі, якщо користувач написав ім'я неіснуючого файла, з'явиться повідомлення про помилку;
- ✓ `ofNoChangeDir` – після клацання користувача на кнопці "OK" відновлює поточний каталог, незалежно від того, який каталог був відкритий при пошуку файла;
- ✓ `ofPathMustExist` – генерує повідомлення про помилку, якщо користувач вказав в імені файла неіснуючий каталог;

✓ `ofOverwritePrompt` – у разі, якщо при збереженні файла користувач написав ім'я існуючого файла, з'явиться зауваження, що файл з таким ім'ям існує, і запит до користувача переписати існуючий файл;

✓ `ofHideReadOnly` – видаляє з діалогу індикатор "Відкрити тільки для читання";

✓ `ofReadOnly` – за замовчуванням встановлює індикатор "Відкрити тільки для читання" при відкритті діалогу.

За замовчуванням усі опції, окрім `ofHideReadOnly`, є вимкнені. Проте, як видно з їхнього опису, більшість з них доцільно встановлювати перед викликом діалогів.

Якщо ви дозволяєте за допомогою опції `ofAllowMultiSelect` множинний вибір файлів, то список обраних файлів можна прочитати у властивості `Files` типу `TStrings`.

Проілюструємо на прикладі роботу з компонентами `Image`, `OpenPictureDialog`, `SavePictureDialog` та методами `LoadFromFile()` та `SaveToFile()`.

Приклад 12.17 Написати програму-фотоальбом, яка дозволяє переглядати файлів зображень, наприклад фотографій, і редагувати (долучати, вилучати, змінювати) вміст альбому.

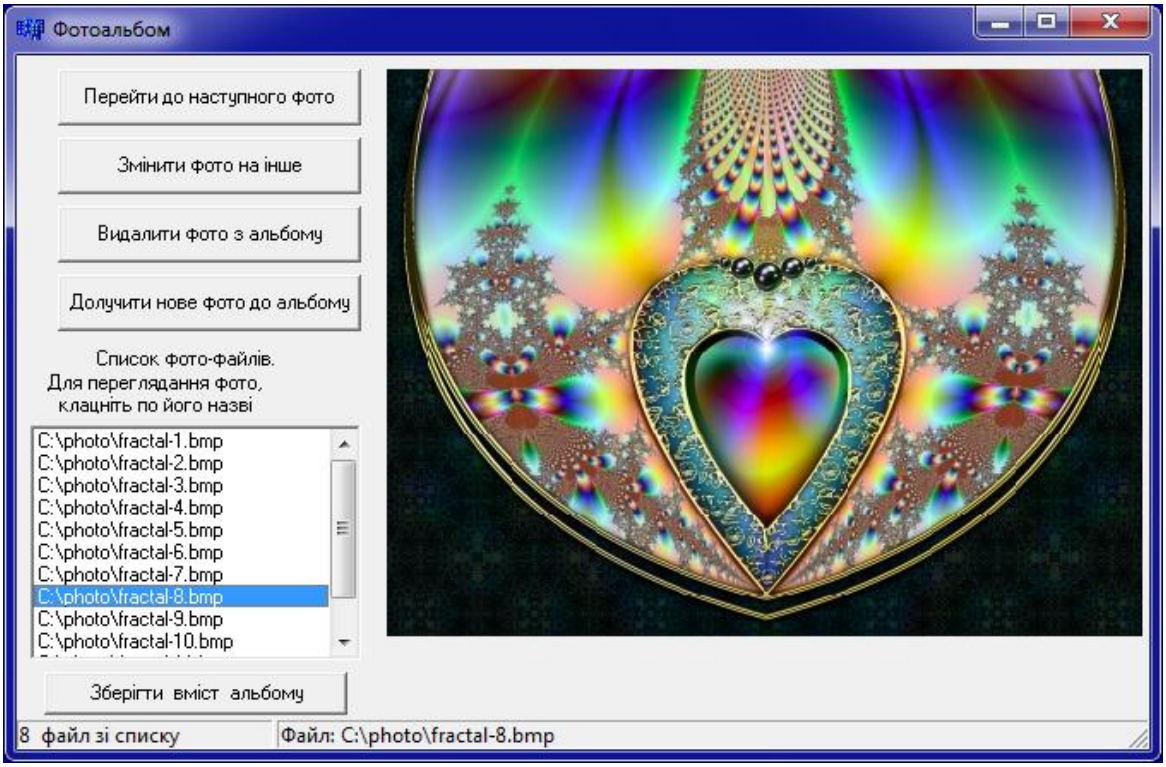
Розв'язок. На формі розмітимо компонент `ListBox1` для переглядання списку фотографій альбому та компонент `Image1` для посереднього переглядання самих фото. Щоби невеликі фотографії виводились посередині компонента `Image1` встановимо властивість `Center` у значення `true`. Напроти для виведення усього, а не лише фрагмента, зображення великих фото встановимо властивість `AutoSize` у значення `true`. Крім того, використаємо спеціалізований діалог для відкриття графічних файлів `OpenPictureDialog` (закладка Dialogs) та компонент `StatusBar` (закладка Win32) для виведення панелі у нижній частині форми з інформацією про ім'я та місцезнаходження файла, який переглядається.

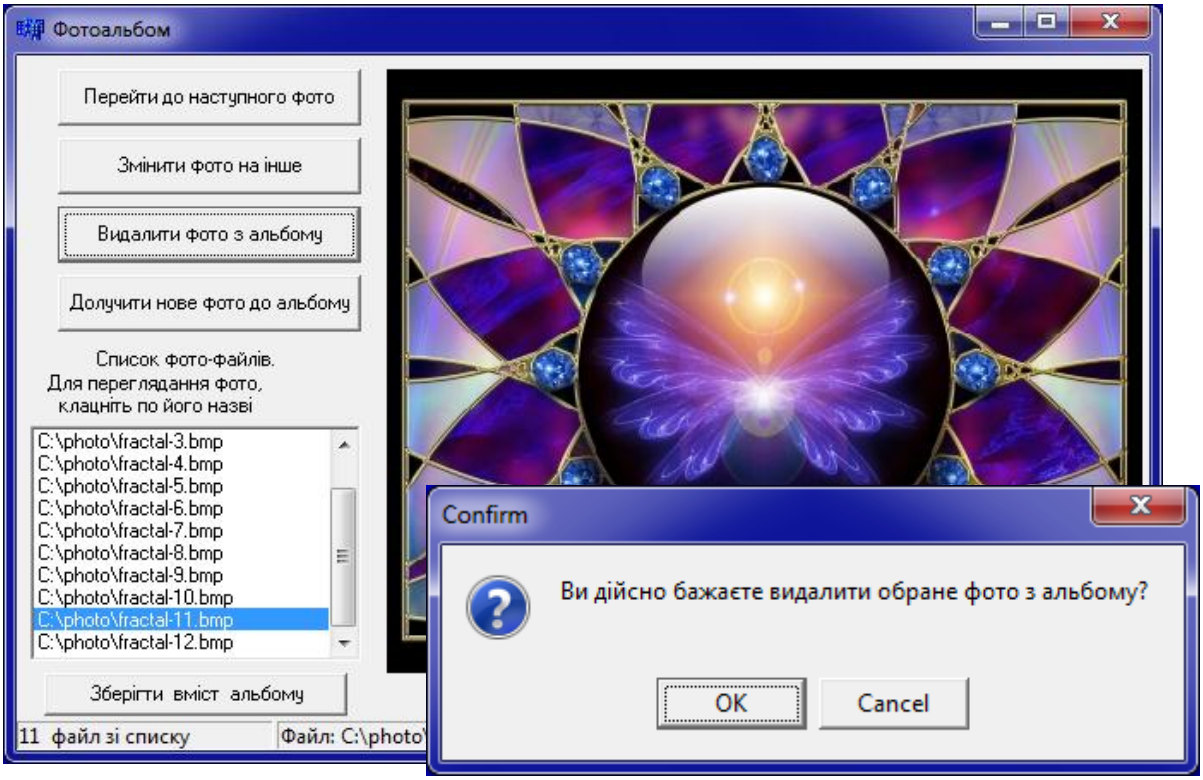
Взагалі кажучи, компонент `StatusBar` доцільно використовувати для виведення різноманітної службової інформації, яка дозволяє вести візуальний контроль за діями програмного додатку. Щоби створити багатосекційну панель (у нашому прикладі – двосекційну) на етапі створення форми, слід двічі клацнути по компонентові на формі та у віконці редагування панелей, яке відкриється, скористатись іконкою  "Add new Ins" для створення бажаної кількості секцій панелі. Нумерація індексів секцій розпочинається з нуля, а розмір ширини кожної секції можна змінити зі значення у 50 пікселів за замовчуванням, приміром на 150, у вікні інспектора об'єктів (Object Inspector), попередньо виокремивши мишкою певну секцію. Аналогічні за дією команди можна прописати у програмі, як приміром у наведеному прикладі для функції **`FormCreate`**.

Крім того зауважимо, що за замовчуванням компоненти `Image` та `OpenPictureDialog` забезпечують доступ лише до файлів зображень у форматах BMP (бітовий образ), ICO (піктограму), WMF чи EMF (метафайли). Щоби забезпечити доступ і до графічних файлів у форматах JPEG та JPG, залучимо до програми бібліотеку `jpeg.hpp`.

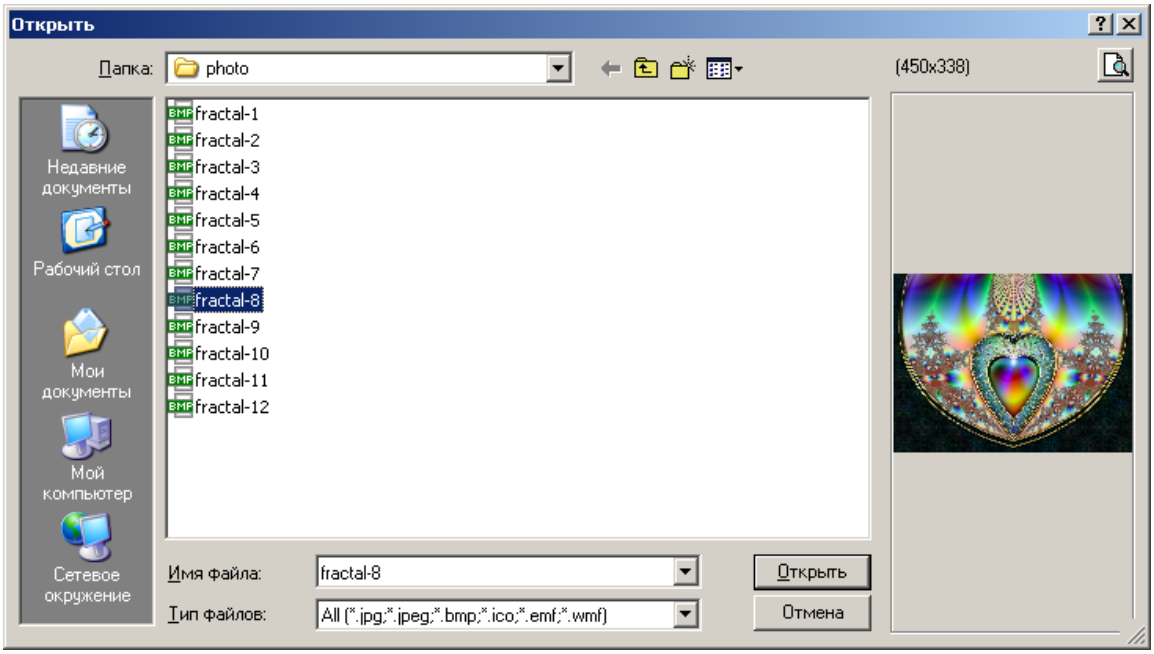
Відмінності компонента `OpenPictureDialog` від подібного компонента `OpenDialog` полягають у наявності фільтрів для вибирання лише графічних файлів з відповідними розширеннями та наявності у діалоговому вікні додаткової панелі попереднього перегляданння обраного файла.

Вікно форми проекту з результатами роботи програми за натискання на командні кнопки “Перейти до наступного фото” та “ Видалити фото з альбому” матиме вигляд:





При клацання на кнопки “Змінити фото на інше” та “Долучити нове фото до альбому” за допомогою компонента `OpenPictureDialog1` відкриватиметься діалогове вікно для обирання графічного файла:



Текст програми:

```
#include "jpeg.hpp"          // Забезпечує роботу з графічними JPEG-файлами
AnsiString albom="photo.dat"; // Ім'я файла зі списком файлів альбому
//---------------------------------------------------------------
// Клацання по обраному фото у списку
void __fastcall TForm1::ListBox1Click(TObject *Sender)
{ Image1->Picture->LoadFromFile(ListBox1->Items->
                                        Strings[ListBox1->ItemIndex]);
  StatusBar1->Panels->Items[0]->Text=IntToStr(ListBox1->
                                     ItemIndex+1)+"  файл зі списку";
  StatusBar1->Panels->Items[1]->Text="Файл: "+ListBox1->Items->
                                       Strings[ListBox1->ItemIndex];
}
//---------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{Image1->AutoSize=false;     // Заборонити змінювати розміри
                             // компонента залежно від розмірів зображення
 Image1->Proportional=true;// Зберігати пропорції зображень
 Image1->Stretch=0;          // Заборонити "підганяти" розміри
                             // зображення до розмірів компонента
 StatusBar1->Panels->Add();// Створити секцію на панелі (індекс –0)
 StatusBar1->Panels->Items[0]->Width=150; // Нова ширина першої секції
 StatusBar1->Panels->Add();// Створити ще одну секцію на панелі (індекс –1)
 if(FileExists(albom))       // Якщо альбом існує,
 { ListBox1->Items->LoadFromFile(albom); // завантажити список файлів
   if(ListBox1->Items->Count>0) // та, якщо список не є пустий,
   { ListBox1->ItemIndex=0;// виокремити саме перше у списку фото
```

```
      ListBox1Click(NULL);   // та показати його за допомогою
    }                        // команд функції ListBox1Click()
 }
 else  ShowMessage("Альбом порожній. Скористайтесь відовідною
                    кнопкою для долучення фото до альбому");
}
//---------------------------------------------------------
// Перейти до наступного фото
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ListBox1->ItemIndex++;
  ListBox1Click(NULL);
}
//---------------------------------------------------------
// Змінити фото на інше
void __fastcall TForm1::Button2Click(TObject *Sender)
{ if(OpenPictureDialog1->Execute())
  { // Обрати нове фото і вставити його на місце небажаного фото
    Image1->Picture->LoadFromFile(OpenPictureDialog1->FileName);
    ListBox1->Items->Insert(ListBox1->ItemIndex,
                                    OpenPictureDialog1->FileName);
    // Видалити небажане фото
    ListBox1->Items->Delete(ListBox1->ItemIndex);
 } }
//---------------------------------------------------------
// Видалити фото з альбому
void __fastcall TForm1::Button3Click(TObject *Sender)
{ int btn=MessageDlg("Ви дійсно бажаєте видалити обране фото
                     з альбому?", mtConfirmation,mbOKCancel,0);
  if(btn==1)                  // Якщо обрано кнопку OK, видалити фото
   { ListBox1->Items->Delete(ListBox1->ItemIndex); // зі списку
     if(ListBox1->Items->Count>0) // та, якщо інші фото залишились,
      { ListBox1->ItemIndex=0;    // виокремити саме перше у списку фото
        ListBox1Click(NULL);      // та показати його за допомогою
}  }  }                           // команд функції ListBox1Click()
//---------------------------------------------------------
// Долучити нове фото до альбому
void __fastcall TForm1::Button4Click(TObject *Sender)
{ if(OpenPictureDialog1->Execute())
  { Image1->Picture->LoadFromFile(OpenPictureDialog1->FileName);
    ListBox1->Items->Insert(ListBox1->ItemIndex+1,
                                    OpenPictureDialog1->FileName);
    ListBox1->ItemIndex++;
    ListBox1Click(NULL);
} }
//---------------------------------------------------------
// Зберігти вміст альбому
void __fastcall TForm1::Button5Click(TObject *Sender)
{ ListBox1->Items->SaveToFile("photo.txt"); // Зберегти список фото
}
```

Питання та завдання для самоконтролю

1) Чим схожі і чим відрізняються файли і масиви?

2) Який файл називають текстовим?

3) Який файл називають бінарним?

4) У чому полягає відмінність між бінарними і текстовими файлами?

5) Наведіть способи роботи з файлами в C++ Builder.

6) Для чого використовуються методи `LoadFromFile` та `SaveToFile`?

7) В якому заголовному файлі означено тип `FILE*`?

8) Що відбувається при відкриванні файла функцією `fopen()`?

9) Назвіть режими відкривання файлів у стилі C.

10) В який спосіб задається режим відкривання бінарного файла у стилі C?

12) Назвіть функції файлового форматованого записування-зчитування даних у стилі C.

12) Запишіть C-функцію створення текстового файла.

13) Якою C-функцією можна визначити розмір даних у файлі?

14) Наведіть фрагмент програми для дописування знаку оклику до першого рядка текстового файла типу `FILE*`.

15) Назвіть основні класи файлових потоків.

16) Запишіть операції “помістити в потік” та “узяти з потоку”.

17) Які режими відкривання файлів у стилі C++ використовуються для дописування даних у кінець файла?

18) Що таке дескриптор файла?

19) Наведіть функцію відкриття файла з ім’ям `DAN.txt` через дескриптор для зчитування і записування.

20) Яка функція дозволяє переміщувати поточну позицію зчитування-записування файла при опрацюванні через дескриптор.

21) Наведіть команди для визначення розміру файла при опрацюванні його через дескриптор.

22) Наведіть фрагмент коду для зчитування і виведення до компонента `Memo` вмісту текстового файла, кожен рядок якого містить два числа (ціле та дійсне) через пробіл.

23) Наведіть фрагмент коду для відкривання текстового файла типу `FILE*` і записування у його кінець структури `z` зі збереженням формату даних:

```
struct LITO { int sun; float sea; }z;
```

24) Наведіть команди для зчитування першого і останнього рядка текстового файла типу `FILE*`, кожен рядок якого містить два числа (ціле та дійсне) через пробіл, у змінну `z` типу `LITO`, оголошену у завданні 23.

25) Наведіть фрагмент коду, який збільшує вдвічі всі значення поля `sun` змінної `z` для файла, зазначеного у завданні 23.

26) Наведіть фрагмент коду, який з одного текстового файла типу `FILE*` створює два: у перший записує рядки, які розпочинаються з великої літери, а у другий – решту.

27) Наведіть фрагмент коду, який записує елементи дійсного масиву (вектора) у текстовий файловий потік у стилі C++:

а) всі елементи в один рядок через кому;

б) кожний елемент окремим рядком.

28) Наведіть команди для редагування 4-ої структури бінарного файлового потоку у стилі C++.

29) Наведіть і порівняйте команди для обнуління файла при опрацюванні у стилі C, C++ та через дескриптор.

30) Наведіть фрагмент коду, який записує елементи дійсної матриці 3×5 у текстовий файловий потік у стилі C++.

Розділ 13

Динамічні структури даних

Динамічна структура даних – це набір однотипних елементів, розміщуваних у пам'яті динамічно, тобто у процесі виконування програми за допомогою операції `new()` чи то функції `malloc()`. Прикладом динамічної структури даних є динамічний масив (див. розд. 6). Окрім динамічних масивів, широко застосовуються інші динамічні структури даних – *списки, стеки, черги* і *бінарні дерева*.

13.1 Поняття списку

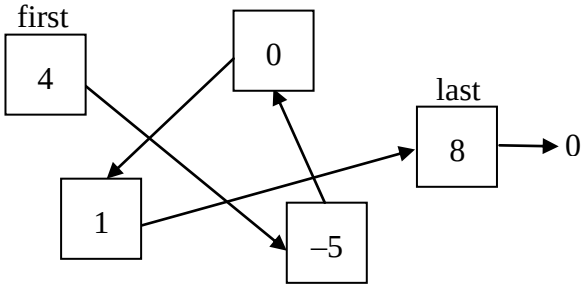
Список – динамічна структура, в якій одне (чи більше) з полів є адресами (чи вказівниками) на такі самі структури. Списки використовуються замість масивів для зберігання й опрацювання однотипних даних, кількість яких заздалегідь є невідома й може змінюватися у перебігу роботи. Окрім того, у списках доволі нескладно вилучити елемент чи то долучити новий на довільне місце.

У пам'яті одновимірний масив з 5-ти елементів (приміром з цілих чисел 4, –5, 0, 1, 8) розміщується в такий спосіб:

4	–5	0	1	8

Тобто всі елементи розташовані в пам'яті один за одним і це дає можливість їх нумерувати, але унеможливлює у перебігу роботи долучення нових елементів (принаймні це зробити непросто).

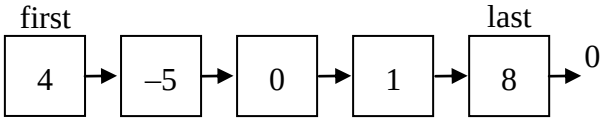
Елементи списку розташовуються у пам'яті хаотично. Це надає можливість безперешкодно долучати чи то вилучати яку завгодно кількість елементів. Наприклад, список з тими самими значеннями може бути розташований у пам'яті в такий спосіб:



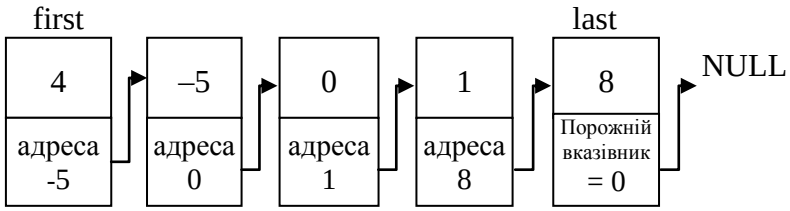
Стрілки на рисунку зазначають порядок проходження елементів. Для того щоб мати можливість працювати з елементами як зі зв'язною структурою, треба знати, де саме в пам'яті розташовано кожний з елементів, тобто знати їхні адреси. Як відомо, адреси змінних зберігаються у вказівниках. Виявляється, що немає потреби зберігати одразу всі адреси. Цілком достатньо зберігати лише адресу першого елемента (вказівник на перший елемент `first`). Кожному елементові треба "знати", де розташовано елемент, який слідує за ним, і необов'язково "знати", де розташовано решту. Тоді до кожного елемента можна буде діставатися послідовно: від першого до другого, від другого до третього і т. д.

У наведеному прикладі списку елемент зі значенням 1 “знає”, де розташовано елемент зі значенням 8 і не “знає”, де розташовано елемент зі значенням 0 і решта. Щоб “дістатися” елемента зі значенням 1, треба звернутися до першого елемента (4), довідатися в нього адресу другого (–5), у другого – довідатися адресу третього (0), а вже у третього – довідатися адресу четвертого (1). Безпосередньо одразу до четвертого звернутися неможливо.

Список для зручності й наочності можна зобразити в такий спосіб:



Отже, окрім даних, кожен елемент повинен зберігати ще одне значення: адресу наступного елемента. Природно є подати елемент списку у вигляді структури, яка містить одне чи то кілька інформаційних полів (полів даних) і вказівник на наступний елемент (який на рисунку зображується стрілкою). Після останнього елемента більше елементів немає, це означає, що вказівник на наступний елемент після останнього є порожнім (NULL), інакше кажучи, дорівнює 0. Це можна зобразити в такий спосіб:



У верхній частині кожного елемента зображено його поле даних (число), а в нижній – поле `next` (адреса наступного елемента в пам’яті).

Оголошення типу “елемент списку” має такий загальний вигляд:

```
struct <елемент списку>
{ <оголошення полів даних>
  <елемент списку>* <вказівник на наступний елемент>;
};
```

Тип елемента наведеного вище списку може бути оголошено у формі

```
struct Element
{ int d;              // ціле число
  Element *next;      // вказівник на наступний елемент
};
```

Вказівник на перший елемент цього списку оголошується як

```
Element *first=0;
```

Його початкове значення завжди дорівнює 0 (тобто списку покищо немає). Після створення списку `first` набуває певного ненульового значення (адреса пам’яті, у якій розташовано перший елемент).

Знаючи вказівник на перший елемент, до другого елемента можна звернутися в такий спосіб: `first->next`, а до третього: `first->next->next` і т. д. Числове значення першого елемента: `first->d`.

Зверніть увагу на те, що `first` є не власне елементом списку, а лише вказівником на нього, тобто є вказівником на структуру. Тому звертання до полів структури відбувається не через крапку, а через стрілочку **->** (символи "–" й ">").

Для початкового *створення списку* слід створити перший елемент:

1) виділити під нього місце у пам'яті:

```
first=new Element;
```

2) занести в нього числове значення:

```
first->d=4;
```

3) обнулити вказівник на наступний елемент:

```
first->next=NULL;
```

4) вказівнику на останній елемент `last` присвоїти адресу створеного тільки що першого елемента, оскільки інших елементів поки немає:

```
last=first;
```

Для *долучення нового елемента* `c` *в кінець списку* слід виконати такі дії:

1) виділити місце в пам'яті під новий елемент:

```
c=new Element;
```

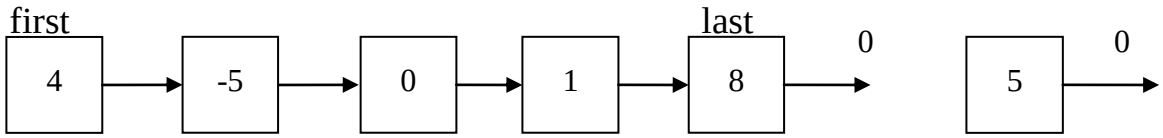
2) занести до нього поля даних (приміром ціле значення 5):

```
c->d=5;
```

3) вказівник на наступний елемент є пустий, а, оскільки наступних елементів покищо немає, йому слід присвоїти значення NULL (0):

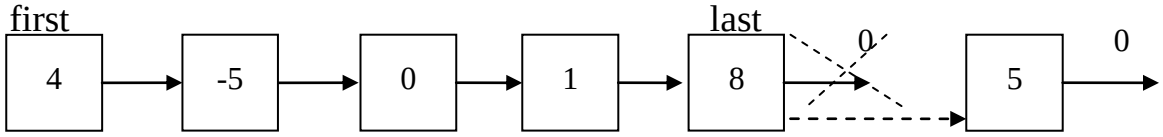
```
c->next=NULL;
```

Схематично це можна зобразити у формі



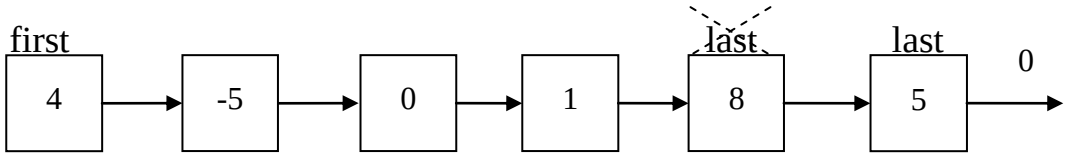
4) створений елемент слід долучити до списку після останнього елемента, на який указує `last`. Тобто елемент `c` повинен стати наступним після `last`. Для цього адресу нового елемента слід записати у поле `next` останнього елемента:

```
last->next=c;
```



5) присвоїти вказівникові на останній елемент `last` вказівник на новий елемент `c`:

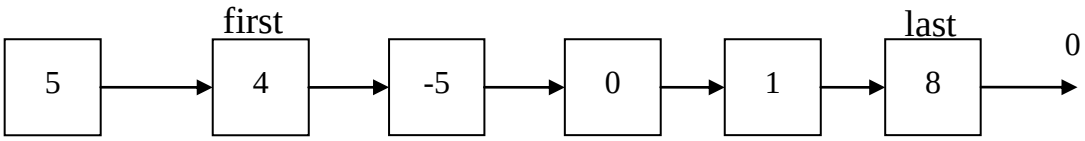
```
last=c;
```



Якщо треба долучити *новий елемент перед першим*, вище наведені пункти 3 – 5 слід змінити на два інших:

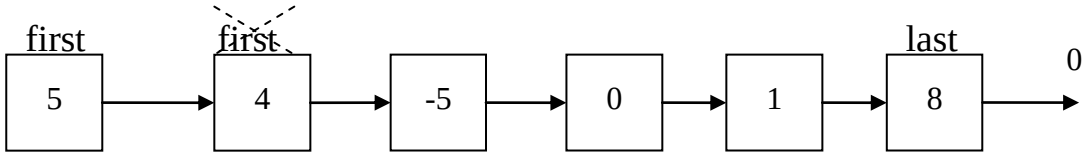
3) новий елемент долучити до списку, тобто в його поле `next` записати адресу першого елемента:

```
c->next=first;
```



4) вказівникові `first` присвоїти вказівник на новий елемент:

```
first=c;
```



Проходження за списком виконується переважно у циклі `while`. Допоки не дістались потрібного елемента чи то допоки список не завершено, переходимо до наступного елемента за його адресою. Для цього використовується допоміжний вказівник `c`, який оголошується як

```
Element *c;
```

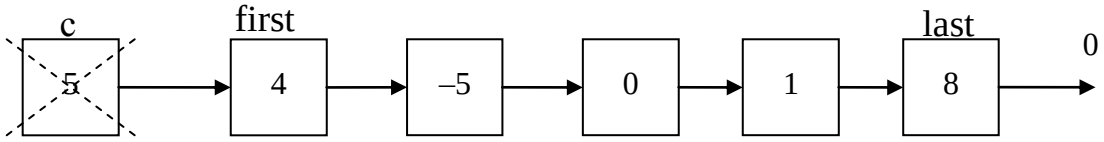
Спочатку треба перемістити до першого елемента:

```
c=first;
```

У циклі, допоки список не завершено, тобто допоки елемент `c` не дорівнює 0, з черговим елементом списку виконується потрібна дія, після чого здійснюється перехід до наступного елемента списку:

```
while(c!=0)    // Допоки список не завершено,
{ . . .        // виконання дії з c->d
  c=c->next;   // й перехід до наступного елемента списку.
}
```

При *вилученні першого елемента списку* його адреса запам'ятовується в допоміжному вказівнику `c`, а вказівник `first` переміщується на другий елемент. Після цього пам'ять від елемента `c` очищується:

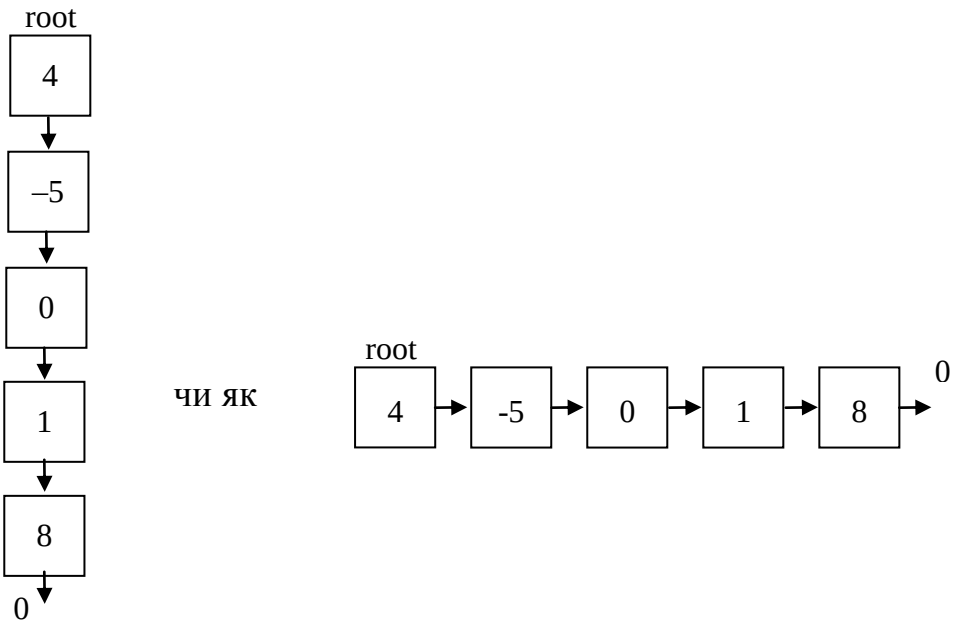


Вставлення й вилучення елементів з середини списку виконується простим присвоюванням адрес і буде розглянуто пізніше разом з прикладами програм. Найбільш простими різновидами списків є СТЕК і ЧЕРГА.

13.2 Стек

Стек – це список, елементи якого можна долучати й вилучати лише з одного кінця. Щоб зрозуміти, що таке стек, слід уявить собі трубку, один з кінців якої запаяно. Будемо закочувати до неї кульки. Всі кульки всередині трубки, окрім останньої, стають неприступними. Щоб вийняти, приміром, третю зверху кульку, треба спочатку вийняти (тобто вилучити зі стека) дві верхніх кульки. Після цього третя кулька стане верхньою і її можна буде витягти. Для стека існує спеціальний термін – FILO ("First In – Last Out", тобто "першим прийшов – підеш останнім").

Зобразити стек можна як



Для роботи зі стеком слід зберігати вказівник на вершину стека (`root`). Кожен елемент повинен зберігати адресу наступного елемента (тобто елемента, який розташований нижче за нього й було долучено до стека раніш за нього).

Елемент стека, як і кожного списку, є структурою, яка містить одне чи більше полів даних і адресу попереднього елемента (тобто вказівник на такий самий елемент). Тип елемента стека оголошується так само, як і елемент списку, наприклад тип елемента наведеного стека

```
struct Elem
{ int d;          // d – числове поле
  Elem *next;     // next – вказівник на наступний елемент
};
```

Початкове оголошення вершини стека:

```
Elem *root=0;
```

У стеку на останньому рисунку елемент зі значенням 8 було долучено першим, а вилучено може бути лише після того, як усю решту елементів буде вилучено. Елемент зі значенням 4 було долучено останнім, а вилучено може бути лише першим. Новий елемент до стека може бути долучено лише вище (ліворуч) від 4. При долученні нового елемента до цього стека поле `next` нового

елемента буде зберігати адресу попереднього елемента, тобто елемента зі значенням 4. Окрім того, після долучення вершиною стане новий елемент.

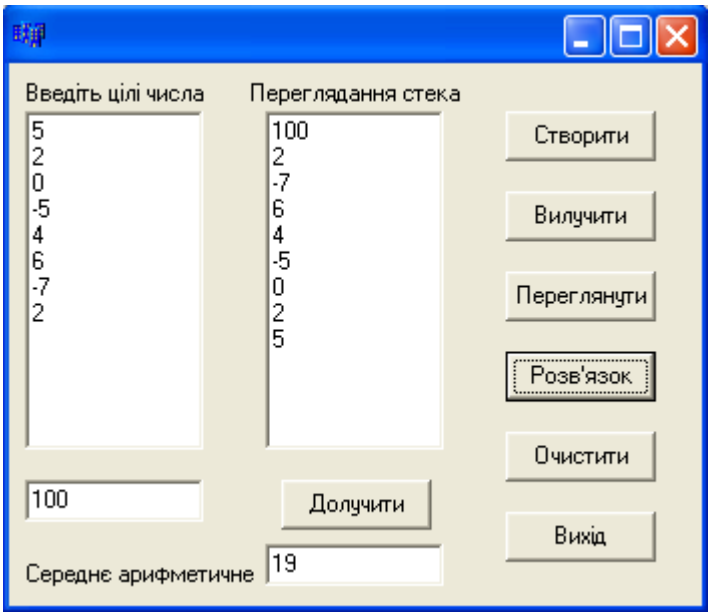
Так само, як і зі списком, для роботи зі стеком використовується проміжний вказівник `c`.

З елементами стека можна виконувати такі дії: долучення нового елемента до стека (аналогічно до долучення нового елемента до списку перед першим елементом), вилучення верхнього елемента стека (аналогічно до вилучення першого елемента списку), переглядання елементів стека (аналогічно до переглядання елементів списку).

Приклад 13.1 Створити стек, який міститиме цілі числа, і передбачити такі можливості:

- ✓долучення елемента до стека;
- ✓вилучення верхнього елемента стека;
- ✓виведення вмісту стека в `Memo`;
- ✓обчислення середнього арифметичного парних значень елементів стека;
- ✓очищення пам'яті від стека.

Форма додатка з результатами роботи матиме вигляд



Текст програми:

```
struct Elem       // Оголошення елемента стека
{ int x;  Elem *next;
};
Elem *root=0;     // Оголошення вершини стека і початкове обнулення
void add (int d) // Долучення елемента зі значенням d до стека
{ // Оголошення й розміщення у пам'яті допоміжного елемента
  Elem *c=new Elem;
  c->x=d;
  c->next=0;      // Попередній елемент покищо не визначено.
  c->next=root;  // Новий елемент пов'язується зі старою вершиною стека
  root=c;         // і після цього він стає новою вершиною.
}
```

```
void del()        // Вилучення верхнього елемента стека
{ Elem *c=root;  // Запам'ятовування першого елемента у додатковій змінній c
  root=root->next;  // Другий елемент тепер стає першим
                    // (root->next – це звертання до другого елемента).
  delete c;         // Звільнення пам'яті від елемента (тобто вилучення його)
}

void print()    // Виведення елементів стека до Memo
{ Form1->Memo2->Clear();
  Elem *c=root;
  while (c!=0)
  { Form1->Memo2->Lines->Add(IntToStr(c->x));
    c=c->next;    // Перехід до наступного елемента стека
  }
}

// Обчислення середнього арифметичного парних значень елементів стека
float srednee()
{ int s=0, k=0;
  float sr;
  Elem *c=root;
  while(c!=0)
  { if(c->x % 2==0)  // Перевірка, якщо значення елемента стека є парне,
     { s+=c->x;      // долучення його до суми
       k++;          // і збільшення кількості на 1.
     }
    c=c->next;       // Перехід до наступного елемента
  }
  if(k!=0) sr=(float)s/k;
  return sr;         // Повернення середнього арифметичного
}

// Очищення пам'яті від стека
// (використовується функція вилучення вершини стека)
void clean()
{ while(root!=0)     // Допоки стек є непорожній (вершина не 0),
     del();          // вилучення вершини.
}

//Кнопка "Долучити"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ add(StrToInt(Edit1->Text));
}

// Кнопка "Створити"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ int n=Memo1->Lines->Count;
  for(int i=0; i<n; i++)
    add(StrToInt(Memo1->Lines->Strings[i]));
}
```

```
// Кнопка "Вилучити"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ del();
  print();
}
// Кнопка "Переглянути"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ print();
}
// Кнопка "Розв'язок"
void __fastcall TForm1::Button5Click(TObject *Sender)
{ if(root==0){ ShowMessage("Стек є порожній");  return;  }
  Edit2->Text=FloatToStr(srednee());
  while(root != 0 && root->x <= 0)  del();
}
// Кнопка "Очистити"
void __fastcall TForm1::Button6Click(TObject *Sender)
{ while(root!=0) del();   // Очищення пам'яті
  Memo1->Clear();  Memo2->Clear();        // Очищення форм
  Edit1->Clear();  Edit2->Clear();
}
```

13.3 Черга

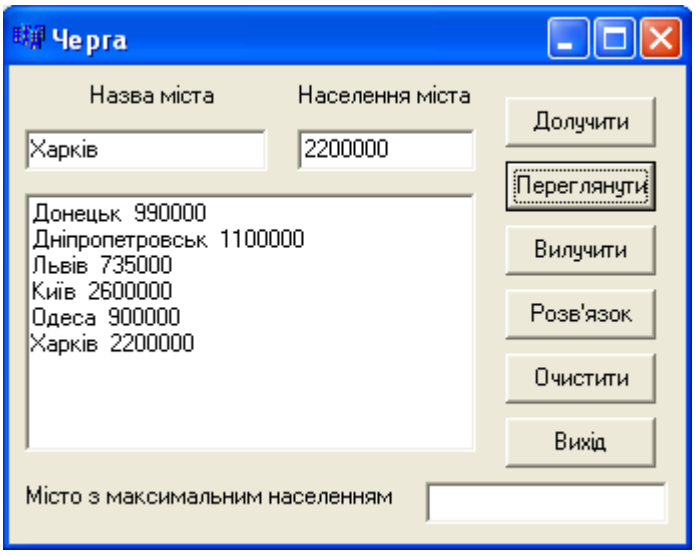
Черга – це різновид списку, який має таку особливість: нові елементи можна долучати лише в кінець черги, а вилучати можна лише перший елемент черги. Уявімо собі чергу до каси магазину. Нові покупці стають у кінець черги (після останнього покупця), а розплачується і йде спочатку той, хто стоїть першим. Для черги існує спеціальний термін FIFO ("first in – first out", тобто "першим прийшов – першим пішов").

З елементами черги можна виконувати такі дії: долучення нового елемента до черги (аналогічно до долучення нового елемента у кінець списку після останнього), вилучення першого елемента черги (аналогічно до вилучення першого елемента списку), переглядання елементів черги (аналогічно до переглядання елементів списку).

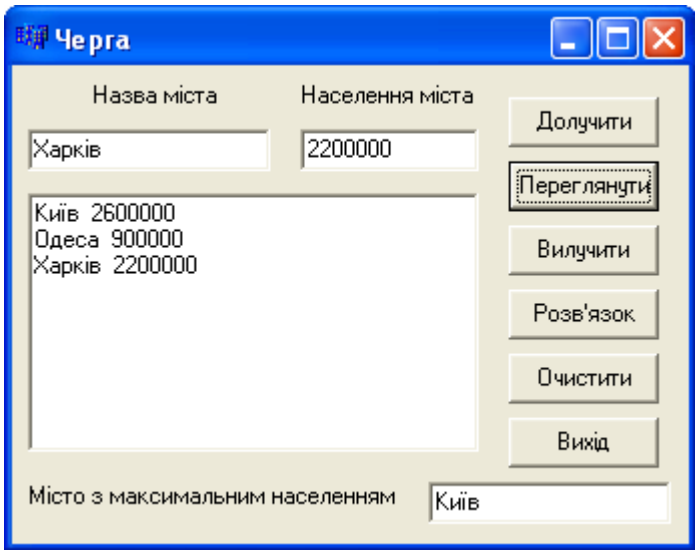
Приклад 13.2 Створити чергу, елементи якої містять інформацію про назву міста і його населення. Передбачити такі можливості:

- ✓ долучення елемента до черги;
- ✓ вилучення першого елемента черги;
- ✓ виведення вмісту черги до `Memo`;
- ✓ очищення пам'яті від черги;
- ✓ вилучення з черги інформації про всі міста до міста з максимальним населенням.

Форма після введення даних матиме вигляд:



Після натискання на кнопку “Розв’язок” форма матиме вигляд:



Текст програми:

```
struct Elem             // Оголошення елемента черги
{ AnsiString gorod;     // Рядок – назва міста
  int nas;              // Ціле число – кількість населення міста
  Elem *next;           // Вказівник на наступний елемент черги (адреса)
};
// Оголошення вказівників на початок і кінець черги та їхнє початкове обнулення
Elem *first=0, *last=0;
void add (AnsiString S, int d)  // Долучення елемента до черги
{ Elem *c=new Elem;     // Виділення пам’яті під новий елемент черги
 c->gorod=S;            // Занесення до нового елемента назви міста
 c->nas=d;              // Занесення до нового елемента кількості населення
 c->next=0;             // Після нового елемента інших елементів покищо немає
 if(first==0)first=c;// Якщо черги ще нема, то новий елемент стає першим
 else                   // Інакше елемент долучається після останнього,
   last->next=c;        // записуючи в last адресу нового елемента
```

```
  last=c;              // Тепер новий елемент стає останнім
}
void del()             // Вилучення першого елемента черги
{ if(first==0)  // Якщо черга є порожня, виводиться повідомлення і виконання
  { ShowMessage("Черга порожня"); return; }    // переривається
  Elem *c=first;
  first=first->next; // first указує на той елемент, який раніше був другим
  delete c;            // Знищення елемента, який вилучається
}
void print()           // Виведення черги в Мето
{ Form1->Memo1->Clear();
  Elem *c=first;
  while(c!=0)
  { Form1->Memo1->Lines->Add(c->gorod+"  "+IntToStr(c->nas));
    c=c->next;         // Перехід до наступного елемента черги
} }
AnsiString udal()
// Вилучення з черги всіх міст  до міста з максимальним населенням
{ if(first==0)               // Якщо черга є порожня,
  { ShowMessage("Черга порожня");    // виводиться повідомлення
    return ""; }                  // і виконання переривається
  //Пошук міста з максимальним населенням
  int max=first->nas;      // Початкове значення максимального населення –
// населення першого міста
  AnsiString maxgor=first->gorod;    // Початкове значення міста
// з максимальним населенням - назва  першого міста.
  Elem *c=first->next;     // Цикл розпочинається з другого елемента черги
  while(c!=0)
  {if(c->nas > max)  // Якщо населення міста більше максимального,
    { max=c->nas;    // у тах запам'ятовується кількість населення міста
      maxgor=c->gorod; }  // та його назва
  c=c->next;
  }
  // Вилучення з черги всіх міст аж до міста з максимальним населенням
  c=first;
  while(c!=0 && c->gorod!=maxgor)    // Допоки не дісталися міста,
  { del();//з максимальним населенням, вилучається перший елемент черги
    c=first; }
  return maxgor;
}
void clean ()          // Очищення пам'яті
{ while(first!=0) del(); }
// Кнопка "Долучити"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString S=Edit1->Text;
  int d=StrToInt(Edit2->Text);
  add (S, d);
}
```

```
// Кнопка "Переглянути"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ print();
}
// Кнопка "Вилучити"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ del();  print();
}
//Кнопка "Розв'язок"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ Edit3->Text = udal();
  print();
}
// Кнопка "Очистити"
void __fastcall TForm1::Button5Click(TObject *Sender)
{ clean();
  Edit1->Clear();  Edit2->Clear();
  Edit3->Clear();  Memo1->Clear();
}
```

13.4 Вставлення і вилучення елементів списку

Список дозволяє вставити новий елемент поміж якими завгодно елементами. Вставлення перед першим і після останнього елементів було розглянуто у підрозд. 13.1.

Для *вставлення* нового елемента `c` *до списку після елемента*, на який указує вказівник `c1`, слід виконати такі дії:

1) виділити місце в пам'яті під новий елемент `c` і записати до нього дані;

2) пов'язати новий елемент з тим, який іде після `c1` (на нього вказує вказівник `c1->next`), для чого до поля `next` елемента `c` записати адресу елемента `c1->next`:

```
c->next = c1->next;
```

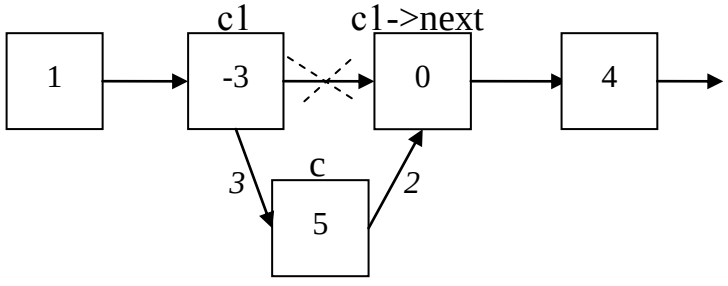
3) пов'язати новий елемент `c` з `c1`, для чого до поля `next` елемента `c1` записати адресу елемента `c`:

```
c1->next = c;
```

4) якщо елемент `c1` був останнім, то тепер останнім буде елемент `c`:

```
if(c1==last) last = c;
```

На рисунку дії 2 та 3 позначені відповідними цифрами:



Якщо у програмі описано функцію вставлення нового елемента після зазначеного, її можна використовувати при створенні списку, коли новий елемент вставляється після останнього.

При *вилученні* елемента `c` *зі списку* треба спочатку знайти вказівник на попередній елемент `c1`. Після цього слід виконати такі дії:

1) пов'язати елемент `c1` з тим, який розташований після `c` (на нього вказує вказівник `c->next`). Для цього до поля `next` елемента `c1` записати адресу елемента `c->next`:

```
c1->next = c->next;
```

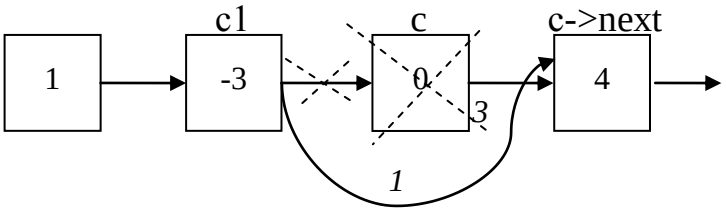
2) якщо елемент `c` був останнім, тепер останнім стає `c1`:

```
last = c1;
```

3) звільнити пам'ять від елемента `c`:

```
delete c;
```

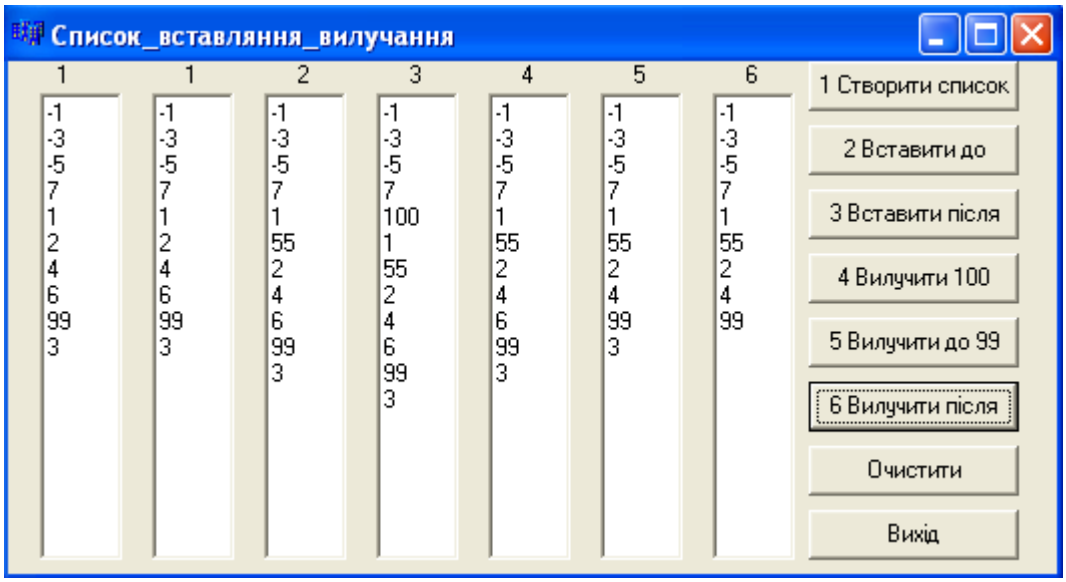
На рисунку дії 1 та 3 позначені відповідними цифрами:



Приклад 13.3 Увести цілі числа до `Memo` і створити з цих чисел список. Передбачити такі можливості:

- ✓ вставлення нового елемента зі значенням 55 перед першим елементом з додатним значенням;
- ✓ вставлення нового елемента зі значенням 100 після першого елемента з парним значенням;
- ✓ вилучення першого елемента зі значенням 100;
- ✓ вилучення елемента перед першим елементом зі значенням 99;
- ✓ вилучення елемента після першого елемента зі значенням 100.

Форма додатка з результатами роботи матиме вигляд



Текст програми:

```
struct Element//Оголошення елемента списку
{ int d;  Element *next; };
// Оголошення вказівників на перший та останній елементи (глобальні змінні)
Element *first=0, *last=0;
// Функція створення першого елемента списку
// Параметр – число, яке буде записано до нового елемента
void fir(int x)
{ first =new Element;
  first->d=x;
  first->next=0;
}
// Функція виведення списку до Memo
// Параметр memo – компонент Memo, до якого виводитиметься список
void print_list(TMemo *memo)
{ Element *c=first;
  while(c!=0){ memo->Lines->Add(IntToStr(c->d));  c=c->next; }
}
void ochistka()// Функція очищення пам'яті
{ Element *c=first;
  while(first!=0)
   { first=first->next;  delete c;  c=first; }
}
// Функція для вставлення елемента зі значенням x до списку після елемента c1
void vstavka(Element* c1, int x)
{ Element* c=new Element;
  c->d=x;
  c->next=c1->next;
  c1->next=c;
  if(c1==last) last=c;
}
//Функція вилучення зі списку елемента, що йде після елемента c1
void del_el(Element* c1)
{ Element* c=c1->next;
  c1->next=c->next;
  if(c==last) last=c1;
  delete c;
}
// Кнопка "1 Створити список"
// Тут створюється список із чисел в  Memo1 і виводиться цей список до  Memo2
void __fastcall TForm1::Button1Click(TObject *Sender)
{ first=0;  last=0;  int i,n=Memo1->Lines->Count;
  for(i=0; i<n; i++)
   if(first==0)
    { fir(StrToInt(Memo1->Lines->Strings[i])); last=first; }
   else vstavka(last, StrToInt(Memo1->Lines->Strings[i]));
  print_list(Memo2);
}
```

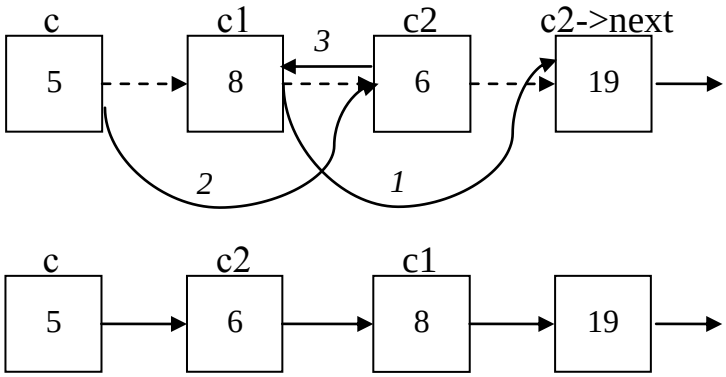
```
//-------------------------------------------------------------
// Кнопка "2 Вставити до"
// Вставити елемент зі значенням 55 перед першим парним числом
void __fastcall TForm1::Button2Click(TObject *Sender)
{ bool ok=false;     // Ознака: новий елемент не вставлено
  Element* c1=first;
  if (first->d%2==0) // Якщо значення першого елемента є парне,
  { Element* c=new Element;
    c->d=55;
    c->next=first;   // вставляється новий елемент перед першим,
    first=c;         // а першим стає новий елемент;
    ok=true;         // ознака: новий елемент вставлено
  }
  else               //Проходження за списком до передостаннього елемента
  while (c1->next!=0)
  { if(c1->next->d%2==0)  // Якщо знайдено парний наступний елемент,
    { vstavka(c1, 55);    // вставляється новий елемент
      ok=true;
      break;              // і цикл зупиняється
    }
    c1=c1->next;
  }
  if(ok==false)
    {ShowMessage ("Парного елемента у списку нема");  return;}
  print_list(Memo3);
}
//-------------------------------------------------------------
// Кнопка "3 Вставити після"
// Вставити елемент зі значенням 100 після першого додатного числа
void __fastcall TForm1::Button3Click(TObject *Sender)
{ Element *c1=first;    bool ok=false;
  while(c1->next!=0)
  { if(c1->d>0)           // Якщо елемент є додатний,
    { vstavka(c1, 100);   // вставляється новий елемент після нього
      ok=true; break;
    }
    c1=c1->next;
  }
  if(ok==false)
   {ShowMessage("Додатного елемента у списку нема"); return; }
  print_list(Memo4);
}
//-------------------------------------------------------------
// Кнопка "4 Вилучити 100"
// Відбувається вилучення усіх елементів списку зі значенням 100
void __fastcall TForm1::Button4Click(TObject *Sender)
{ Element* c1=first;    bool ok=false;
  if(first->d==100)       // Якщо перший дорівнює 100,
```

```
    { first=first->next;    // першим стає той, що був другим
      delete c1;            // і вилучається той, що спочатку був першим
      ok=true;
    }
    else
     while(c1->next!=0)
      {if(c1->next->d==100)      // Якщо наступній елемент дорівнює 100,
        { del_el(c1);            // відбувається його вилучення
          ok=true;
          break;
        }
        c1=c1->next;
      }
  if(ok==false) {ShowMessage ("Елемента 100 нема");  return; }
  print_list(Memo5);
}
//---------------------------------------------------------------
// Кнопка "5 Вилучити до 99"
// Вилучити елемент до першого елемента зі значенням 99
void __fastcall TForm1::Button5Click(TObject *Sender)
{ Element* c1=first;
  bool ok=false;
  if(first->next->d==99)  // Якщо значення другого елемента 99,
   {first=first->next;    // першим стає той, що був другим (99)
    delete c1;            // і вилучається перший
    ok=true;
   }
   else
// Проходження за списком, допоки існують два наступні елементи
    while(c1->next->next!=0)
    { if(c1->next->next->d==99)// Якщо наступний після наступного є  99,
      { del_el(c1);              // вилучається поточний елемент
        ok=true;
        break;    }
      c1=c1->next;
    }
  if(ok==false)
   {ShowMessage ("Елемент вилучити неможливо");  return;}
  print_list(Memo6);
}
//---------------------------------------------------------------
// Кнопка "6 Вилучити після" – вилучити елемент після елемента зі значенням 99
void __fastcall TForm1::Button6Click(TObject *Sender)
{ bool ok=false;    Element* c1=first, *c;
  while(c1->next!=0)
  { if(c1->d==99)    // Якщо поточний елемент має значення 99,
     {del_el(c1);    // вилучається наступний
      ok=true;
      break;
```

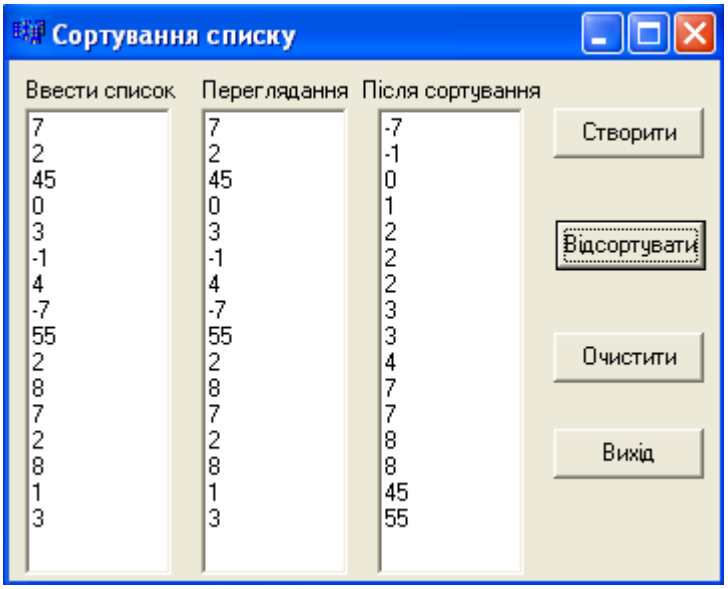
```
      }
     c1=c1->next;
   }
   if(ok==false)
    { ShowMessage("Елемент вилучити неможливо");  return; }
   print_list(Memo7);
}
//---------------------------------------------------------------
//Кнопка "Очистити".
void __fastcall TForm1::Button7Click(TObject *Sender)
{ ochistka();        Memo1->Clear();
  Memo2->Clear();    Memo3->Clear();  Memo4->Clear();
  Memo5->Clear();    Memo6->Clear();  Memo7->Clear();
}
```

Приклад 13.4 Заповнити список цілими числами з `Memo` і відсортувати список за зростанням.

Розв'язок. У підґрунті кожного сортування лежить переставляння значень елементів. Якщо розмір елементів та їхня кількість є надто великі, обмін значеннями займає багато часу. Тому ефективніше при сортуванні не обмінювати значення елементів, а змінювати їхній порядок у списку, тобто змінювати адреси наступних елементів у полі `next`.



Форма додатка з результатами роботи матиме вигляд



Оскільки більшість операцій зі списком було розглянуто у прикладі 13.3, наведемо текст лише кнопки “Сортувати”:

Текст програми:

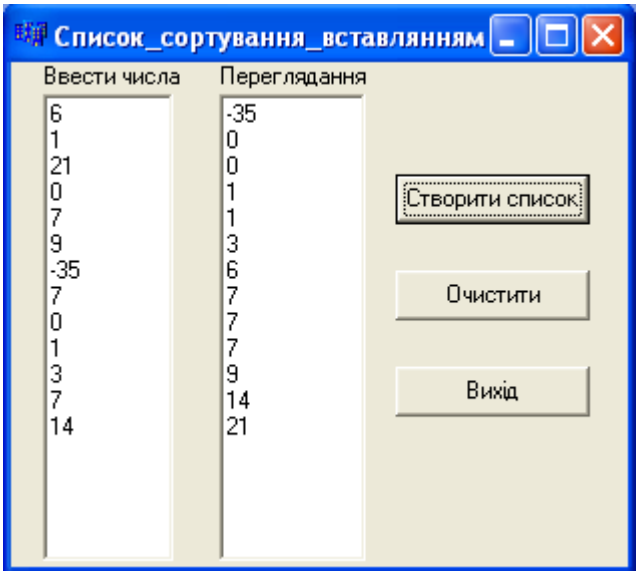
```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Element *c, *c1, *c2;
  bool pr;           // Ознака, чи відсортовано масив
  do
  { pr=true;
    c=first; c1=first->next;
    if (c->d > c1->d)// Якщо перший і другий елементи йдуть неправильно,
    { first=c1;           // другий елемент стає першим,
      c->next=c1->next;   // той, що був першим, зв'язується з третім,
      first->next=c;      // а той, що був першим, стає другим
      pr=false;           // Ознака: зміни відбулися, масив не відсортовано
    }
    // Запам'ятовування трійки елементів, які йдуть один за одним.
    c=first; c1=first->next; c2=c1->next;
    while(c2!=0)          // Допоки останній у трійці не 0,
    { if(c1->d > c2->d)   // якщо елементи c1 та c2 йдуть неправильно,
       {c1->next=c2->next;      // переставляння елементів
        c->next=c2;             // згідно з наведеним рисунком
        c2->next=c1;
        c=c2; c2=c1->next;      // Присвоювання для наступного кроку
        pr=false; // Ознака: зміни відбулися, масив не відсортовано
       }
      else
      { c=c1;
        c1=c2;
        c2=c2->next;
      }                         // Присвоювання для наступного кроку
    }
  }while (!pr);   // Допоки жодної зміни не відбудеться (масив відсортовано)
  print_list(Memo3);
}
```

Приклад 13.5 Увести числа до `Memo` і створити відсортований за зростанням список.

Розв'язок. Використовуватимемо алгоритм сортування вставленням. На кожному кроці припускається, що існуючий список вже відсортований і треба вставити новий елемент до списку в такий спосіб, щоб не порушити порядок. Для цього будемо йти за списком і зупинимось, коли попередній елемент буде менш за нове число, а поточний не перевищуватиме (це означає, що треба вставити новий елемент поміж ними).

Основні функції для вставлення елемента до списку було розглянуто у прикладі 13.3. Тому наведемо лише текст кнопки “Створити список”.

Форма додатка з результатами роботи матиме вигляд



Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ first=0; last=0;   int i, n=Memo1->Lines->Count;
  for(i=0; i<n;i++)
   if(first==0)
    { fir(StrToInt(Memo1->Lines->Strings[i]));
      last=first;
    }
   else
    { if(first->d>StrToInt(Memo1->Lines->Strings[i]))
       { Element * c=new Element;
         c->d=StrToInt(Memo1->Lines->Strings[i]);
         c->next=first;
         first=c;
       }
      else
       if(last->d<StrToInt(Memo1->Lines->Strings[i]))
         vstavka(last, StrToInt(Memo1->Lines->Strings[i]));
       else
         {Element* c1=first, * c2=first->next;
          while(c2!=0)
          {if (c1->d<=StrToInt(Memo1->Lines->Strings[i])&&
              c2->d>StrToInt(Memo1->Lines->Strings[i]))
              {vstavka(c1, StrToInt(Memo1->Lines->Strings[i]));
               break;
              }
           c1=c2;
           c2=c2->next;
        } }
   }
  print_list(Memo2);
}
```

Приклад 13.6 Заповнити StringGrid цілими випадковими числами. Створити список з чисел, модуль яких є менше за 5. Обчислити середнє арифметичне додатних елементів списку. Вилучити елементи, які є менше за –1.

Розв'язок. У цьому прикладі оголосимо вказівники на перший та останній елементи локально. Оскільки в такому разі ці вказівники невідомі багатьом функціям, доведеться передавати вказівники на початок і кінець списку як параметри.

У багатьох програмах зі списками доводиться виконувати одні й ті ж самі дії (тобто писати аналогічні функції). Створимо заголовний файл з прототипів згаданих функцій. Файл Unit2.h міститиме оголошення типу Element "елемент списку" і прототипи таких функцій: створення першого елемента fir(), долучення нового елемента у кінець списку add(), вилучення елемента після зазначеного del_el(), вставлення елемента після зазначеного vstavka(), очищення пам'яті, яку було виділено під елементи списку ochistka().

Оскільки всі зазначені функції розміщено у файлі Unit2.h, змінні first та last, які оголошено в Unit1.cpp, у них є невідомі. Тому вказівники на початок і кінець списку мають бути параметрами функцій.

Функція долучення елемента в кінець списку має тип void. Параметри функції: вказівник на перший елемент, вказівник на вказівник на останній елемент (адреса вказівника на останній елемент) і число, яке буде записано до нового елемента. Перший та останній параметри функція використовуватиме, але не змінюватиме, а ось вказівник на останній елемент списку при долученні нового елемента в кінець неодмінно зміниться (останнім тепер стане новий елемент). Тому функція повинна мати доступ для змінювання цього вказівника. Для цього його має бути передано до функції за адресою. Вказівник на останній елемент має тип Element*. Отже, адреса вказівника на останній елемент має тип вказівник на Element* , тобто Element **.

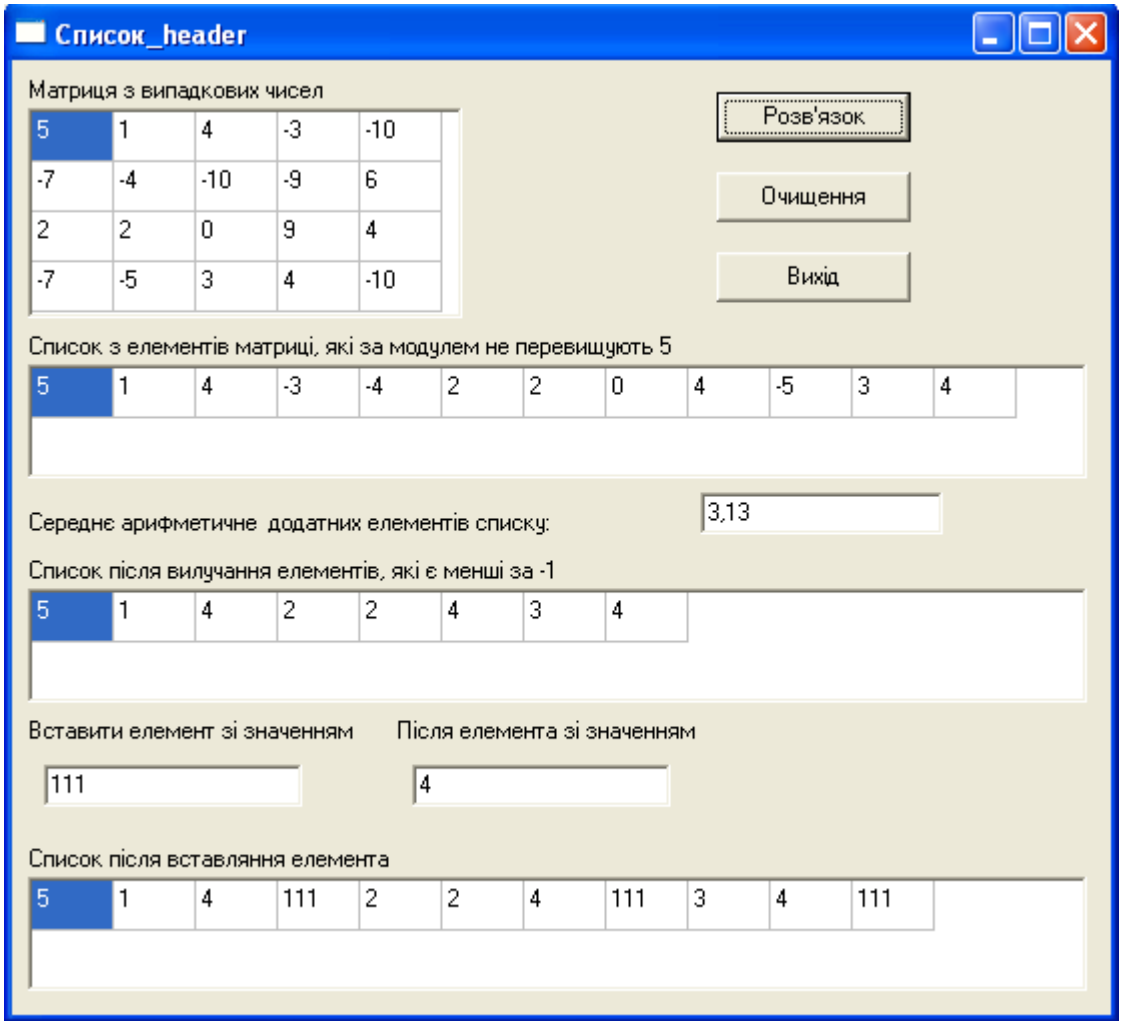
Оскільки las – це вказівник на вказівник на останній елемент, то слід користуватися операцією розадресації *las для звертання до вказівника на останній елемент.

Параметри функції вилучення елемента зі списку – вказівник c1 на елемент, який передує тому, що буде видалено, і вказівник на останній елемент списку las, який може бути змінено. Аналогічно до функції долучення, треба писати перед las дві "зірочки", щоб забезпечити доступ для змінювання вказівника на останній елемент списку.

Файл Unit2.h:

```
struct Element
{ int d;  Element *next;};
Element* fir(int x);
void add(Element* fir, Element** las, int x);
void del_el(Element* c1, Element ** las);
void vstavka(Element* c1, Element** las, int x);
void ochistka(Element *fir);
```

Форма додатка з результатами роботи матиме вигляд



Файл `Unit2.cpp`:

```
// Створення першого елемента списку
Element* fir(int x)
{ Element *c=new Element;
  c->d=x;
  c->next=0;
  return c;
}
// Функція долучення елемента у кінець списку
void add(Element* fir, Element** las, int x)
{ Element *c=new Element;
  c->d=x;
  c->next=0;
  (*las)->next = c;
  *las = c;
}
// Функція вставлення елемента до списку
void vstavka(Element* c1, Element** las, int x)
{ Element* c=new Element;
  c->d=x;
  c->next=c1->next;
```

```
  c1->next=c;
  if(c1==(*las)) (*las)=c;
}
// Функція вилучення елемента зі списку
void del_el(Element* c1, Element** las)
{ Element* c=c1->next;
  c1->next=c->next;
  if(c==(*las)) (*las)=c1;
  delete c;
}
// Функція звільнення пам'яті від списку
void ochistka(Element *fir)
{ Element *c=fir;
  while (fir!=0)
  { fir=fir->next;
    delete c;
    c=fir;
  }
}
```

Файл Unit1.cpp:

```
#include "Unit2.h"
// Функція виведення списку до StringGrid
// (параметри – вказівник на початок списку і StringGrid, до якого виводиться список)
void setka(Element *fir, TStringGrid *sg)
{ Element *c=fir;  int n=0;
  while(c!=0)
  { sg->Cells[n][0]=c->d;
    n++;
    c=c->next;
  }
  sg->ColCount=n;
}
// Функція обчислювання середнього арифметичного додатних елементів списку
float sr_ar(Element * fir)
{ int sum=0, kol=0;
  Element *c=fir;
  while(c!=0)
  { if(c->d>0) { sum+=c->d;  kol++; }
    c=c->next;
  }
  return 1.0*sum/kol;
}
// Функція вилучення зі списку елементів з числовим значенням менше за –1
void vyluch (Element ** fir, Element** las)
{ Element* c1=(*fir);
  bool ok=false;
  while((*fir)->d<-1)      //Якщо значення першого елемента є менше за –1,
  { (*fir)=(*fir)->next;  //першим стає той, що був другим
```

```
    delete c1;              // і вилучається той, що спочатку був першим
    ok=true;
  }
  while (c1->next!=0)
  { if(c1->next->d < -1)
     { del_el(c1, las); ok=true;  }
    else c1=c1->next;
  }
  if(ok==false)
   {ShowMessage("Елементів < -1 нема");  return;}
}
//---------------------------------------------------------------
// При запуску форми StringGrid1 заповнюється випадковими числами
void __fastcall TForm1::FormCreate(TObject *Sender)
{ randomize();
  for(int i=0; i<4; i++)
  for(int j=0; j<5; j++)
    StringGrid1->Cells[j][i]=IntToStr(random(21)-10);
}
//---------------------------------------------------------------
// Кнопка "Розв'язок"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ Element *first=0,*last=0, *ce;
  int i,j,k=0,sum=0;
  for(i=0; i<4;i++)
  for(j=0; j<5; j++)
   if(abs(StrToInt(StringGrid1->Cells[j][i]))<=5)
     { if (first==0)
        { first=fir(StrToInt(StringGrid1->Cells[j][i]));
          last=first;
        }
        else
          add(first,&last,StrToInt(StringGrid1->Cells[j][i]));
     }
  setka(first, StringGrid2);   // Виведення списку у StringGrid2
  Edit1->Text=FormatFloat("0.00",sr_ar(first));
  vyluch(&first, &last);       // Вилучення елементів зі списку
  setka(first, StringGrid3);   // Виведення списку до StringGrid3
  int x=StrToInt(Edit2->Text);
  int y=StrToInt(Edit3->Text);
  ce=first;
  while(ce!=0)
  { if(ce->d==y)
      vstavka(ce, &last, x);   // Вставлення нового елемента до списку
    ce=ce->next;
  }
  setka(first, StringGrid4);   // Виведення списку до StringGrid4
  ochistka(first);
}
```

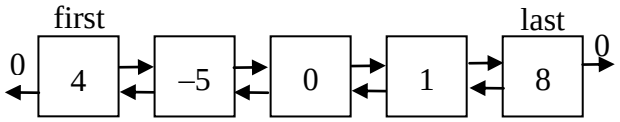
13.5 Різновиди списків

За кількістю зв'язків між елементами списки бувають:
- ✓ однозв'язні;
- ✓ двозв'язні.

За топологією списки можуть бути:
- ✓ лінійні;
- ✓ циклічні.

Список, який було розглянуто у цьому розділі, є *лінійний однозв'язний* (однонапрямлений). У такому списку кожний попередній елемент посилається на наступний, і за таким списком можна рухатися лише у напрямку від першого елемента до останнього (лінійно). Інакше кажучи, елементи лінійного списку і зв'язки поміж ними можна розташувати у пряму лінію, окрім того, зв'язки поміж елементами мають чіткий і незмінний напрямок.

Інколи виникає потреба проходити за списком в обох напрямках і "знати" не лише наступний елемент, а й попередній. У такому разі до елемента слід долучити ще одне поле – адресу попереднього елемента, тобто кожний елемент матиме два зв'язки з сусідніми елементами. На рисунку це позначається двома стрілками: вліво (зв'язок елемента з попереднім) і вправо (зв'язок елемента з наступним). Такий список називається *лінійним двозв'язним* (двонапрямленим), оскільки зв'язки мають два напрямки: від першого елемента до останнього і від останнього до першого. Схематично зобразити цей список можна так:



Кожний елемент, окрім поля з даними, має два додаткових поля, які містять адреси попереднього й наступного елементів. Унаслідок цього виникає можливість проходження за списком у якому завгодно напрямку.

Оголошення лінійного двозв'язного списку має вигляд:

```
struct Element
{ int d;  Element* next;  Element* prev;  };
```

Функція створення першого елемента масиву:

```
void fir(int x)
{ first=new Element;
  first->d=x;
  first->next=0;
  first->prev=0;
  last=first;
}
```

Функція долучення елемента до списку після останнього елемента:

```
void add_end(int x)
{ Element* c=new Element;
  c->d=x;
  c->next=0;
  c->prev=last;
```

```
  last->next=c;
  last=c;
}
```

Функція долучення елемента до списку перед першим елементом:

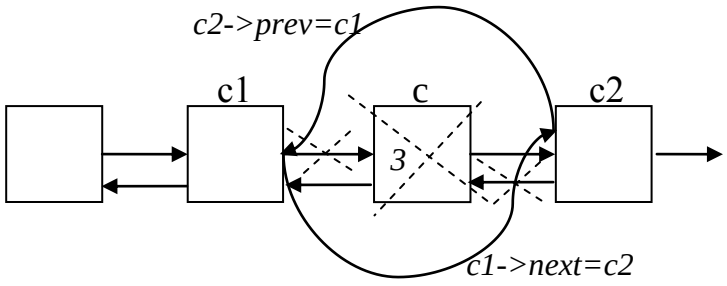
```
void add_beg(int x)
{ Element* c=new Element;
  c->d=x;
  c->prev=0;
  c->next=first;
  first->prev=c;
  first=c;
}
```

Функція вставлення нового елемента зі значенням `x` після елемента `c1`:

```
void insert(Element *c1, int x)
{ Element *c=new Element, *c2=c1->next;
  c->d=x; c->next=0; c->prev=0;
  c->prev=c1;
  c1->next=c;
  if(c2!=0){ c->next=c2;  c2->prev=c; }
  else last=c;
}
```

Функція вилучення елемента `c`:

```
void del_el(Element* c)
{ Element* c1,*c2;
  c1=c->prev;
  c2=c->next;
  if(c1!=0) c1->next=c2;
  else first=c2;
  if(c2!=0) c2->prev=c1;
  else last=c1;
  delete c;
}
```

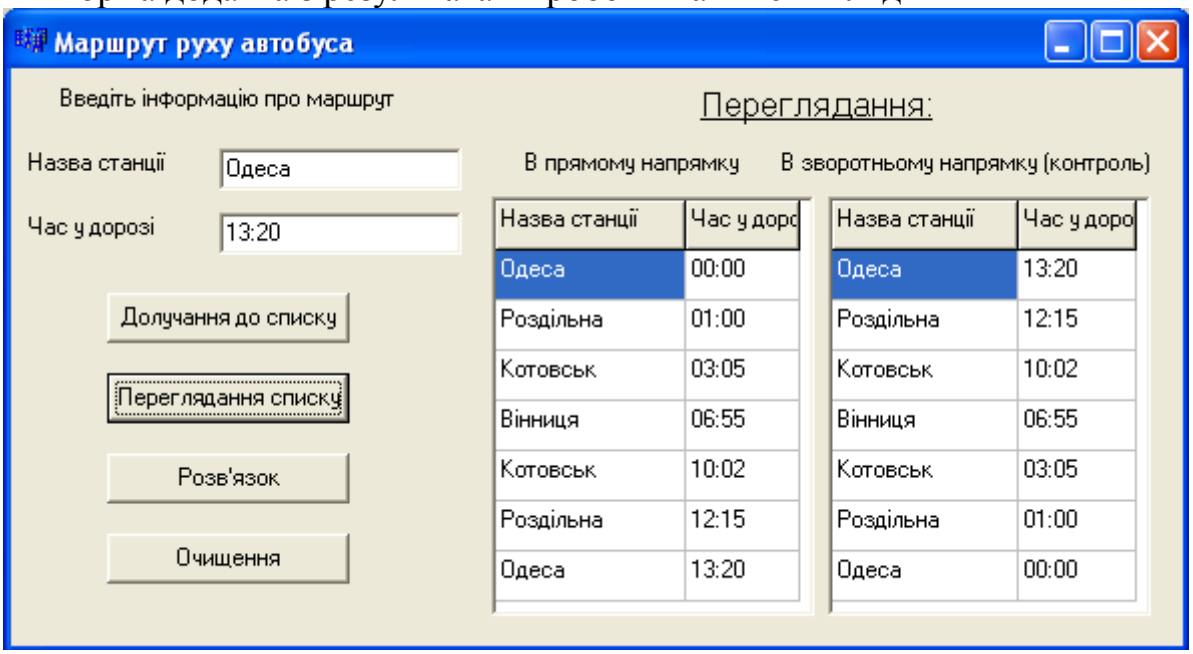


Приклад 13.7 Створити список з інформацією про маршрут руху автобуса: назва станції й час, який минув від виїзду з кінцевого пункту до приїзду на зазначену станцію. Перевірити, чи є у списку інформація про зворотний рух (назва першої станції збігається з назвою останньої, назва другої – з назвою передостанньої тощо). Визначити назву кінцевої станції та час у дорозі.

Розв'язок. Оскільки для порівняння назв станцій треба буде йти за списком водночас з обох кінців, список має бути *двонапрямленим лінійним*.

З умови завдання невідомо, чи уведено станції до списку за зростанням часу, який минув від моменту відправлення. Тому при створюванні списку слід долучати кожний новий елемент так, щоб наприкінці введення список був відсортованим за зростанням часу руху.

Форма додатка з результатами роботи матиме вигляд



Текст програми:

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
  StringGrid1->RowCount=2;
  StringGrid1->Cells[0][0]="Назва станції";
  StringGrid1->Cells[1][0]="Час у дорозі";
  StringGrid2->RowCount=2;
  StringGrid2->Cells[0][0]="Назва станції";
  StringGrid2->Cells[1][0]="Час у дорозі";
}
//---------------------------------------------------------
//Оголошення елемента списку
struct Element
{ char naz[20];   char tim[10];
  Element* next;  Element* prev;
};
Element* first=0, *last=0;
//Створення першого елемента. Параметри функції – рядки: назва станції та час
void fir(char *naz, char* tim)
{ first=new Element;
  strcpy(first->naz,naz);
  strcpy(first->tim,tim);
  first->next=0;  first->prev=0;
  last=first;
}
```

```
// Долучення елемента до початку списку.
// Параметри функції – рядки: назва станції та час
void add_beg(char *naz, char* tim)
{ Element* c=new Element;
  strcpy(c->naz,naz);
  strcpy(c->tim,tim);
  c->prev=0;
  c->next=first;
  first->prev=c;
  first=c;
}
// Долучення елемента до кінця списку.
// Параметри функції – рядки: назва станції та час
void add_end(char *naz, char* tim)
{ Element* c=new Element;
  strcpy(c->naz,naz);
  strcpy(c->tim,tim);
  c->next=0;
  c->prev=last;
  last->next=c;
  last=c;
}
// Виведення елементів списку з першого до останнього.
// Параметр функції – компонент StringGrid
void print_beg(TStringGrid*sg)
{ int i=1; sg->RowCount=i+1;
  Element* c=first;
  if(first==0) ShowMessage("Порожній");
  while(c!=0)
  { sg->RowCount=i+1;
    sg->Cells[0][i]=AnsiString(c->naz);
    sg->Cells[1][i]=AnsiString(c->tim);
    c=c->next; i++;
  }
}
// Виведення елементів списку з останнього до першого.
// Параметр функції – компонент StringGrid
void print_end(TStringGrid*sg)
{ int i=1; sg->RowCount=i+1;
  Element* c=last;
  if (last==0) ShowMessage("Empty2");
  while(c!=0)
  { sg->RowCount=i+1;
    sg->Cells[0][i]=AnsiString(c->naz);
    sg->Cells[1][i]=AnsiString(c->tim);
    c=c->prev;  i++;
  }
}
```

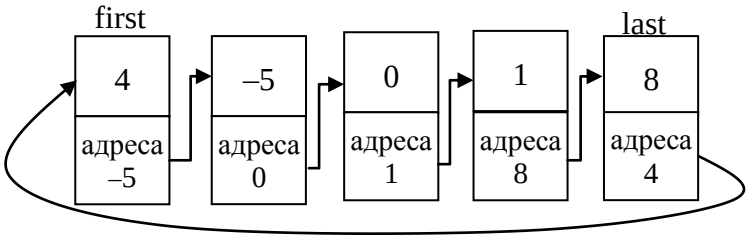
```
//Вставлення нового елемента після елемента c1
void insert(Element *c1, char *naz, char* tim)
{ Element *c=new Element, *c2=c1->next;
  strcpy(c->naz,naz);     // Поля даних нового елемента
  strcpy(c->tim,tim);
  c->next=0;  c->prev=0;
  c->prev=c1;         // Зв'язування нового елемента з попереднім елементом
  c1->next=c;
  if(c2!=0)               // Якщо є наступний елемент,
  { c->next=c2;           // новий елемент зв'язується з наступним,
    c2->prev=c;
  }
  else last=c;            // інакше новий елемент стає останнім.
}
// Вилучення елемента c
void del_el(Element* c)
{ Element* c1,*c2;
  c1=c->prev;             // Попередній елемент
  c2=c->next;             // Наступний елемент
  if(c1!=0)               // Якщо є попередній елемент,
     c1->next=c2;         // він зв'язується з  наступним елементом,
  else first=c2;          // інакше наступний стає першим.
  if(c2!=0)               // Якщо є наступний елемент,
     c2->prev=c1;         // попередній елемент зв'язується з наступним,
  else last=c1;           // інакше попередній елемент стає останнім.
  delete c;               // Звільнення пам'яті
}
// Вставлення нового елемента з сортуванням за часом
void sort(char *naz, char* tim)
{ Element *c=first;
  TTime tm=TTime(tim);     // Час у елементі, який вставляється.
  if(tm<TTime(first->tim))// Якщо час нового елемента менше за час першого,
  { add_beg(naz, tim);     // новий елемент вставляється перед першим
    return;  }
  // Проходження за списком, допоки час tm є більше за час елементів
  // (відшукування місця для вставлення)
  while(c->next!=0 && tm>TTime(c->next->tim)) c=c->next;
  if(c->next==0)          // Якщо елемент  c є останнім у списку,
    add_end(naz, tim);    // новий елемент вставляється після останнього,
  else                    // інакше новий елемент
    insert(c,naz,tim);    // вставляється після знайденого елемента
}
// Очищення списку з кінця
void ochistka()
{ Element* c;
  while(last!=0)  { c=last;  last=last->prev; delete c;  }
  first=0;
}
```

```
// Кнопка "Долучення до списку"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ if(first==0)
    fir(Edit1->Text.c_str(), Edit2->Text.c_str());
  else
    sort(Edit1->Text.c_str(), Edit2->Text.c_str());
}
//---------------------------------------------------
// Кнопка "Переглядання списку"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ print_beg(StringGrid1);
  print_end(StringGrid2);
}
//---------------------------------------------------
// Кнопка "Розв'язок"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ Element* c1=first;   Element* c2=last;
  bool ok=true;
  if(strcmp(c1->naz, c2->naz)==0)
    ShowMessage("Повернулися до початкової станції.
                 Час руху=" + AnsiString(c2->tim));
  else
  { ShowMessage("Не повернулися до початкової станції");
    return;
  }
  // Проходження за списком з обох кінців і порівнювання станцій
  while(TTime(c1->tim)<=TTime(c2->tim))
  { if(strcmp(c1->naz, c2->naz)!=0)
    { ok=false;
      ShowMessage("Зворотний маршрут є інший");
      break;
    }
    c1=c1->next;
    c2=c2->prev;
  }
  if(ok) ShowMessage("Зворотний маршрут є такий самий");
}
//---------------------------------------------------
// Кнопка "Очищення"
void __fastcall TForm1::Button4Click(TObject *Sender)
{ ochistka();
  Edit1->Clear();  Edit2->Clear();
  for (int i=1; i<StringGrid1->RowCount; i++)
  for (int j=0; j<2; j++)
    StringGrid1->Cells[j][i]="";
  for (int i=1; i<StringGrid2->RowCount; i++)
  for (int j=0; j<2; j++)
    StringGrid2->Cells[j][i]="";
}
```

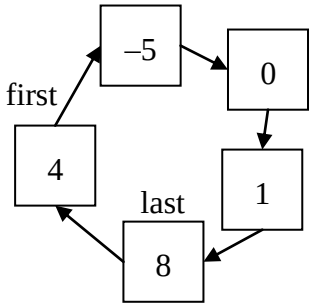
У *циклічному* однозв'язному списку за останнім елементом знов іде перший елемент. Це означає, що у полі `next` останнього елемента записано адресу першого елемента, тобто

```
last->next=first;
```

Схематично такий список можна зобразити як



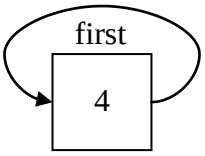
чи як



Оголошення циклічного однозв'язного списку буде таким самим, як оголошення лінійного однозв'язного списку.

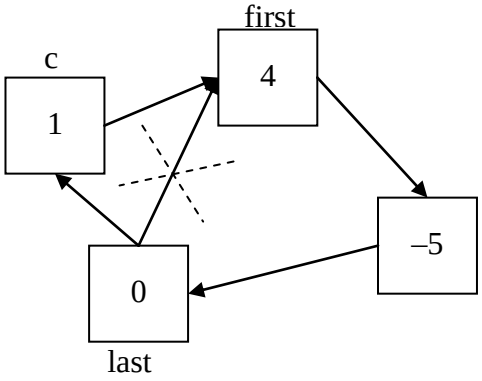
Функція створювання першого елемента:

```
void fir(int x)
{ first=new Element;
  first->d=x;
  first->next=first;
  last=first;
}
```



Функція долучення нового елемента до списку (між останнім і першим):

```
void add_el(int x)
{ Element* c=new Element;
  c->d=x;
  c->next=first;
  last->next=c;
  last=c;
}
```



Функція виведення списку до `Memo`:

```
void print_el(TMemo* memo)
{ Element* c=first;
  if(first==0) {ShowMessage ("Empty"); return;}
  do{
    memo->Lines->Add(IntToStr(c->d));
    c=c->next;
  }while(c!=first); // Допоки не дістались першого елемента
}
```

Функція вставлення нового елемента після елемента c1:

```
void insert_el(Element* c1, int x)
{ Element* c=new Element;
  c->d=x;
  c->next=c1->next;
  c1->next=c;
  if (c1==last) last=c;
}
```

Функція вилучення зі списку елемента, який слідує після елемента c1:

```
void del_el(Element* c1)
{ Element* c=c1->next;
  c1->next=c->next;
  if(c==first)              // Якщо вилучається перший елемент,
    first=c->next;          // першим стає другий
  if(c==last)               // Якщо вилучається останній елемент,
    last=c1;                // останнім стає передостанній
  delete c;
}
```

Функція звільнення пам'яті від списку:

```
void ochistka()
{ Element* c;
  last->next=0;
  while(first!=0)
  { c=first;
    first=first->next;
    delete c;
  }
}
```

Приклад 13.8 Створити список з іменами хлопчиків та дівчаток. Створити файл з лічилками. Обрати у довільний спосіб одну з лічилок і полічити дітей. Вивести до Memo імена дітей з відповідним словом лічилки. Вилучити зі списку ім'я того, хто вибув.

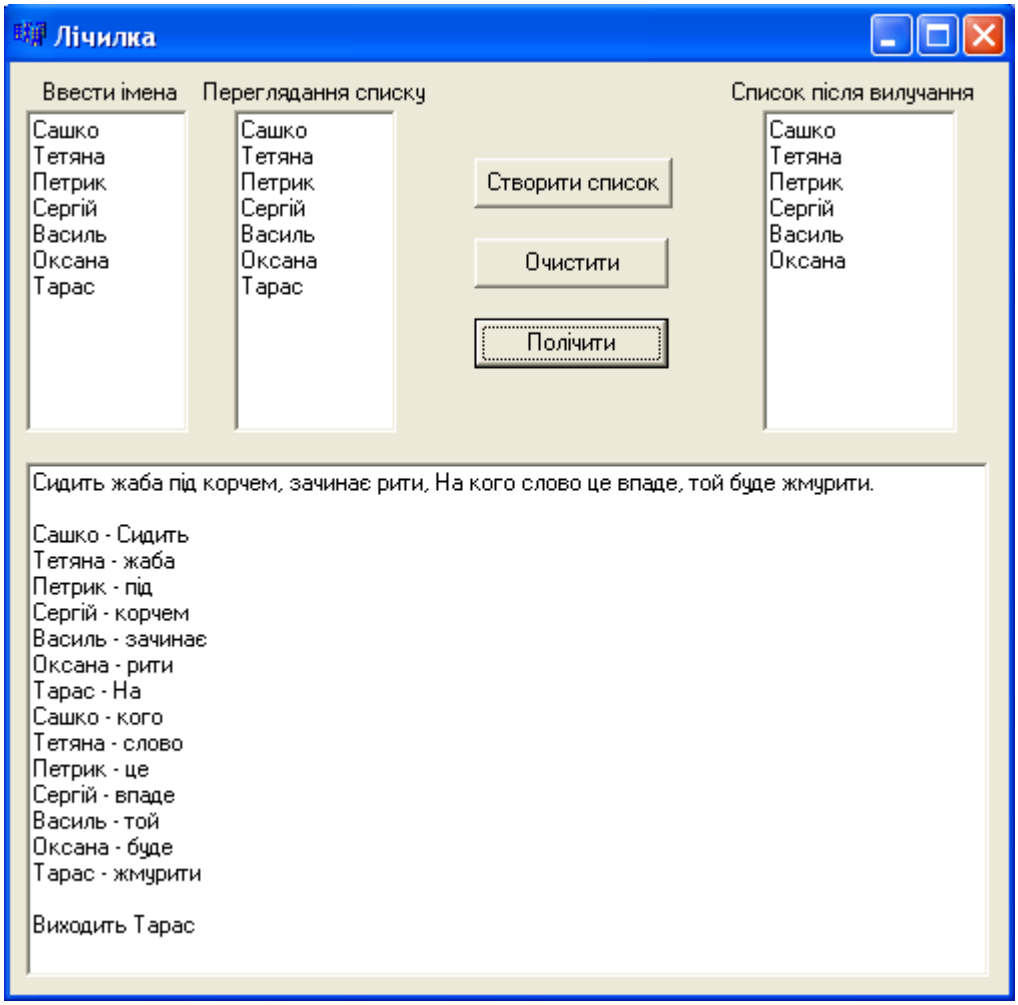
Розв'язок. Оскільки імена дітей під час лічби змінюватимуться циклічно, то список слід обрати однозв'язний циклічний.

Лічилки запишемо до файла lichilki.txt заздалегідь. У програмі буде згенеровано випадкове число і рядок з цим номером буде зчитано з файла. Саме цей рядок і буде обраною лічилкою. Рядок слід поділити на слова (це можна зробити, приміром, за допомогою функції strtok).

Уміст файла lichilki.txt, створеного у Блокноті:

```
Ходила квочка коло кілочка. Водила діточок біля квіточок! Квок!
Сітка, вітка, дуб, дубки – поставали козаки. Шабельками брязь – вийди, князь.
Сидить жаба під корчем, зачинає рити. На кого слово це впаде, той буде жмурити.
Еники-беники їли вареники. Еники-беники, квас: вийшов маленький Тарас.
```

Форма додатка з результатами роботи матиме вигляд



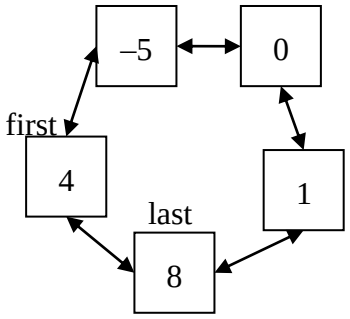
Текст програми:

```
FILE* f;
char fn[]="lichilki.txt";        // Рядок з назвою файла
//Оголошення елемента списку
struct Element
{ char name[15];
  Element* next;
} *first=0, *last=0;
// Функція створювання першого елемента. Параметр – ім'я (рядок)
void fir(char * x)
{ first=new Element;
  strcpy(first->name, x);
  first->next=first;
  last=first;
}
// Функція долучення елемента до списку.  Параметр – ім'я (рядок)
void add_el(char* x)
{ Element* c=new Element;
  strcpy(c->name,x);
  c->next=first;
  last->next=c;
```

```
  last=c;
}
// Функція виведення списку до компонента memo
void print_el(TMemo* memo)
{ memo->Clear();
  Element* c=first;
  if (first==0) {ShowMessage("Empty"); return;}
  do{
     memo->Lines->Add(AnsiString(c->name));
     c=c->next;
    } while(c!=first);
}
// Функція вилучення елемента, який слідує після елемента c1
void del_el(Element* c1)
{ Element* c=c1->next;
  c1->next=c->next;
  if(c==first) first=c->next;
  if(c==last) last=c1;
  delete c;
}
// Функція звільнення пам'яті й очищення форми
void ochistka()
{ Element* c;last->next=0;
  while(first!=0)
  { c=first;
    first=first->next;
    delete c;
  }
}
// Кнопка "Створити список"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, N=Memo1->Lines->Count;
  for(i=0; i<N; i++)
    if(first==0) fir(Memo1->Lines->Strings[i].c_str());
    else add_el(Memo1->Lines->Strings[i].c_str());
  print_el(Memo2);
}
// Кнопка "Очистити"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ ochistka();
  print_el(Memo4);
}
// Кнопка "Полічити"
void __fastcall TForm1::Button3Click(TObject *Sender)
{ randomize();
  int n=random(4);   // Генерування випадкового числа з проміжку [0,3]
  int i=0;   char s[200];
  if((f=fopen(fn, "rt"))==0)
    {ShowMessage("Can't open file"); return;}
```

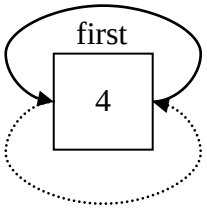
```
  while(i<=n && fgets(s,200,f))
   i++;    // Зчитування рядків з файла, допоки не зчитано рядок з номером n
  fclose(f);
  if (s[strlen(s)-1]=='\n')
    s[strlen(s)-1]='\0';
  Memo4->Lines->Add(AnsiString(s));
  Memo4->Lines->Add("");
  char* D=new char[10];
  strcpy(D," .,-!?:");    // Символи-розділювачі
  char* p=new char [50];
  p=strtok(s,D);
  Element* c=first;
  while(p!=0)
  { Memo4->Lines->Add(AnsiString(c->name)+" - "+AnsiString(p));
    p=strtok(NULL, D);    // Виокремлення наступного слова з рядка
    c=c->next;            // і перехід до наступного імені дитини
  }
  delete []p; delete []D;
  // Пошук елемента списку, після якого треба вилучити елемент.
  Element* c1=first;
  while (c1->next->next!=c)  c1=c1->next;
  Memo4->Lines->Add("");
  Memo4->Lines->Add("Виходить "+AnsiString(c1->next->name));
  del_el(c1);             // Вилучення обраного лічилкою імені
  print_el(Memo3);
}
```

Циклічний двозв'язний (двонапрямлений) список – це циклічний список, кожний елемент якого зберігає вказівники на наступний і попередній елементи. Схематичне зображення такого списку:



Оголошення циклічного двозв'язного списку буде таким самим, як оголошення лінійного двозв'язного списку.

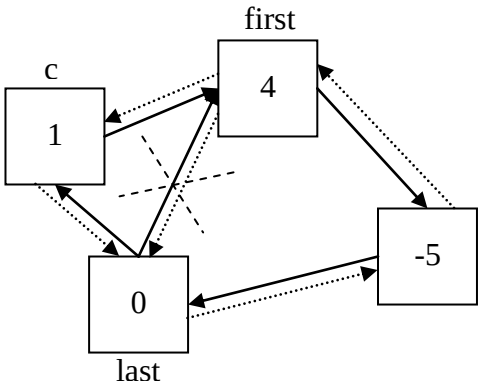
Функція створювання першого елемента:




```
void fir(int x)
{ first=new Element;
  first->d=x;
  first->next=first;
  first->prev=first;
  last=first;
}
```

Функція долучення нового елемента до списку (поміж останнім і першим):

```
void add_el(int x)
{ Element* c=new Element;
  c->d=x;
  c->next=first;
  c->prev=last;
  last->next=c;
  first->prev=c;
  last=c;
}
```



Функція виведення списку до Memo від першого елемента до останнього:

```
void print_beg(TMemo* memo)
{ memo->Clear();
  Element* c=first;
  if(first==0) {ShowMessage ("Порожній"); return;}
  do{ memo->Lines->Add(IntToStr(c->d));
     c=c->next;
    }while(c!=first);
}
```

Функція виведення списку до Memo від останнього елемента до першого:

```
void print_end(TMemo* memo)
{ memo->Clear();
  Element* c=last;
  if(last==0) { ShowMessage ("Порожній"); return;}
  do{ memo->Lines->Add(IntToStr(c->d));
      c=c->prev;
    } while(c!=last);
}
```

Функція вставлення нового елемента після елемента c1:

```
void insert_el(Element* c1, int x)
{ Element* c=new Element;
  c->d=x;
  c->next=c1->next;
  c1->next->prev=c;
  c1->next=c;
  c->prev=c1;
  if(c1==last) last=c;
}
```

Функція вилучення зі списку елемента, який слідує після елемента c1:

```
void del_el(Element* c1)
{ Element* c=c1->next;
  c1->next=c->next;
  c->next->prev=c1;
  if(c==first) first=c->next;
  if(c==last)  last=c1;
```

```
  delete c;
}
```

Функція звільнення пам’яті від списку:

```
void ochistka()
{ Element* c;
  last->next=0;
  first->prev=0;
  while(first!=0)
  { c=first;
    first=first->next;
    delete c;
  }
  last=0;
}
```

13.6 Бінарні дерева

Бінарне дерево – це динамічна структура даних, яка складається з *вузлів*, кожен з яких містить, окрім даних, не більше двох посилань на різні бінарні дерева. На кожний вузол є лише одне посилання. Початковий вузол називається *коренем дерева*. Схематичне зображення бінарного дерева наведено на рис. 13.1.

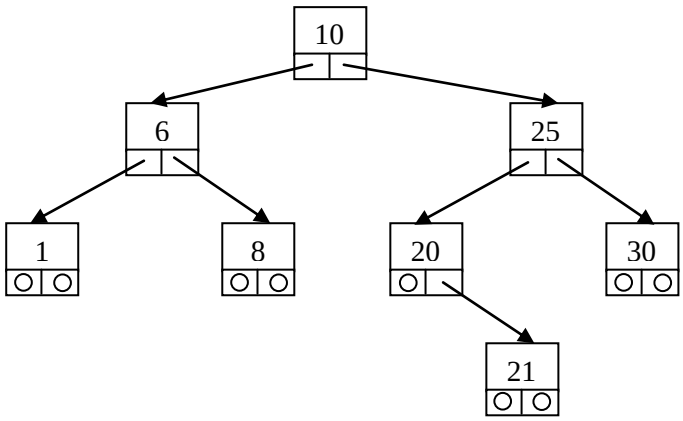


Рис. 13.1. Приклад зображення бінарного дерева

В оперативній пам’яті комірки може бути розташовано лінійно за зростанням адрес чи то хаотично, а дерево – лише метод логічної організації даних.

Вузол, який не має піддерев, називається *листом*. Вузол зі значенням 6 є *предком* для вузлів 1 та 8, а вузли 1 та 8 є *нащадками* вузла 6. *Висота* дерева визначається кількістю рівнів, на яких розташовуються його вузли. Значення вузла дерева називається *ключем*.

Оголошення вузла дерева з цілих чисел:

```
struct Node
{ int d;          // ключ – числове поле
  Node *left;     // вказівник на ліве піддерево
  Node *right;    // вказівник на праве піддерево
};
```

Якщо для кожного вузла дерева всі ключі його лівого піддерева є менше за ключ цього вузла, а всі ключі його правого піддерева – більше, то воно називається *деревом пошуку*. У такому дереві однакові ключі заборонено. У дереві пошуку можна віднайти елемент за ключем, рухаючись від кореня й переходячи на ліве чи то праве піддерево залежно від значення ключа в кожному вузлі.

Дерево є *рекурсивною* структурою даних, оскільки кожне піддерево також є деревом. Дії з такими структурами найоптимальніш описуються за допомогою рекурсивних алгоритмів. Приміром, функцію обходу всіх вузлів дерева у загальному вигляді можна описати як

```
void way_around (<дерево>)
{ way_around (<ліве піддерево>)
  <відвідування кореня>
  way_around (<праве піддерево>)
}
```

Можна обходити дерево й у іншому порядку, наприклад, спочатку корінь, потім піддерева, але наведена функція дозволяє одержати на виході відсортовану послідовність ключів, оскільки спочатку відвідуються вершини з меншими ключами, розташовані у лівому піддереві. Результат обходу дерева, зображеного на рис. 13.1:

```
1, 6, 8, 10, 20, 21, 25, 30
```

Якщо у функції обходу перше звертання іде до правого піддерева, результат обходу буде іншим:

```
30, 25, 21, 20, 10, 8, 6, 1
```

Отже, дерева пошуку можна застосовувати для швидкого *сортування* та *пошуку* значень. При обході дерева вузли не вилучаються.

Для бінарних дерев визначені такі операції:

- ✓ вилучення вузла до дерева;
- ✓ пошук по дереву;
- ✓ обхід дерева;
- ✓ вилучення вузла.

Для кожного рекурсивного алгоритму можна створити його нерекурсивний еквівалент.

Приклад 13.9 Увести цілі числа до `Memo` і сформувати з них дерево пошуку.

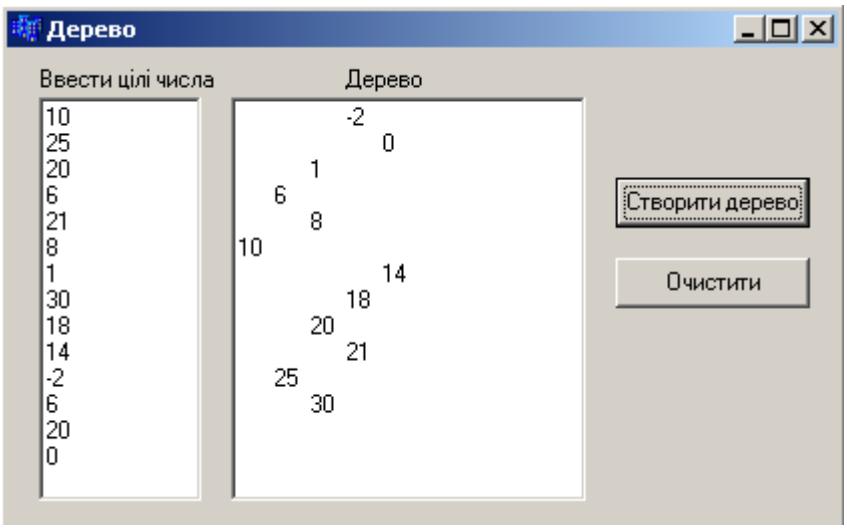
Розв'язок. У програмі реалізуємо нерекурсивну функцію пошуку по дереву із включенням і рекурсивну функцію обходу дерева.

Перша функція здійснює пошук елемента із заданим ключем. Якщо елемент знайдено, вона повертає вказівник на нього (елементи з однаковими ключами у дереві пошуку заборонено), а якщо елемента немає – включає його у відповідне місце дерева і повертає вказівник на нього. Для включення елемента слід пам'ятати пройдений деревом шлях на один крок назад і знати, чи виконується включення нового елемента в ліве чи праве піддерево його предка.

У функції `search_insert()` поточний вказівник для пошуку по дереву позначений `pv`, вказівник на предка `pv` позначений `prev`, змінна `pnew` використовується для виділення пам'яті під вузол, який включається в дерево. Рекурсії вдалося уникнути, зберігши всього одну змінну `prev` і повторивши при включенні оператори, які визначають, до якого піддерева долучається новий вузол.

У функції `print_tree()` другим параметром передається ціла змінна, яка визначає, на якому рівні розміщено вузол. Корінь перебуває на рівні 0. Дерево виводиться на екран по горизонталі в такий спосіб, щоб корінь містився ліворуч (дерево повернути набік). Перед значенням вузла для імітації структури дерева виводиться кількість пробілів, пропорційна до рівня вузла. Якщо закоментувати цикл виведення пробілів, відсортований за зростанням масив буде виведено у стовпчик.

Форма додатка з результатами роботи матиме вигляд



Текст програми:

```
struct Node
{ int d;
  Node *left;
  Node *right;
};
Node* root=0;
// Створення першого вузла (кореня) дерева
Node * first(int x)
{ Node *pv = new Node;
  pv->d= x;
  pv->left = 0;
  pv->right = 0;
  return pv;
}
// Пошук із долученням – вставлення нового елемента у дерево з перевіркою на збіг
Node * search_insert(Node *root, int x)
{ Node *pv = root, *prev;
  bool found=false; //Ознака: чи знайшли вузол зі значенням x (спочатку false)
```

```
  // Допоки в дереві є елементи і елемент зі значенням x не знайдено
  while(pv && !found)
  { prev = pv;
    if(x==pv->d) found = true;
    else                    // Якщо значення вузла не збігається з x
     if(x<pv->d)            // і, якщо x є менше за значення вузла,
       pv = pv->left;       // перехід до лівого піддерева,
     else pv = pv->right;// або перехід до правого піддерева.
  }
  if(found)           // Якщо вузол зі значенням x  було знайдено,
    return pv;        // повертання вказівника на нього
  Node *pnew = new Node;   // Створення нового вузла
  pnew->d = x;
  pnew->left = 0;
  pnew->right = 0;
  if(x<prev->d)
    prev->left = pnew;    // Долучення до лівого піддерева предка
  else
    prev->right = pnew;   //Долучення до правого піддерева предка
  return pnew;            //Повертання вказівника на новий вузол
}
// Виведення дерева – рекурсивна функція
void print_tree(Node *p, int level)
{ AnsiString S="";
  if(p)
  { print_tree(p->left, level+1);    //Виведення лівого піддерева
    for(int i=0; i<level; i++)  S=S+"      ";
    S=S+IntToStr(p->d);
    Form1->Memo2->Lines->Add(S);     //Виведення кореня піддерева
    print_tree(p->right, level +1); //Виведення правого піддерева
  }
}
void ochistka(Node* p)
{ if(p)
  { ochistka(p->left);
    ochistka(p->right);
    delete p;
  }
  else return;
}
// Кнопка "Створити дерево"
void __fastcall TForm1::Button1Click(TObject *Sender)
{int i, N=Memo1->Lines->Count;
 for(i=0; i<N; i++)
  if(root==0) root=first(StrToInt(Memo1->Lines->Strings[i]));
  else search_insert(root, StrToInt(Memo1->Lines->Strings[i]));
 print_tree(root, 0);
}
//---------------------------------------------------------
```

```
// Кнопка "Очистити"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ ochistka(root);
  Memo1->Clear();   Memo2->Clear();
}
```

Розглянемо рекурсивний аналог функції `search_insert()`. У функції число порівнюється зі значенням у корені. Якщо воно є менше, функція викликається для долучення цього значення до лівого піддерева, якщо – навпаки, функція викликається для долучення до правого піддерева. В результаті виконується спуск до порожнього піддерева, замість якого формується новий елемент і долучається до дерева.

```
Node * search_insert(Node *root, int x)
{ if(!root)                 //Створювання вузла, якщо дерева ще немає.
  { root = new Node;
    root->d=x;
    root->left=0;
    root->right=0;
  }
  else                      // Якщо дерево вже є,
    if(x < root->d)         // якщо нове значення є менше за значення вузла
                            // викликається функція від лівого піддерева
      root->left=search_insert(root->left, x);
    else
      // Викликається функція від правого піддерева
      root->right=search_insert(root->right, x);
  return root;
}
```

Наприклад, треба створити дерево з числами 10, 25, 20, 6, 21, 8, 1, 30 (див. рис. 13.1).

1) Викликаємо для 10:

```
search_insert(root, 10)
```

Кореня ще немає (`root=0`), тому створюємо корінь зі значенням 10.

2) Викликаємо для 25:

```
search_insert(root, 25)
```

Корінь є, тому порівнюємо 25 і 10. Оскільки 25>10, викликаємо цю ж саму функцію для правого піддерева:

```
search_insert(root->right, 25)
```

Правої гілки ще немає, тому створюємо новий елемент.

3) Викликаємо для 20:

```
search_insert(root, 20)
```

Корінь є, тому порівнюємо 20 і 10. Оскільки 20>10, викликаємо цю ж саму функцію для правого піддерева:

```
search_insert(root->right, 20)
```

Права гілка вже є, порівнюємо 20 і 25. 20<25, тому викликаємо функцію від лівого піддерева відносно елемента зі значенням 25:

```
search_insert(root->right->left, 20)
```

Створюємо новий елемент.

4) Викликаємо для 6:

```
search_insert(root, 6)
```

Корінь є, тому порівнюємо 6 і 10. Оскільки 6<10, викликаємо цю ж саму функцію для лівого піддерева:

```
search_insert(root->left, 6)
```

Створюємо новий елемент.

5) Викликаємо для 21:

```
search_insert(root, 21)
```

Корінь є, тому порівнюємо 21 і 10. Оскільки 21>10, викликаємо цю ж саму функцію для правого піддерева:

```
search_insert(root->right, 21)
```

Права гілка вже є, тому порівнюємо 21 і 25. 21<25, тому викликаємо функцію від лівого піддерева відносно елемента зі значенням 25:

```
search_insert(root->right->left, 21)
```

Там вже є елемент зі значенням 20, тому порівнюємо 21 і 20. Оскільки 21>20, тому викликаємо функцію від правого піддерева відносно елемента зі значенням 20:

```
search_insert(root->right->left->right, 20)
```

Створюємо новий елемент.

Далі продовжуємо рекурсивно викликати функції від лівого й правого піддерев, допоки не дістанемось кінця гілки (вузол=0). Тоді створюємо новий вузол.

Функція пошуку елемента в дереві:

```
int Find(Node *root, int x)
{ if (!root) return 0;
  else if (root->d==x) return 1;
       else if (x < root->d) return Find(root->left, x);
         else return Find(root->right, x);
}
```

Вилучення вузла з дерева являє собою не таке просте завдання, оскільки вузол, що вилучається, може бути кореневим, містити два (рис. 13.2, в), одне (рис. 13.2, б) чи не містити посилання (рис. 13.2, а) на піддерева. Для вузлів, які містять менше двох посилань, вилучення є тривіальне. Щоб зберегти впорядкованість дерева при вилученні вузла із двома посиланнями, його замінюють на вузол з найближчим до нього ключем. Це може бути крайній лівий вузол його правого піддерева чи то крайній правий вузол лівого піддерева (наприклад, щоб вилучити з дерева на рис. 13.1 вузол із ключем 25, його треба замінити на 21 або 30, вузол 10 замінити на 20 або 8).

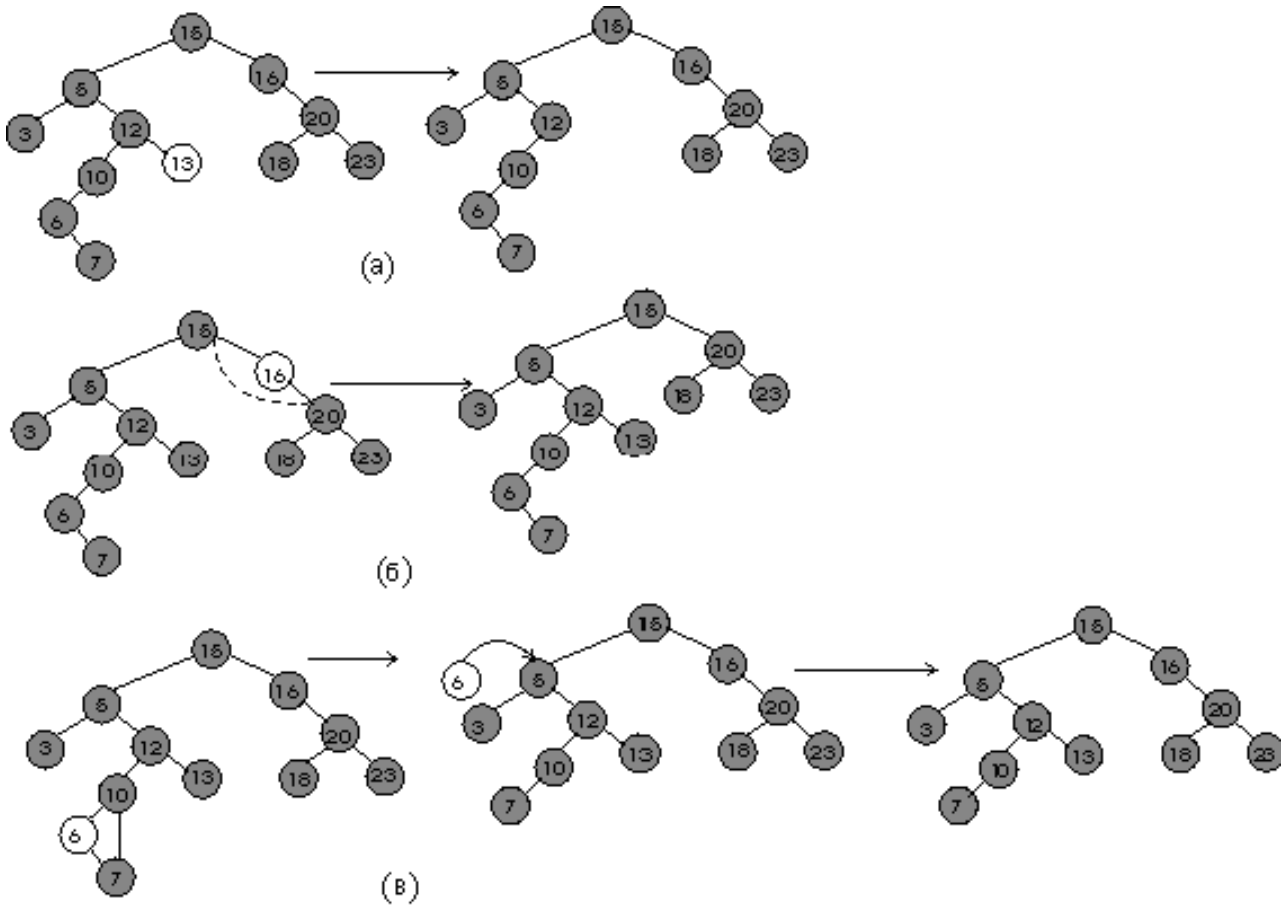


Рис. 13.2. Різні випадки вилучення вузлів дерева:

а) вилучення вузла без нащадків (листа зі значенням 13);

б) вилучення вузла зі значенням 16, в якого існує лише лівий або правий нащадок (одне посилання);

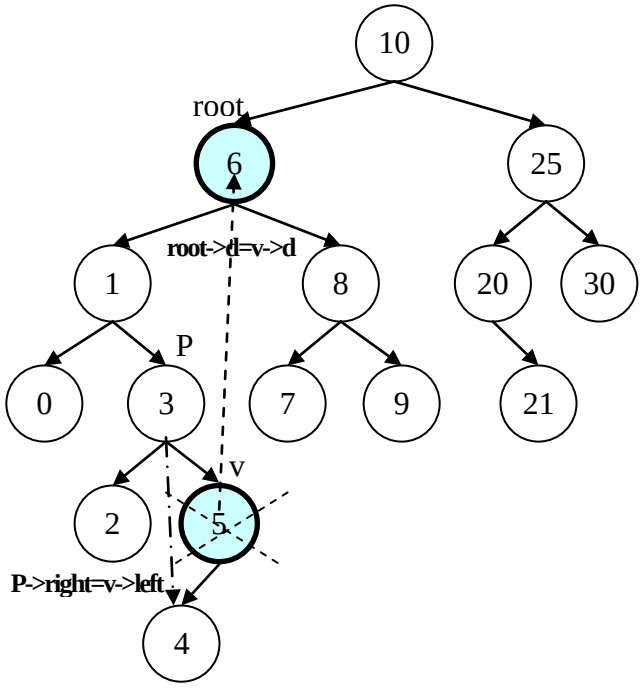
в) вилучення листа зі значенням 6, в якого існують обидва нащадки (два посилання).

Розглянемо докладно алгоритм функції `Del_el()`, яка вилучає вузол із заданим значенням з дерева. Відшукуємо вузол із заданим значенням. Якщо його буде віднайдено, функція вилучить його, або повідомить, що елемента немає. Якщо шукане значення є менше за значення вузла, викликаємо функцію від лівого піддерева, якщо більше – від правого піддерева. Якщо значення збігаються, функція запам'ятовує віднайдений вузол у вказівнику `P` і перевіряє наявну кількість посилань від цього вузла. Якщо посилання є лише одне (`root->left` чи `root->right` становить нуль), вузол, який підлягає вилученню, замінюється на існуючого лівого чи правого нащадка. Якщо посилання є два, переходимо до лівого піддерева і відшукуємо крайній правий його елемент. На цей елемент замінюємо вузол, який вилучається.

```
Node * Del_el(Node *root, int x)
{ Node* P, *v;
  if(!root) { ShowMessage("Елемента нема");  return 0; }
  if(x < root->d) root->left = Del_el(root->left, x);
  else
```

```
    if(x > root->d) root->right = Del_el(root->right, x);
    else
    { P = root;
      if (!root->right)          //Випадок 2: праве піддерево не існує,
        root = root->left;       //лівий елемент стає на місце root.
      else
        if (!root->left)         //Випадок 2: ліве піддерево не існує,
          root = root->right;    //правий елемент стає на місце root.
     else                        //Випадок 3: існують два піддерева.
        { v = root->left;        //Перехід на лівий вузол,
          while(v->right)        //проходження праворуч до кінця.
             {P = v; v = v->right;} //P – попередній вузол.
          root->d = v->d;        //Змінювання значення вузла root.
          if(P!=root)            //Якщо P дорівнює root, долучення праворуч
            P->right = v->left;      //елементів з лівого піддерева  v
          else P->left=v->left;      //або долучення їх ліворуч.
        }
//Звільнення пам'яті від крайнього правого елемента (на який замінено root)
     delete v;
   }
  return root;
}
```

На поданому нижче рисунку схематично зображено вилучення вузла зі значенням 6



Вузол `root` має два піддерева. Переходимо на ліве (`1`). Йдемо лівим піддеревом до крайнього правого елемента (`5`). Замінюємо значення `root` (`6`) на `5`. Перш ніж звільнити пам'ять від елемента зі значенням `5`, долучаємо його ліву гілку до попереднього елемента у якості правої гілки.

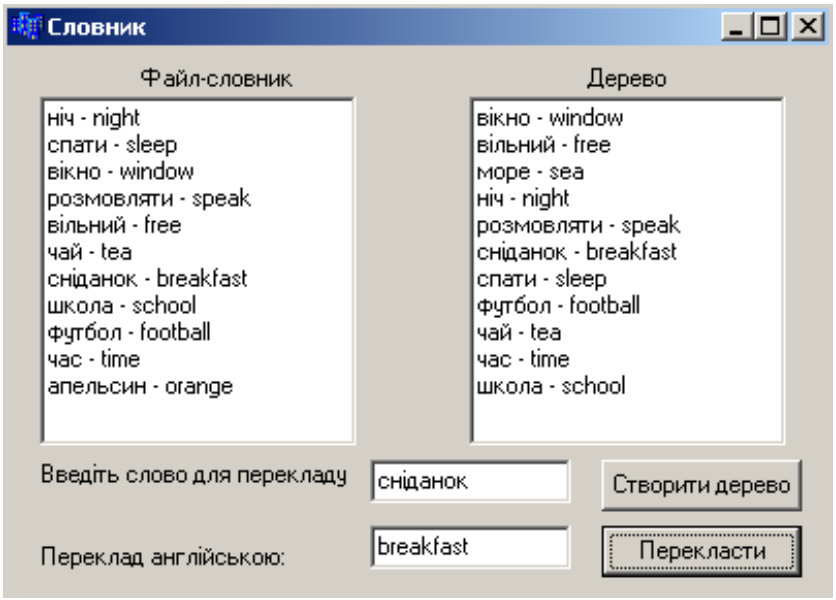
Приклад 13.10 Створити дерево – українсько-англійський словник. Надати можливість швидкого пошуку українських слів для перекладання.

Розв'язок. Кожен вузол дерева містить два інформаційні поля: український та англійський варіанти слова. Слова з перекладом заздалегідь запишемо у файлі `vocabulary.txt` і дерево пошуку створюватимемо за даними з файла. Домовимося, що у файлі кожне слово з перекладом займатиме окремий рядок. Український варіант буде розташовано на початку рядка, а англійський – після нього. Поміж словами будуть пробіл, тире і пробіл.

Наприклад, уміст файла `vocabulary.txt` може бути таким:

```
море - sea
ніч - night
спати - sleep
вікно - window
розмовляти - speak
вільний - free
чай - tea
сніданок - breakfast
школа - school
футбол - football
час - time
апельсин - orange
```

Форма додатка з результатами роботи матиме вигляд



Текст програми:

```
FILE* f;
char fn[]="vocabulary.txt";
char* v=0;            // Порожній рядок
struct Node
{ char ukr[20];       // Слово українською мовою
  char eng[20];       // Слово англійською мовою
  Node *left; Node *right; };
```

```
Node* root=0;
Node * first(char ukr[20], char eng[20])
{ Node *pv = new Node;
  strcpy(pv->ukr, ukr);
  strcpy(pv->eng, eng);
  pv->left = 0;
  pv->right = 0;
  return pv;
}
// Пошук з долученням
Node * search_insert(Node *root, char ukr[20], char eng[20])
{ if(!root)
  { root = new Node;
    strcpy(root->ukr, ukr);
    strcpy(root->eng, eng);
    root->left=0; root->right=0;
  }
  else
    if(strcmp(ukr, root->ukr)<0)
      root->left=search_insert(root->left, ukr, eng);
    else root->right=search_insert(root->right, ukr, eng);
  return root;
}
//Пошук слова українською мовою
char* Find(Node *root, char ukr[20])
{ if(!root) return v;
  else
    if(strcmp(root->ukr,ukr)==0) //Якщо слово віднайдено,
       return root->eng;         //повертання його англійського перекладу
    else
      if(strcmp(ukr,root->ukr)<0)
        return Find(root->left, ukr);
      else
        return Find(root->right, ukr);
}
// Виведення слів з дерева до Memo2 (слова відсортовано за алфавітом)
void print_tree(Node *p)
{ AnsiString S;
  if(p)
  { print_tree(p->left);
    S=AnsiString(p->ukr)+" - "+AnsiString(p->eng);
    Form1->Memo2->Lines->Add(S);
    print_tree(p->right);
   }
}
// Звільнення пам'яті від дерева
void ochistka(Node* p)
{ if(p)
  { ochistka(p->left);
```

```
    ochistka(p->right);
    delete p;
  }
}
// Кнопка "Створити дерево"
void __fastcall TForm1::Button1Click(TObject *Sender)
{ char s[50], ukr[21], eng[21];
  int i;
  if((f=fopen(fn,"rt"))==0)
  { ShowMessage("Неможливо відкрити файл"); return;}
  while(fgets(s, 50, f))
  { if(s[strlen(s)-1]=='\n') s[strlen(s)-1]='\0';
    Memo1->Lines->Add(AnsiString(s));
    for (i=0; i<strlen(s) && s[i]!=' '; i++);
    strncpy(ukr, s, i); ukr[i]='\0';  // Копіювання слова українською
    strcpy(eng, &s[i+3]);             // Копіювання слова англійською
    if(root==0)root = first(ukr, eng);// Долучення слова до дерева
    else search_insert(root, ukr, eng);
  }
  fclose(f);
  print_tree(root);
}
// Кнопка "Перекласти"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ char ukr[21], eng[21];
  strcpy(ukr, Edit1->Text.c_str());
  strcpy(eng,Find(root, ukr));
  if(strlen(eng)==0)ShowMessage("Слово у словнику є відсутнє");
  else Edit2->Text=AnsiString(eng);
}
```

13.7 Автомати

13.7.1 Поняття автомата

Автомат – це абстрактний об'єкт, призначений для виконання певної роботи. Саме теорія автоматів застосовується для шифрування текстів і в теорії алгоритмів. Також автомати використовуються для описування поведінки різноманітних систем керування, таких як цифрові схеми, протоколи обчислювальних систем тощо. Фактично за допомогою автомата можна реалізувати перекладання текстів з однієї мови на іншу. Певним продовженням ідеї автоматів є поняття найважливішої частини комп'ютера – процесора (це автомат з пам'яттю), який перетворює одну двійкову послідовність на іншу.

Кожен автомат отримує для опрацювання вхідну послідовність даних (сигналів), а як результат видає вихідну послідовність даних (сигналів).

Означення скінченого автомата (в англійській літературі використовується також поняття finite state mashine – машина скінчених станів) було наведено

1955 р. Джорджем Х. Мілі й 1956 р. Едвардом Ф. Муром. Автомати Мілі та Мура дещо відрізнялися один від одного, але ці відмінності є неістотні. Наведемо означення, більш подібне до автомата Мілі.

Автоматом називається п'ятірка M = (S, X, Y, F, s_0),

де S – непорожня скінчена множина, елементи якої (s_1, s_2, s_3, …) називаються станами автомата;

X – непорожня скінчена множина, яка називається вхідним алфавітом автомата;

Y – непорожня скінчена множина, яка називається вихідним алфавітом автомата;

F – відображення F: S × X → S × Y;

s_0 – початковий стан автомата.

Кожен автомат має скінчену множину станів. Автомат переходить з одного стану до іншого під впливом послідовності літер вхідного алфавіту. На момент запуску він перебуває у початковому стані s_0. На вхід автомата поступає перший елемент вхідної послідовності x_1, й відображення перетворює пару (s_0, x_1) до пари (s_1, y_1). Елемент s_1 стає новим станом автомата, а елемент y_1 – першим елементом вихідної послідовності. Отже, автомат перетворює певну вхідну послідовність у вихідну.

Алфавіт автомата – це скінчена множина певних символів. Наприклад, множина {1, 2} – це алфавіт, який складається з одиниці та двійки, множина {A, B, …, Z} – це алфавіт великих латинських літер.

Вхідним алфавітом є скінчена множина допустимих вхідних символів, з якої формуються зчитуванні автоматом рядки. *Вихідний алфавіт* – це скінчена множина допустимих вихідних символів, з якої формуються рядки-результати. Найпоширенішим є різновид автоматів, коли набори елементів вхідного й вихідного алфавітів автомата збігаються.

За приклад доволі простого автомата розглянемо ліфт у двоповерховому будинку. Припустімо, що цей ліфт вміє реагувати на такі *команди*:

- ✓ O – відчинити двері (від англ. open – відчинити);
- ✓ C – зачинити двері (від англ. close – зачинити);
- ✓ D – спуститися на перший поверх (від англ. down – донизу);
- ✓ U – піднятися до другого поверху (від англ. up – догори).

Окрім того, ліфт може виконувати комбінацію вище наведених команд, наприклад: "CUO", що означає: "зачинити двері", "піднятися до другого поверху", "відчинити двері". Але існують команди, які для такого ліфта будуть некоректними. Приміром, не можна рухатися з відчиненими дверима чи підніматися на другий поверх, коли ліфт уже перебуває на другому поверсі.

Тепер наведемо чотири можливі *стани* ліфта:

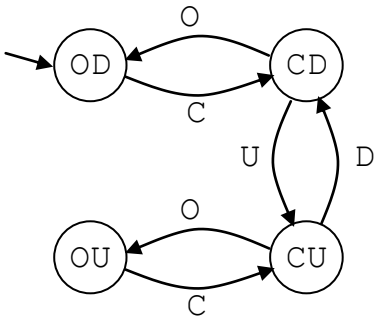
- ✓ CD – перший поверх, двері зачинено;
- ✓ OD – перший поверх, двері відчинено;
- ✓ CU – другий поверх, двері зачинено;
- ✓ OU – другий поверх, двері відчинено.

Вважатимемо, що спочатку ліфт (автомат) перебуває на першому поверсі, його двері є відчинено, тобто початковий стан ліфту є “CD”.

Завдання полягає у тому, щоб змоделювати такий ліфт на комп’ютері, тобто написати програму, спроможну виконувати команди ліфта (чи то повідомляти, що команда є некоректна).

Кожна з команд переводитиме ліфт із одного стану до іншого, приміром, якщо ліфт перебував у стані “CU” і надійшла команда “D”, то ліфт перейде до стану “CD”.

Переходи з одного стану до іншого можна описати графом переходів автомата, вершинами якого будуть стани автомата, а дугами – можливі переходи кожного такту роботи. Усі можливі команди і стани ліфта можна графічно зобразити так:



На цій схемі чотирма кружечками позначено стани автомата (початковим станом є стан “OD” – перший поверх, двері відчинено), дугами зі стрілками – переходи з одного стану до іншого, текстовим підписом біля дуг – команди, виконувані ліфтом. Наприклад, зі стану “OU” за команди “C” автомат перейде до стану “CU”.

Ідентичним до графічного поданням автомата є так звана “таблиця правил” (чи “таблиця переходів”):

Стан	**Команди**			
	O	**C**	**U**	**D**
CD	OD	–	CU	–
OD	–	CD	–	–
CU	OU	–	–	CD
OU	–	CU	–	–

Якщо ліфт перебуває у стані “CD” і надійшла команда “O”, то ліфт відчинить двері і, згідно до таблиці, перейде до стану “OD” (перший поверх, двері відчинено).

Отже розглянутий автомат здатен опрацьовувати команди: “O”, “C”, “U”, “D”. Ця множина формує алфавіт наведеного автомата, тобто набір можливих опрацьовуваних автоматом даних: {O, C, D, U}.

З алфавітних символів можна формувати рядки, приміром, “CUO” означає три послідовні команди: “зачинити двері”, “піднятися до другого поверху” і “відчинити двері”. Для виконання цих трьох команд слід застосувати автомат тричі. За першого застосування автомат стартує з початкового стану “OD”. Після

першого перетворення автомат змінить стан: зі стану "OD" дугою "C" здійсниться перехід до стану "CD". Після другого переходу зі стану "CD" за команди "U" автомат змінить свій стан на "CU". Після третього переходу за команди "O" автомат перейде до стану "OU".

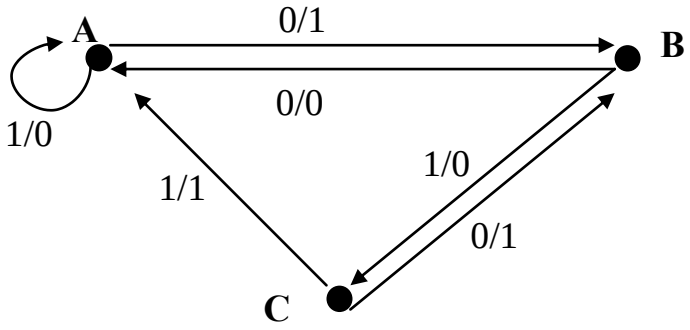
13.7.2 Синхронні автомати

Автомат називають *синхронним*, якщо він перетворює один елемент вхідного на один елемент вихідного алфавіту.

Програми, що реалізують синхронні автомати, можуть використовувати оператори циклу `for` для опрацювання послідовності елементів та вкладені в нього оператори `switch` та `if` для аналізу й перетворювання елементів.

Розглянемо на прикладі засоби програмування такого автомата.

Приклад 13.11 Автомат задано алфавітом X = Y ={0, 1} і трьома станами – A, B, C:



Ввести вхідну послідовність символів S з алфавітом X і здобути вихідну послідовність після дії заданого автомата (перетворений рядок S).

Розв'язок. По дугах автомата можна визначити, наприклад, що в стані A елемент 0 буде перетворено на 1 і при цьому автомат перейде до стану B, а елемент 1 перетвориться на 0 і автомат залишиться у тому ж самому стані A. Аналогічно інтерпретуються інші дуги. Цей автомат, заданий графічно, можна подати також у вигляді таблиці переходів:

Стан	Вхідний елемент	
	0	1
A	B,1	A,0
B	A,0	C,0
C	B,1	A,1

У наведеній програмі опрацювання вхідної послідовності реалізовано у функції. Зауважимо, що цю функцію можна реалізувати багатьма іншими способами. Стан автомата (A, B чи C) обирається послідовно для кожного з опрацьовуваних символів рядка за допомогою оператора `switch`.

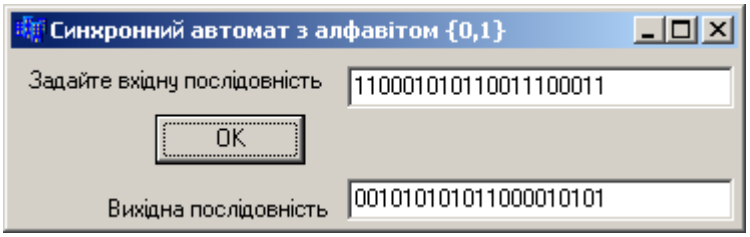
На вхід автомата подається рядок символів з вхідного алфавіту. Кожен з цих символів буде перетворено згідно до команд переходу, при цьому автомат змінюватиме свій стан. Розглянемо послідовно дії автомата у кожному стані.

Якщо автомат перебуває у стані A можливі дві ситуації: 1) при опрацю-

ванні символу '0' автомат перейде до стану B і видасть символ '1'; 2) при опрацюванні символу '1' автомат залишиться у стані A і видасть '0'.

У стані B при опрацюванні символу '0' автомат перейде до стану A без зміни символу, а символ '1' перетворить на '0' і перейде до стану C.

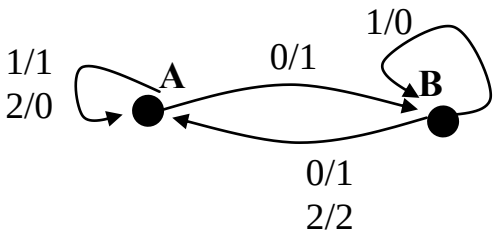
Стан C для обох можливих символів ('0' чи '1') як результат видаватиме '1' і змінюватиме свій стан: для символу '0' автомат перейде до стану B, а '1' – до стану A.



Текст функції та програми для командної кнопки:

```
AnsiString Automat1(AnsiString S)
{ // Для визначення станів автомата використовується перераховний тип даних
 enum conditions{A, B, C};
 conditions sit=A;                  // Початковий стан автомата
 for(int i=1;i<=S.Length();i++)
 switch (sit)
 {case A:{if (S[i]=='0') {S[i]='1'; sit=B;}else S[i]='0';break;}
  case B:{if (S[i]=='0') sit=A; else sit=C; S[i]='0';break;}
  case C:{if (S[i]=='0') sit=B; else sit=A; S[i]='1';break;}
 }
  return S;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString S=Edit1->Text;
  Edit2->Text=Automat1(S);    }
```

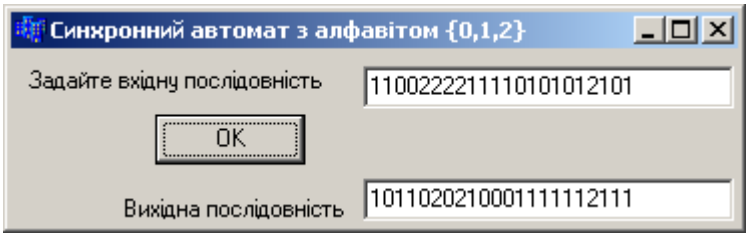
Приклад 13.12 Задано автомат алфавітом X = Y = {0, 1, 2} та двома станами – A та B. Ввести вхідну послідовність символів з алфавіту X й здобути вихідну послідовність після дії заданого автомата.



Розв'язок. Цей автомат, заданий графічно, можна подати також у вигляді таблиці переходів:

Стан	Вхідний елемент		
	0	1	2
A	B,1	A,1	A,0
B	A,1	B,0	A,2

Згідно до рисунка й таблиці, якщо автомат перебуває у стані A, за елементів 1, 2 він залишиться у тому ж стані, а за елемента 0 він перейде до стану B.



Текст функції та програми для командної кнопки:

```
AnsiString Automat2(AnsiString s)
{ enum conditions {A,B,C};
  conditions sit=A;
  sit=A;
  for(int i=1; i<=s.Length();i++)
   switch (sit)
   { case A: { if(s[i]=='0') { sit=B; s[i]='1'; }
               else
                 if(s[i]=='2') s[i]='0';
               break;
             }
     case B: { if(s[i]=='1') s[i]='0';
               else
                { sit=A;
                  if(s[i]=='0') s[i]='1';
                }
               break;
             }
   }
  return s;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString s=Edit1->Text;
  Edit2->Text= Automat2(s);
}
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, char &Key)
{ // Користувач не зможе ввести символ вхідного алфавіту відмінний від 0, 1 чи 2
  if(Key<'0' || Key>'2') Key=0;
}
```

13.7.3 Асинхронні автомати

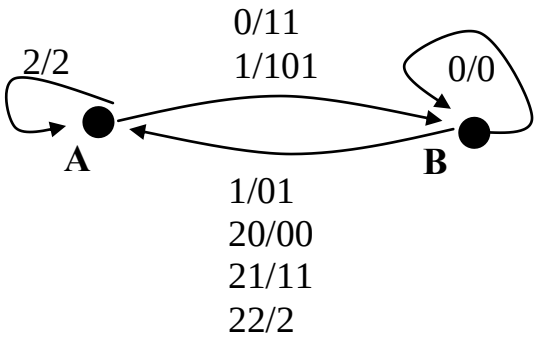
Автомат називається *асинхронним*, якщо вхідна послідовність може бути перетворена ним на послідовність іншої довжини, тобто один символ може бути перетворено у довільну кількість символів.

Асинхронні автомати переважно є більш складні за синхронні. Це доволі поширений вид автоматів, але надто часто асинхронні автомати можуть губити частину тексту, який вони перетворюють.

Автомат A називають *оборотним*, якщо існує інший автомат B, який перетворює довільну вихідну послідовність автомата A на вхідну послідовність автомата A. Зрозуміло, що автомат є оборотним тоді, й лише тоді, коли він не губить жодної з частин перетворюваного тексту. Вочевидь, що, якщо, приміром, автомат A перетворює довільний текст на порожній, то такий автомат є необоротний. Автомат неодмінно має бути оборотним, якщо його використовують для шифрування (оскільки після шифрування треба буде розшифровувати текст).

При програмуванні асинхронних автоматів для вихідного рядка доцільно використовувати нову змінну, не змінюючи вхідний рядок, як це було у разі синхронніх автоматів.

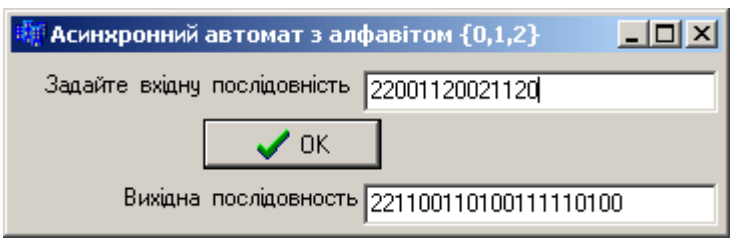
Приклад 13.13 Задано автомат алфавітом X=Y={0,1,2} і двома станами A та B:



Ввести вхідну послідовність символів з алфавіту X і здобути вихідну послідовність Y після дії заданого автомата.

Розв'язок. Для цього автомата незавжди можливо визначити дію автомата за першим символом вхідної послідовності і виникає потреба перевірки наступного символу, тобто дія автомата визначатиметься послідовністю декількох символів. Наприклад, у стані B для вхідного символу '1' правило переходу є відоме – 1/01, а для вхідного символу '2' визначити однозначно дію автомата є неможливо. Тоді для визначення правила переходу слід прочитати ще один символ вхідної послідовності (приміром, якщо другий символ є '1', то перехід буде визначено: 21/11).

Алгоритм роботи даного автомата є такий. Розглядаємо перший, а, якщо треба, то й другий символ рядка S, і відповідно до стану автомата використовуємо ці дані для формування рядка-відповіді T. Опрацьовані символи з рядка S вилучаємо, щоб опрацювання розпочиналося завжди лише з перших символів цього рядка і допоки рядок не стане порожнім.



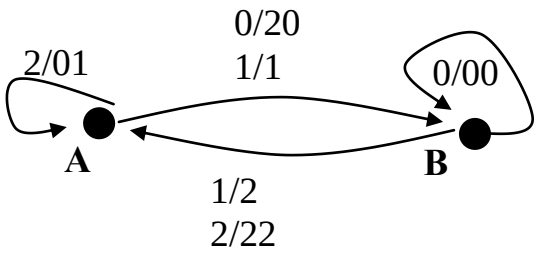
Текст функції та програми для командної кнопки:

```
AnsiString Assinhr1(AnsiString S)
{ enum conditions {A, B}; // Перераховний тип даних
  conditions sit=A;
  AnsiString T="";          // Рядок, в якому формуватиметься результат.
  while(!S.IsEmpty( ))      // Допоки рядок не є порожній,
  { switch (sit)            // перевіряються усі стани автомата
    { case A:
      { if(S[1]=='0'){T=T+"11"; sit=B;}
         else
          if(S[1]=='1'){T=T+"101"; sit=B;}
           else T=T+"2";
        S.Delete(1,1);      // Видалення опрацьованого символу.
        break;
      } // end case A
      case B:
      { if(S[1]=='0') T=T+"0";
        else
          {if(S[1]=='1'){T=T+"01"; sit=A;}
           else
             if(S.Length()>1)
             { String sub=S.SubString(1,2);
               if(sub=="20") T=T+"00"; else
                 if(sub=="21")T=T+"11"; else
                   if(sub=="22") T=T+"2";
               sit=A;
          }  }
        if((S[1]=='2')&&(S.Length()>=1))S.Delete(1,2);
        else S.Delete(1,1);
        break;
      } // end case B
    }   // switch
  }     // while
  return T;
}
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{ AnsiString s=Edit1->Text;
  Edit2->Text=Assinhr1(s);
}
```

Слід зазначити, що задані в такий спосіб асинхронні автомати у певному розумінні не є строго коректними, оскільки автомат не завжди має можливість за вхідною послідовністю однозначно видати вихідну послідовність. Наприклад, як було зазначено вище, якщо в нашому автоматі вхідна послідовність є `S="02"`, то перший символ видасть вихідну послідовність `"11"`, а другий – не видасть жодного результату. Такі автомати є частково визначеними. Але вони є невизначені лише на малих “залишках” вхідної послідовності. Природно, що автомати використовують на текстах досить

великої довжини. Тому невизначеність на одному-двох останніх символах не є критичною. В таких випадках автомат просто не перетворює залишок.

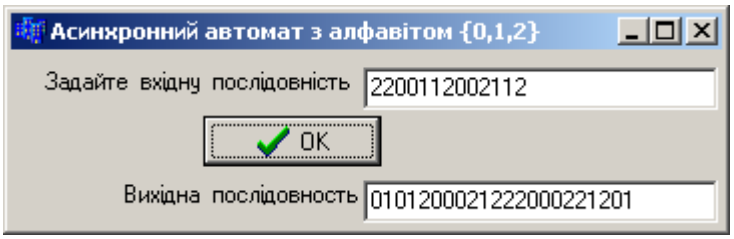
Приклад 13.14 Задано автомат алфавітом X=Y={0, 1, 2} і двома станами – A та B:



Ввести вхідну послідовність символів з алфавіту X і здобути вихідну послідовність після дії заданого автомата.

Розв'язок. Зрозуміло одразу, що цей автомат є асинхронним, оскільки один символ може перетворюватися на два. Але програмування його не потребує зчитування кожний раз більш ніж одного символу вхідного рядка, на відміну від попереднього прикладу. Тому зчитування символів вхідної послідовності можна організувати за допомогою циклу for.

У цьому разі ідея програмування автомата поєднує випадки синхронних та асинхронних автоматів.



Текст функції та програми для командної кнопки:

```
AnsiString Assinhr2 (AnsiString S)
{ enum conditions{A,B};
  conditions sit=A;
  AnsiString T="";   // Рядок–результат
  for (int i=1;i<=S.Length();i++)
  switch (sit)
  { case A:
      { if (S[i]=='1'){T=T+"1"; sit=B;}
        else
          if (S[i]=='0'){T=T+"20"; sit=B;}
          else T=T+"01";
        break;
      } // end case A
    case B:
      { if(S[i]=='0') T=T+"00";
        else
          if(S[i]=='1'){T=T+"2"; sit=A;}
          else
```

```
                { T=T+"22"; sit=A;}
            break;
          } // end case B
    }// switch
    return T;
}
//---------------------------------------------------------
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{ String s=Edit1->Text;
  Edit2->Text= Assinhr2(s);
}
//---------------------------------------------------------
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, char &Key)
{ if(Key<'0' || Key>'2') Key=0;
}
```

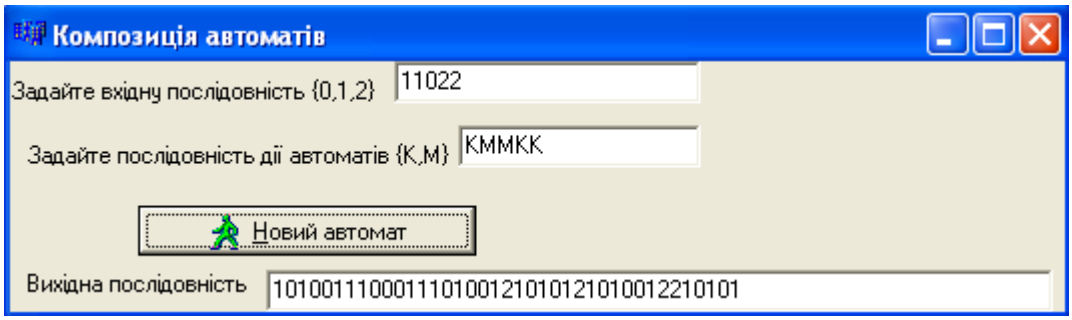
13.7.4 Композиція автоматів

Вище було розглянуто приклади роботи різних автоматів. Але теорія автоматів здебільшого має справу не з одним автоматом, а з множиною автоматів. Наприклад, для заданих автоматів M та K можна утворити новий автомат N, який є результатом композиції цих двох автоматів. X, Y – відповідно вхідний і вихідний алфавіти автомата M; Y, Z – вхідний і вихідний алфавіти автомата K. Для автомата N, який є композицією автоматів M та K, будуть виконуватись такі перетворення. Спочатку автомат N перетворить вхідний рядок алфавіту X, як це робить автомат M, потім автомат N перетворить дістаний текст алфавіту Y на текст алфавіту Z, як це робить автомат K. Якщо вхідний і вихідний алфавіти автомату M збігаються (X=Y), то можна розглянути композицію автомата M на самого себе. Тоді $N=M^2$. Аналогічно утворюються інші степені автомата. Комбінуючи дії автоматів M та K, можна утворити нескінченну кількість нових автоматів.

Приклад 13.15 Нехай задано два асинхронні автомати K та M з алфавітами X=Y={0, 1, 2} і двома станами A та B (див. приклади 13.13 та 13.14). Створити новий автомат N як композицію автоматів K та M. Порядок використання автоматів у композиції (зліва-направо) задано послідовністю елементів алфавіту {K, M}.

Розв'язок. У запропонованій програмі організовано два рядки: S – вхідна послідовність символів множини {'0', '1', '2'}; T – послідовність дії автоматів K та M. Результатом є перетворений рядок вихідного алфавіту множини символів {'0', '1', '2'}, який отримується після дії композиції автоматів.

Програма використовуватиме функції Assinhr1() й Assinhr2() з прикладів 13.13 та 13.14.



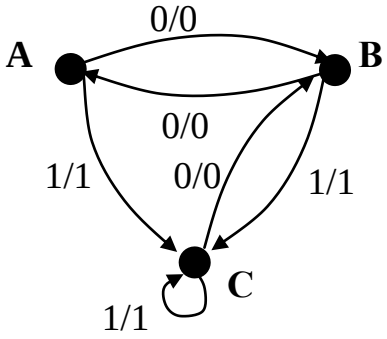
Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString S=Edit1->Text;
  AnsiString T=Edit2->Text;
  while(T!="")
   { if(T[1]=='K') S = Assinhr1(S); else S= Assinhr2(S);
     T.Delete(1,1);
   }
  Edit3->Text=S;
}
```

Очевидним є збільшення розміру перетвореного тексту після дій цих асинхронних автоматів. У запропонованій програмі автомати K та M спрацьовують згідно до порядку, заданого в компоненті `Edit2` (рядок `T`). У наведеному на формі прикладі роботи програми, першим спрацює автомат `K`, який перетворить вхідну послідовність “`11022`” на послідовність “`10101112`”. Наступним спрацює автомат `M`, який перетворить здобуту на попередньому кроці послідовность “`10101112`” на вихідну послідовність “`10022021201`”. І для цієї послідовності знову буде викликано функцію дії другого автомата, оскільки третім символом рядка `T` є також символ `'M'`. Як наслідок буде обчислено вихідну послідовність “`1000022012022122202`”. Далі буде викликано функцію дії першого автомата, оскільки у рядку `T` наступний символ є `'K'`. Вихідна послідовність стане “`1010000211012112101221 1`”. В останній раз буде знову викликано функцію дії першого автомата, оскільки у рядку `T` останній символ є `'K'`.

Зауважимо, що кожний розглядуваний автомат мав так званий початковий стан (у наших прикладах стан `A`). Але, якщо змінити початковий стан автомата на інший, буде утворено інший (новий) автомат. В сучасній теорії автоматів часто розглядають композиції автоматів, які утворені з одного автомата, але з різними початковими станами.

Також слід зазначити, що різні схеми (графи) автоматів можуть визначати один і той самий автомат. Так, наведений поряд автомат не змінює жодний текст. Нескладно створити інші види такого автомата.



Питання та завдання для самоконтролю

1) Дайте означення лінійного списку.

2) Дайте означення циклічного списку.

3) Які є різновиди списків?

4) Що зберігає елемент списку?

5) Наведіть оголошення елемента лінійного однозв'язного списку, який містить дійсні числа.

6) Як звернутися до другого елемента списку, оголошеного у завданні 5?

7) Елемент якого списку "знає", де розташовано попередній і наступний елементи? Наведіть оголошення елемента такого списку.

8) Як звернутися до передостаннього елемента списку, оголошеного у завданні 5?

9) Нехай у програмі список оголошено в такий спосіб:

```
struct List { char c; List n; } first;
```

Цей список містить такі значення: `'d'`, `'y'`, `'5'`, `'*'`, `'2'`, `'h'`. Поясніть значення наступних виразів:

а) `first->n->c;` б) `first->n->n;` в) `first->n->n->c;`

10) Наведіть оголошення елемента однозв'язного списку, який "знає", де розташовано попередній елемент.

11) В який спосіб можна вилучити з двонапрямленого списку елемент із заданим номером.

12) Дайте визначення стека і черги.

13) Які дії можна виконувати при роботі зі стеком і чергою?

14) Які вказівники треба використовувати для роботи зі стеком і чергою?

15) Яким чином долучити та вилучити перший, останній і довільний елементи лінійного списку?

16) Які елементи заборонено вилучати з черги?

17) Які елементи заборонено вилучати зі стека?

18) Чим дерево відрізняється від лінійних списків?

19) Дайте визначення листа та кореня дерева.

20) У чому полягає особливість бінарних дерев?

21) Як долучаються вузли до дерева?

22) Як вилучаються вузли з дерева?

23) В якому дереві вузли не містять більше двох посилань?

24) Як називається вузол дерева, що не має предків?

25) Як називається вузол дерева, що не має нащадків?

26) Зобразити бінарне дерево, обхід якого зліва направо формує таку послідовність значень: 27, 11, 19, 18, 49, 32, 53, 67, 72, 60, 81, 97, 91, 98.

27) Дайте визначення автомата.

28) Чим відрізняється синхронний автомат від асинхронного?

29) Чи можуть збігатися вхідний і вихідний алфавіти автомата?

30) Чи може асинхронний автомат перетворювати послідовність елементів у послідовність тої самої довжини?

Розділ 14

Об’єктно-орієнтоване програмування

14.1 Модульне й об’єктно-орієнтоване програмування

Об’єктно-орієнтований підхід до програмування є пріоритетним при створюванні переважної більшості програмних проектів.

В остаточному вигляді кожна програма є набором інструкцій для процесора. І чим вище є рівень мови, тим у більш простій формі записуються одні й ті ж самі дії. З нарощуванням обсягу програм стає потрібним структурувати інформацію, виокремлювати в ній головне та відкидати несуттєве. Цей процес називається підвищенням ступеня *абстракції* програми.

Першим кроком до підвищення абстракції є *використання функцій*, що дозволяє після написання та налагодження функції дистанціюватись від деталей її реалізації, оскільки для виклику функції треба знати лише її інтерфейс.

Наступний крок – *оголошення власних типів даних*, які дозволяють структурувати та групувати інформацію, подаючи її в більш природному вигляді. Оскільки для роботи з власними типами даних потрібні спеціальні функції, тому вважається за природне згруповувати їх разом з оголошенням цих типів в одному місці програми і у певний спосіб відокремити від решти її частин.

Отже, *об’єднання в модулі* оголошень типів даних та функцій, призначених для роботи з цими типами, разом із приховуванням від користувача модуля несуттєвих деталей, є подальшим розвитком структуризації програми.

Всі три згаданих вище методи підвищування абстракції ставлять за мету спростити структуру програми, тобто подати її у вигляді невеликої кількості більш потужних блоків та мінімізувати зв’язки поміж ними. Це дозволяє керувати великим обсягом інформації, а отже, успішно налагоджувати великі програми.

Введення поняття *класу* є розвитком ідей модульності. У класі поєднуються структури даних і функції їхнього опрацювання. Класи дуже схожі на структури, які було розглянуто у підрозд. 11.2. Ідея класів є підґрунтям об’єктно-орієнтованого програмування (ООП). Програми на C++ широко використовують класи.

Клас є типом даних, який визначає користувач. У класі задаються властивості та характер певного предмета чи процесу у вигляді полів даних (аналогічно до структури) і функцій для роботи з ними. Наприклад, клас телефон може мати поля даних: номер, тип телефону (кнопковий чи дисковий) і функції роботи з телефоном: дзвінок, набирання номера, з’єднання з абонентом тощо. Групування даних про об’єкт та кодування їх однією змінною спрощує процес програмування та збільшує можливість повторного використання коду.

Суттєвою властивістю класу є те, що деталі його реалізації приховано від користувачів класу за інтерфейсом. Це захищає їх від випадкових змінювань. Інтерфейсом класу є заголовки його методів.

Ідея класів відображає будову об'єктів реального світу. Адже кожен об'єкт чи процес має набір певних характеристик чи відмінностей, інакше кажучи, певні властивості й поведінку. Так характеристиками автомобіля є марка, колір, максимальна швидкість, потужність двигуна тощо; характеристиками посилювача є частотний діапазон, потужність, коефіцієнт нелінійних перекручувань тощо. Функціональні можливості об'єктів теж відрізняються: автомобіль може їздити, посилювач – підвищувати рівень сигналів. А користуватись об'єктами можна, не знаючи їхньої внутрішньої будови. Наприклад, керувати автомобілем можна, не маючи жодного уявлення про будову його двигуна чи будь-яких інших його частин.

Кожен клас займає певне місце в ієрархії класів, наприклад, усі автомобілі належать до класу наземний транспорт (більш високого в ієрархії), а клас автомобілі містить багато різновидів автомобілів: вантажні, легкові, позашляховики тощо. Отже, будь-який клас визначає певну категорію об'єктів, а кожен об'єкт є екземпляром деякого класу. ООП – це методика, яка концентрує основну увагу програміста на зв'язках поміж об'єктами, а не на деталях їхньої реалізації.

Основними поняттями, на яких базується ООП, є:

- ✓ інкапсуляція;
- ✓ успадкування;
- ✓ поліморфізм.

Інкапсуляцією (encapsulation) називається поєднання даних з функціями їхнього опрацювання разом із приховуванням зайвої для користування цими даними інформації. Інкапсуляція підвищує рівень абстракції програми, дозволяє змінювати реалізацію класу без модифікації основної частини програми та використовувати клас в іншому оточенні.

Успадкування (inheritance) – це можливість створювання ієрархії класів, коли класи-нащадки успадковують властивості своїх базових класів (предків), можуть змінювати ці властивості й набувати нових. Властивості базових класів (предків) при успадкуванні не описуються, що скорочує обсяг програми.

Поліморфізм (polymorphism) – це можливість використовувати у різних класах ієрархії одне ім'я для позначення близьких за змістом дій та гнучко обирати відповідні дії у перебігу виконування програми.

Отже, об'єктно-орієнтоване програмування у жодний спосіб не пов'язане з процесом виконування програми, а є лише новим способом її організації, тобто новою *парадигмою* програмування (парадигма – набір теорій та методів, які визначають спосіб організації знань).

Існує три групи мов програмування, які пов'язані з поняттям клас: об'єктно-орієнтовані, об'єктні, об'єктно-базовані. *Об'єктно-орієнтовані мови* у повному обсязі підтримують парадигму ООП: інкапсуляцію, успадкування та поліморфізм. Типовими представниками таких мов є C++, Java, C#. До *об'єктних мов* відносять мови програмування, які підтримують тільки інкапсуляцію та дозволяють створювати об'єкти; це мови Visual Basic (до шостої версії включно) та Ada. До *об'єктно-базованих* мов програмування відносять мови, які можуть використовувати існуючі об'єкти, але не мають механізму створення об'єктів користувача. Мова JavaScript є об'єктно-базованою мовою програ-

мування. Так, за допомогою цієї мови можна використовувати численні об'єкти об'єктної моделі документа (DOM – Document Object Model), за допомогою яких можна надавати зміст веб-сторінки.

Існує помилкова думка, що об'єктно-орієнтоване програмування є щось складне та незрозуміле. Але об'єктна декомпозиція є не менш природною та інтуїтивно зрозумілою, ніж алгоритмічна, яка панувала до появи ООП. До програмування основні поняття ООП перейшли з інших галузей знань, таких як філософія, логіка й математика, причому, без особливих змін, принаймні того, що стосується суттєвості цих понять. Об'єктний спосіб декомпозиції (уявлення) є природним, і використовується протягом багатьох століть. Тому й не дивно, що в процесі еволюції технології програмування цей спосіб посідає гідне місце та підтримується так чи інакше практично всіма сучасними алгоритмічними мовами.

14.2 Визначення класу

Клас є абстрактним типом даних, визначуваним користувачем, і зображує модель реального світу у вигляді даних та функцій їхнього опрацювання. Дані класу називають *полями* чи *даними-членами* (data members), а функції класу – *методами* чи *функціями-членами* (methods, member functions). Поля та методи називаються елементами класу.

Здебільшого специфікація класу складається з двох частин:

- ✓ *оголошення* класу – у ньому прописано всі поля й методи (елементи даних) класу на рівні інтерфейсу в термінах функцій-елементів;
- ✓ *визначення* методів класу, тобто реалізації конкретних функцій-членів даного класу.

Оголошення класу має такий формат:

```
class <ім'я класу>
{ [ private: ]
    < Оголошення прихованих елементів класу >
  protected:
    < Оголошення елементів, доступних тільки нащадкам >
    public:
    < Оголошення загальнодоступних елементів >
};      // Оголошення завершується крапкою з комою
```

Специфікатори доступу `private`, `protected` та `public` керують видимістю елементів класу. Елементи, визначені після ключового слова `private`, є доступними лише у цьому класі. Цей різновид доступу прийнято у класі за замовчуванням. Специфікатор доступу `protected` містить поля і методи, які приховані від усіх, окрім нащадків класу. Поля і методи, визначені ключовим словом `public`, є доступними поза класом, тобто до них можна звертатися безпосередньо з програми, використовуючи об'єкти класу. Вміст загальнодоступного розділу `public` становить абстрактну частину конструкції, тобто загальнодоступний інтерфейс, який містить прототипи функцій-елементів чи повне визначення (для невеликих функцій).

Розглянемо приклад оголошення класу:

```
class myclass
{ private: // Приховані елементи
    int a;
  public:  // Загальнодоступні елементи
    void set_a(int num){a = num;} // Метод, який змінює значення поля
    int get_a(){return a;}        // Метод, який повертає значення поля
};
```

У цьому прикладі клас `myclass` містить лише одне поле даних `a`, яке має тип `int`, причому це поле є доступним лише у класі, оскільки воно оголошено з ключовим словом `private`. Звертання до цього поля з програми неможливе і спричинить помилку. Однак доступ до даних можна організувати за допомогою методів, які ще називають методами доступу. Такими у класі `myclass` є два методи: `set_a()`, який присвоює полю значення, та `get_a()`, який повертає значення поля. Ці методи є доступними за межами класу, оскільки їх оголошено ключовим словом `public`. Тіла обох методів (коди функцій) містяться безпосередньо в оголошенні класу. Тут реалізація функцій не означає, що код функції розміщується у пам'яті. Це відбудеться лише при створенні об'єкта класу. Методи класу, визначені у подібний спосіб, за замовчуванням є *вбудованими функціями*.

При збільшенні методів за кількістю та обсягом, визначення функцій класу як вбудованих може спричинити безладдя в оголошенні класу. Щоб уникнути цього, функції всередині класу найчастіше не визначаються, а лише оголошуються (тобто розміщуються їхні прототипи), а саме визначення функцій відбувається в іншому місці. *Функція, визначена поза класом*, за замовчуванням вже не є вбудованою.

Для визначення функції-члена класу поза класом необхідно поєднати ім'я класу з ім'ям функції за допомогою *операції визначення області видимості* "`::`" за форматом:

<тип функції> <ім'я класу>`::`*< ім'я функції-члена>*

Наприклад, оголосимо клас `myclass` *без вбудованих функцій-членів класу*:

```
class myclass
{ private:   // Приховані елементи
    int a;
  public:    // Загальнодоступні елементи (методи)
    void set_a(int num);
    int get_a();
};
```

Функції класу тепер можна визначити в іншому місті (чи то після оголошення класу, чи навіть в іншому файлі). Наприклад, реалізація функцій класу `myclass` поза класом матиме вигляд:

```
void myclass::set_a(int num)
 { a = num; }
int myclass::get_a()
{ return a; }
```

Приклад 14.1 Побудувати клас `worker` (робітник) без нащадків, який містить оголошення полів даних (`name` – прізвище, `worker_id` – код, `salary` – розмір зарплатні робітника) і визначення методу класу – функцію поза класом (`show_worker()` – виведення інформації про робітника у консольному режимі):

```
class worker
{ private:     // Прихованні поля класу
    char name[64] ;
    long worker_id;
    float salary;
  public:     // Загальнодоступний метод класу
    void show_worker(void);
};
// Реалізація методу класу поза класом
void worker::show_worker(void)
{ cout << "Прізвище: " << name << endl;
  cout << "Код робітника: " << worker_id << endl;
  cout << "Зарплатня: " << salary << endl;
};
```

14.3 Створення об'єктів класу

Тепер, коли певний клас визначено (див. п.14.2), можна створювати конкретні змінні цього класу, які називають *екземпляри класу* чи *об'єкти*. Час їхнього життя та видимість об'єктів залежить від форми та місця їхнього оголошення і підпорядковуються загальним правилам C++. Насправді об'єкт в ООП – це змінна, тип для якої оголошує програміст. Кожен елемент даних такого типу є складеною змінною.

Наприклад, створимо три об'єкти класу `myclass`:

```
myclass ob1, ob2, *ob3;   // Об'єкти типу myclass
```

Після того, як об'єкти класу створено, можна звертатися до відкритих елементів класу. Використовуючи операцію доступу до члена класу (**.**) звернемося до методів класів об'єктів `ob1` та `ob2`:

```
ob1.set_a(10);      // Звертання до методу set_a() для об'єкта ob1
ob2.set_a(99);      // Звертання до методу set_a() для об'єкта ob2
```

Тут доступ до елементів об'єкта (у даному разі до методів) є аналогічний доступу до полів структури. Для цього після імені об'єкта (`ob1`) ставиться крапка (**.**), а після неї зазначається ім'я методу (`set_a()`). У такий спосіб задається метод та об'єкт застосування цього методу, тобто крапка пов'язує метод з ім'ям об'єкта. Доступ до такого (статичного) члена класу здійснюється за допомогою операції-крапки (**.**). Тобто оператор `ob1.set_a(10)` викликає метод `set_a(int num)` об'єкта `ob1`, який присвоює полю `a` об'єкта `ob1` значення `10`. Оператор `ob2.set_a(99)` присвоює полю `a` об'єкта `ob2` значення `99`.

Оператори

```
int x1 = ob1.get_a();
int x2 = ob2.get_a();
```

викликають метод `get_a()` для об'єктів `ob1` та `ob2` і присвоюють змінним `x1` та `x2` значення полів відповідних об'єктів, повернутих цим методом. Як бачимо, дужки після імені методу є обов'язковими, навіть якщо немає параметрів усередині. Дужки "говорять" про те, що здійснюється виклик функції, а не використовується значення змінної. Тепер можна вивести здобуті дані про об'єкти `ob1` та `ob2`, наприклад до компонента `Edit`:

```
Edit1->Text = "В об'єкті ob1 поле a=" + IntToStr(x1);
Edit2->Text = "В об'єкті ob2 поле a=" + IntToStr(x2);
```

Для *динамічної змінної* `*ob3` необхідно попередньо виділити пам'ять, а вже потім використовувати операцію. Доступ до динамічного члена класу здійснюється за допомогою операції-стрілка (**->**), наприклад:

```
myclass *ob3 = new myclass; // Виділення пам'яті для об'єкта
ob3->set_a(46);             // Полю a об'єкта *ob3 надано значення 46
Edit3->Text = "В об'єкті ob3 поле a=" + IntToStr(ob3.get_a());
```

Отже, результат виконання розглянутих вище операторів матиме вигляд:

```
В об'єкті ob1 поле a=10
В об'єкті ob2 поле a=99
В об'єкті ob3 поле a=46
```

Примітки. Звернутися за допомогою операції крапки (**.**) чи операції (`->`) можна лише до елементів зі специфікатором `public`. Здобути доступ чи змінити значення елементів зі специфікаторами `private` чи `protected` можна лише через звертання до відповідних методів.

Операція вибору члена (.) працює в точності так само, як операція (->), за винятком того, що імені об'єкта передує неявно згенерований компілятором оператор адреси (&). Отже, інструкцію

```
ob1.set_a(10);
```

компілятор трактує як

```
(&ob1)->set_a(10);
```

Якщо до закритого методу елемента класу звернутися безпосередньо через об'єкт, а не через методи класу, то така дія призведе до помилки. Наприклад, інструкція `ob1.a=10;` спричинить помилку (поле `a` оголошено у розділі `private`), а якщо елемент класу `int a`, оголосити у розділі доступу `public`, то тоді до змінної `a` можна буде звертатися з будь-якої частини програми, приміром так: `ob1.a=10;`

14.4 Використання загальнодоступних та приватних елементів класу

Наведемо програму, яка оголошує клас з трьома полями даних (`name`, `worker_id`, `salary`) та функцією-членом (`show_worker()`), яку реалізовано поза класом (див. приклад 14.1). Усі елементи класу оголосимо як загальнодоступні.

У програмі створимо два об'єкти для класу `worker`, надамо значення елементам даних і за допомогою функції `show_worker()` виведемо інформацію про робітників у консольному режимі.

```
#include <iostream.h>
#include <string.h>
class worker
{ public: // Загальнодоступні елементи класу
    char name[64] ;
    long worker_id;
    float salary;
    void show_worker(void);
};

// Реалізація функції-члена класу поза класом
void worker::show_worker(void)
{ cout << "Прізвище: " << name << endl;
  cout << "Код робітника: " << worker_id << endl;
  cout << "Зарплатня: " << salary << endl;
};

// Головна програма – створення та робота з об'єктами класу
void main(void)
{ worker engineer, boss; // створення об'єктів типу worker
  strcpy(engineer.name, "Павленко");
  engineer.worker_id = 345;
  engineer.salary = 5000.00;
  strcpy(boss.name, "Марченко");
  boss.worker_id = 101;
  boss.salary = 10500.00;
  engineer.show_worker();
  boss.show_worker();
}
```

Як бачимо, у головній програмі `main()` оголошено два об'єкти типу `worker`: `engineer` та `boss`, для яких використано операцію крапки при присвоюванні значень елементам об'єкта та при звертанні до функції `show_worker()` для виведення результатів.

Результати виконання програми:

```
Прізвище: Павленко
Код робітника: 345
Зарплатня: 5000.00

Прізвище: Марченко
Код робітника: 101
Зарплатня: 10500.00
```

Зверніть ще раз увагу на те, що в цьому прикладі всі елементи класу є загальнодоступними, що надало змогу звертатися до всіх елементів класу безпо-

середньо через об'єкти `engineer` та `boss` (`engineer.name`, `boss.show_worker()`, `boss.salary`).

При створенні класу можливо мати елементи, значення яких використовується тільки в класі, але звертатися до яких у самій програмі немає потреби. Такі елементи є приватними (`private`), їх слід приховувати від програми. Якщо спеціально не використовувати специфікатор `public`, то за замовчуванням C++ вважає, що всі елементи класу є приватними. Зазвичай, треба захищати елементи класу від прямого доступу до них, оголошуючи для них приватний доступ. Тільки при такому доступі можна гарантувати користувачеві програмного продукту, що елементам класу буде надано припустимі значення. Приміром, у наведеному прикладі програми об'єкти `engineer` та `boss` використовують змінну на ім'я `salary`, яка може набувати значення тільки в діапазоні від 1000 до 15000. Якщо елемент `salary` є загальнодоступний як у вищенаведеному прикладі, то програма може безпосередньо звертатися до елемента, змінюючи його значення без обмежень:

```
engineer.salary = 101; boss.salary = 20000;
```

Якщо оголосити змінну `salary` приватною, то для присвоювання значень цій змінній треба створити та використати додатковий метод класу. Наприклад, функція `assign_salary()` може перевіряти, чи є значення поля `salary` припустимим.

```
class worker
{ private:
   float salary;
   . . . .
  public:
   int assign_salary (int value);
   . . . .
};

int worker::assign_salary (int value)
{ if ((value > =1000) && (value <= 15000))
    { salary = value; return(0); } // Успішне присвоєння
  else return(-1);                 // Неприпустиме значення
}
```

Методи класу, які керують доступом до елементів даних, – це інтерфейсні функції. При створенні класів бажано використовувати ці інтерфейсні функції для захисту даних своїх класів.

Вищерозглянуті приклади демонструють способи використання методів класу для ініціалізації полів об'єкта класу. Зручнішою є ініціалізація поля об'єкта на момент його створювання, а не явний виклик у програмі відповідного методу. Такий спосіб можна реалізувати за допомогою спеціального методу класу, який називається конструктором.

14.5 Конструктори

14.5.1 Властивості конструкторів

При створюванні об'єктів одною з найважливіших операцій, яку виконують в усіх програмах, є ініціалізація елементів даних об'єкта. Для того, щоби звернутися до приватних елементів класу у попередніх прикладах ми спеціально створювали функції (`assign_salary()`, `set_a()`, `get_a()`). Для спрощення процесу ініціалізації елементів класу, алгоритмічна мова C++ має спеціальну функцію-член, яку називають конструктор.

Конструктор (constructor) – це спеціальна функція (метод) класу, яка автоматично виконується при створюванні об'єкта та присвоює значення елементам класу для об'єкта, що створюється. Він може не лише ініціалізовувати змінні класу, а й виконувати будь-які інші завдання ініціалізації, необхідні для підготовки об'єкта до використання (наприклад, перевірку припустимості значень). Конструктор пов'язує ім'я класу з його закритими для доступу полями за форматом:

<ім'я класу>(*<змінна поля1>*, *<змінна поля2>*, . . .)

Наприклад, оголошення класу `worker` з конструктором матиме вигляд:

```
class worker // Оголошення імені класу
{ private:    // Прихованні поля класу
   char name[64] ;
   long worker_id;
   float salary;
  public:    // Загальнодоступні методи класу
 worker(char *iname, long iworker_id, float isalary);// Конструктор
   void show_worker(void); // Метод для виведення інформації
};
```

У цьому конструкторі будуть ініціалізуватись змінні (поля) з розділу доступу `private: name, worker_id, salary`. Щоби відрізняти відповідну змінну від подібного імені поля класу в оголошенні конструктора до імені полів класу було долучено літеру "`i`" (**i**`name, `**i**`worker_id, `**i**`salary`)

Для *створення екземпляра класу* необхідно, щоби конструктор було оголошено як загальнодоступний (`public`). При оголошенні об'єкта класу значення його полів передаються конструкторові за форматом:

<ім'я класу> <ім'я об'єкта>(*<значення поля1>*,*< значення поля2>*, . . .);

Наприклад, використовуючи оголошення конструктора класу `worker` створимо об'єкт `engineer`:

```
worker engineer ("Павленко", 345, 5000.00);
```

Використання конструктора у наведеному прикладі зменшило кількість операторів ініціалізації полів об'єкта класу, причому поля даних є захищені від загального доступу.

Розглянемо *основні властивості* конструктора.

✓ Конструктор *виконується автоматично* в момент створення об'єкта, чим спрощує задачу ініціалізації даних об'єкта.

✓ Ім'я конструктора абсолютно *збігається з іменем класу*. Отже, компілятор відрізнює конструктор від інших методів класу.

✓ Конструктор *не повертає жодного значення*, навіть типу `void`. Не можна здобути і вказівник на конструктор. Це пояснюється тим, що конструктор автоматично викликається системою, а отже, не існує програми чи функції, яка його викликає, і якій конструктор міг би повернути значення. Отже, задавати значення, яке повертається, для конструктора немає сенсу.

✓ Клас може мати *кілька конструкторів з різними параметрами* для різних видів ініціалізації, при цьому використовується механізм перевантаження.

✓ Конструктор може прийняти *яку завгодно кількість аргументів* (включаючи нульову).

✓ *Параметри конструктора можуть бути якого завгодно типу*, окрім цього класу. Можна задавати значення параметрів за замовчуванням. Їх може містити лише один із конструкторів.

✓ Конструктори *не успадковуються*.

✓ Конструктори *не можна оголошувати з модифікаторами* `const`, `virtual` та `static`.

✓ Знищення об'єкта з пам'яті комп'ютера виконує спеціальна функція – *деструктор*.

Розглянемо різні способи визначення конструкторів.

14.5.2 Конструктор з параметрами

Конструктор може ініціалізувати поля класу, використовуючи оператор присвоювання за форматом:

<ім'я класу>(*<змінна поля1>*,*< змінна поля2>*, …)
{*<ім'я поля1>* **=** *< змінна поля1>*; *<ім'я поля2>* **=** *< змінна поля2>*; . . . }

Наприклад, визначимо конструктор з операторами присвоєння для класу `worker` (як вбудовану функцію):

```
worker (char *iname, long iworker_id, float isalary)
{ name = iname; worker_id = iworker_id;salary = isalary; }
```

У цьому разі для створення об'єкта викликається конструктор, до якого передаються відповідні значення параметрів:

```
worker boss ("Марченко",101,10500.00);
```

За цього способу визначення конструктора легко реалізувати умови на припустимість значень. Наприклад, при створенні об'єкта завжди будемо перевіряти: чи належить значення поля `salary` діапазону від 1000 до 15000:

```
worker (char *iname, long iworker_id, float isalary)
{ name = iname;  worker_id = iworker_id;
  if ((isalary>=1000) && (isalary<=15000)) salary = isalary;
  else salary = 0;     // Якщо значення є неприпустиме, присвоюємо 0
}
```

Окрім того, за такого способу *створення екземпляра класу з вказівником*, значення параметрів можна передавати до конструктора, використовуючи оператор `new`. Наприклад, створимо динамічний об'єкт `*menedger`:

```
worker *menedger = new worker("Зименко", 223, 7500.00);
```

14.5.3 Конструктор зі списком ініціалізації

Вищеописаний конструктор не може бути використаним при ініціалізації полів-констант чи полів-посилань, оскільки їм не можуть бути присвоєні значення. Для цього передбачено спеціальну властивість конструктора, називану *списком ініціалізації*, який дозволяє ініціалізувати одну чи більше змінних і не надавати їм значення. Список ініціалізації розташовується поміж прототипом методу та його тілом і після двокрапки, при цьому ініціалізуюче значення розміщується у дужках після імені поля, значення у списку розділяються комами за форматом:

<ім'я класу>`()`:*<ім'я поля1>*`(`*<значення поля1>*`)`,
<ім'я поля2>`(`*<значення поля2>*`)`, . . . { . . . }

Наприклад, конструктор зі списком ініціалізації усіх полів класу `worker` матиме вигляд:

```
worker():name("Марченко"),worker_id(101),salary(10500.00){}
```

Такий конструктор називають *конструктором без параметрів*. Для створення об'єктів з використанням такого конструктора достатньо записати ім'я класу та імена об'єктів через кому, наприклад:

```
worker w1, w2, w3;
```

Тут усі створювані об'єкти (`w1, w2, w3`) матимуть такі само початкові значення, як у об'єкта `boss`.

При створенні об'єктів поля можна також ініціалізувати за допомогою списку змінних поля, в якому значення можуть бути виразами, наприклад:

```
worker (char *iname, long iworker_id, float isalary):
name(iname), worker_id(iworker_id), salary(isalary) {}
```

При створюванні цього об'єкта буде викликано конструктор з відповідними, зазначеними у дужках параметрами, приміром:

```
worker w4("Ткаченко", 454, 6000.00);
```

Можливо також використовувати комбінацію способів визначення конструкторів зі списками ініціалізації та змінних поля, наприклад:

```
worker(char *iname, long worker_id):
   name(*iname), worker_id(worker_id), salary(1000.00) {}
```

Цей конструктор має тільки дві змінні поля та одне константне значення (`salary = 1000.00`). Створення об'єкта (наприклад, `w5`) за таким конструктором матиме вигляд:

```
worker w5("Соловейко",255);
```

14.5.4 Конструктор за замовчуванням

Конструктор без параметрів називають *конструктором за замовчуванням*. Такий конструктор зазвичай ініціалізує поля класу константними значеннями, як для об'єктів `w1`, `w2`, `w3`, які розглянуто у п. 14.5.3. Якщо конструктор для будь-якого класу не визначено, то компілятор сам генерує конструктор за замовчуванням.

Якщо програмістові треба однозначно ініціалізувати поля, треба визначити власний конструктор (ним може бути і конструктор за замовчуванням). Тоді для класу, який має конструктор за замовчуванням, можна визначити об'єкт класу без передавання параметрів, наприклад:

```
worker obj1, obj2;
```

У класі може бути визначено не один конструктор з одним і тим самим іменем. Тоді говорять, що *конструктор є перевантаженим*. Який з них буде виконуватись при створюванні об'єктів, залежить від того, скільки аргументів використовується у виклику. *Якщо у класі визначено будь-який конструктор, то компілятор не створить конструктора за замовчуванням.* І якщо у класі не буде конструктора за замовчуванням, у певних ситуаціях можуть виникати помилки. У такому разі слід визначити свій власний конструктор за замовчуванням, наприклад:

```
worker ():name (""), worker_id (0), salary (0) {}
```

Примітка. Конструктори за замовчуванням не присвоюють початкових значень полям класу.

14.5.5 Конструктор копіювання

Вище розглянуто два види конструкторів – конструктор без аргументів, який ініціалізує поля об'єкта константними значеннями, та конструктор, який має хоча б один аргумент, який ініціалізує поля значеннями, переданими йому в якості аргументів.

Тепер розглянемо ще один спосіб ініціалізації об'єкта, який використовує значення полів вже існуючого об'єкта. Такий конструктор не треба самим створювати, він надається компілятором для кожного створюваного класу і називається *конструктором копіювання за замовчуванням*. Він може мати лише один аргумент, який є об'єктом того ж самого класу.

Наприклад, створимо три об'єкти `work1`, `work2`, `work3` у різні способи:

```
worker work1("Серченко", 300, 5000.00); // Створення об'єкта work1
worker work2(work1); // Об'єкт work2 – копія об'єкта work1
work3 = work1;       // Об'єкт work3 дорівнює об'єктові work1
```

Тут ініціалізовано об'єкт `work1` значеннями `("Серченко", 300, 5000.00)` за допомогою конструктора з трьома параметрами. Потім визначено ще два об'єкти класу з іменами `work2` та `work3`, які ініціалізуються значеннями об'єкта `work1`. Для копіювання значень полів об'єкта `work1` у відповідні поля об'єктів `work2` та `work3` двічі викликається конструктор копіювання. У результаті всі три об'єкти матимуть однакові значення, ідентичні значенням об'єкта `work1`.

Примітка. Якщо клас містить *вказівники чи посилання*, виклик конструктора копіювання призведе до того, що й копія, й оригінал вказуватимуть на одну й ту саму ділянку пам'яті. У такому разі конструктор копіювання має бути створений програмістом у такому вигляді:

<ім'я класу> `::` *<ім'я класу>* `(const` *<ім'я класу>* `&) {...}`

Приклад 14.2 Створити об'єкт "працівник", в якому передбачити наявність методів обчислення фактичної зарплатні та податку, утриманого з неї. У цьому прикладі податок визначається в такий спосіб: якщо за основним місцем роботи зарплатня працівника не перевищує 300 гривень, то вона не оподатковується, інакше – податок становить 15 % від зарплатні. Але, якщо працівник працює за сумісництвом, то незалежно від розміру зарплатні відраховується податок у 20 %. Використати цей об'єкт для обчислення фактичної зарплатні, яку отримає окремий працівник, та величини сплачуваного податку.

Розв'язок. Програмний код з визначенням класу запишемо *в окремих модулях* `Prac.h` та `Prac.cpp`. Для створення цих файлів виконаємо команди меню Borland C++: **File/New/Unit**. При цьому у проекті буде створено файли `Unit2.cpp` та `Unit2.h`. Для файла `Unit2.cpp` виконаємо команду **File/Save as…** та надамо йому нове ім'я `Prac.cpp` (у теці проекту); автоматично зміниться ім'я файла `Unit2.h` на `Prac.h`.

Запишемо у файлі `Prac.h` оголошення класу `pracivnik` (перед директивою `#endif`)

Файл Prac.h:

```
#ifndef PracH
#define PracH
//---------------------------------------------------------
// Оголошення класу
class pracivnik
{ private:
    double oklad; int vid;
  public:
    // Конструктор за замовчуванням
    pracivnik(): oklad(0), vid(0) {}
    // Конструктор з параметрами
    pracivnik (double okl, int v) { oklad = okl; vid = v; }
    // Метод обчислювання зарплатні
    double zarplatnia();
    // Метод обчислювання податку
    double podatok();
};
#endif
```

Оголошений у програмі клас `pracivnik` містить два поля: `oklad` – для зберігання значення окладу працівника, та `vid`, яке набуває значення 0, якщо ця робота є основною, та значення 1, якщо це є робота за сумісництвом. Клас міс-

тить такі методи: конструктор за замовчуванням, конструктор з двома параметрами; `podatok()` – метод який обчислює податок працівника за встановленим правилом; `zarplatnia()` – метод який обчислює "чисту" зарплатню працівника.

Примітка. Якщо серед методів класу є такі, що мають тип `AnsiString`, наприклад,

```
AnsiString info()
{ return " Зарплатня =" + FloatToStr(zarplatnia()); }
```

тоді першою у файлі необхідно записати директиву

```
#include <vcl.h>
```

Запишемо у файлі `Prac.cpp` визначення (реалізацію) методів класу `podatok()` та `zarplatnia()` (після всіх стандартних директив файла):

Файл Prac.cpp

```
#pragma hdrstop
#include "Prac.h" // Автоматичне долучення файлів визначення класу
// Реалізація методу обчислювання податку
double pracivnik::podatok()
{ double nal;
  switch(vid)
  { case 0: if(oklad<=300) nal=0; else nal=oklad*0.15; break;
    case 1: nal=oklad*0.2;
  }
  return nal;
}
// Реалізація методу обчислювання зарплатні
double pracivnik::zarplatnia()
{ return oklad - podatok();
}
```

Запишемо у головному файлі проекту `Unit1.cpp` код програми для кнопки "Обчислити" та долучимо до нього файл визначення класу `Prac.h`.

Головний файл проекту Unit1.cpp

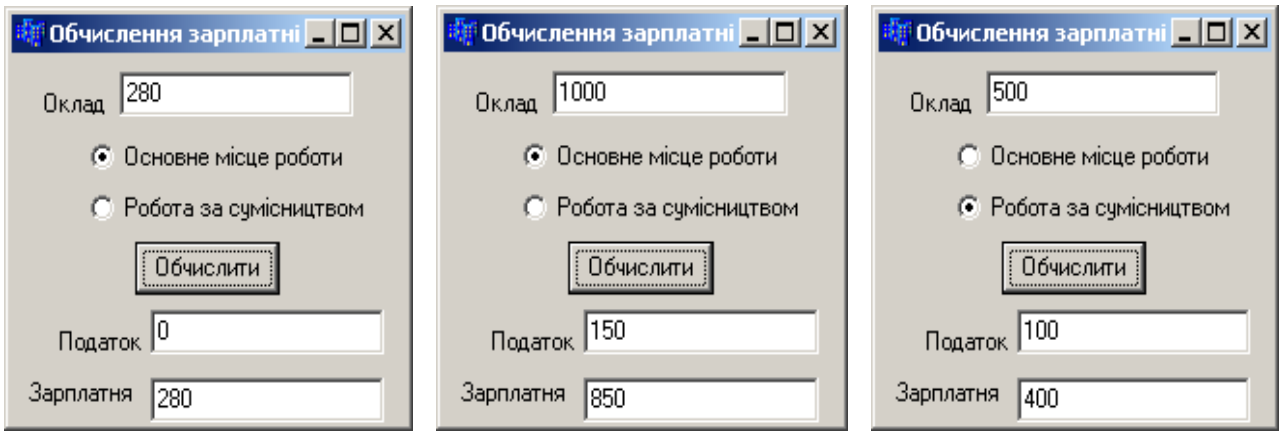
```
#include <vcl.h>
// Долучення файла оголошення та визначення методів класу
#include  "Prac.h"
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
```

```
//--------- Відгук на кнопку «Обчислити» --------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int k;
  if(RadioGrooup1->ItemIndex == 0) k=0; else k=1;
  pracivnik a(StrToFloat(Edit1->Text), k); // Створення об'єкта a
  Edit2->Text = FloatToStr(a.podatok());
  Edit3->Text = FloatToStr(a.zarplatnia());
}
```

У програмному коді для події `OnClick` кнопки "Обчислити" оголошено змінну об'єкта класу **a**, поля якої задаються за допомогою конструктора з двома параметрами. Для цього об'єкта викликатимуться методи `podatok()` та `zarplatnia()`, а їхні результати виводитимуться до текстових полів.

На формі розташовано, окрім текстових полів (компонентів `Edit`) для введення значення окладу певного працівника та виведення результатів, ще й компонент `RadioGroup` для обирання виду діяльності (значення поля `vid`).

Можливі варіанти робочого вигляду форми при виконанні проекту:



Приклад 14.3. Створити клас для динамічної структури, яку називають черга. Структура містить два інформаційні поля: прізвище студента і екзаменаційний бал з інформатики. Клас матиме такі методи: конструктор, долучення елемента до черги, вилучення елемента з черги, переглядання елементів черги.

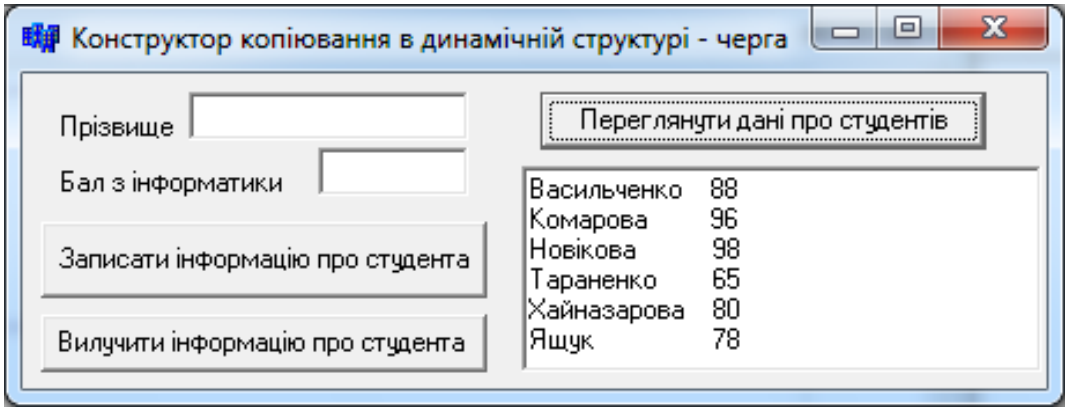
Текст програми:

```
struct listec                          // Оголошення структури типу черга
{ AnsiString Fio;  int num;
  listec *next;
};
class queue                            // Оголошення класу
{ private:
   listec *curr,*first,*last;  // Поля класу – вказівники на поточний,
                               // перший та останній елементи класу
 public:
  queue(){curr=first=last=NULL;}  // Конструктор класу
  void add(String fio, int d);    // Долучення елементів до черги
```

```
  void del();                    // Вилучення елемента з черги
  AnsiString view();             // Переглядання змісту черги
};
// Реалізація функції долучення елемента до черги
void queue::add(String fio,int d)
{ curr=new listec;   // Виділення пам'яті під новий елемент черги
  curr->Fio=fio;     // Занесення до нового елемента прізвища
  curr->num=d;       // Занесення до нового елемента значення балу
  curr->next=0; // Після нового елемента інших елементів покищо немає
  if (first==0)      // Якщо черги ще нема,
   first=curr;       // то новий елемент стає першим,
  else               // інакше елемент долучається після останнього,
   last->next=curr; // записуючи в last адресу нового елемента
  last=curr;         // Тепер новий елемент стає останнім
}
// Реалізація функції вилучення першого елемента з черги
void queue::del()
{ curr=first;
  if (first==0)      // Якщо черга є порожня,
   { ShowMessage("Черга порожня!"); // виводиться повідомлення
     return;         // і виконання переривається
   }
  first=first->next;// first указує на наступний елемент
  delete curr;       // Знищення елемента, який вилучається
}
// Реалізація функції переглядання елементів черги
AnsiString queue::view()
{ AnsiString s="";
  if (curr==0)
   { ShowMessage("Черга порожня!"); s=""; return s; }
  while (curr!=0)
   { s=s+curr->Fio+" "+IntToStr(curr->num)+"\r\n";
     curr=curr->next;
   }
  return s;
}
queue stt; // Створення об'єкта класу
// Кнопка  долучання до черги нового елемента (інформації про студента)
void __fastcall TForm1::Button1Click(TObject *Sender)
{ String fio=Edit1->Text;
  int d=StrToInt(Edit2->Text);
  stt.add(fio,d);
  Edit1->Text="";
  Edit2->Text="";
  Edit1->SetFocus();
}
```

```
// Кнопка  переглядання даних черги (інформації про студентів)
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Memo1->Clear();
  Memo1->Lines->Add(stt.view());
}
// Кнопка  вилучання елемента (інформації про студента) з черги
void __fastcall TForm1::Button3Click(TObject *Sender)
{ stt.del();
}
```

Форма з результатами роботи програми:



14.6 Деструктори

Деструктор – це особлива форма методу класу, який застосовується для звільнення пам'яті, зайнятої об'єктом. Деструктор за суттю є антиподом конструктора. Якщо конструктор – це функція, яка допомагає будувати (конструювати) об'єкт, то деструктор являє собою функцію, яка допомагає знищувати об'єкт, тобто звільняти від нього пам'ять. Деструктор викликається автоматично, коли об'єкт виходить за межі області видимості.

Деструктор при визначенні класу має такий формат:

~ <*ім'я класу*> () { <*оператори деструктора*> }

Розглянемо *основні властивості* деструктора.

✓ Ім'я деструктора розпочинається з тільди (~), безпосередньо за якою йде ім'я класу.

✓ Деструктор – це метод, який *виконується автоматично*.

✓ Конструктор *не повертає жодного значення*, навіть типу `void`.

✓ Деструктор *не має аргументів*.

✓ Вказівник на деструктор визначити не можна.

✓ Деструктор *не успадковується*.

✓ Якщо деструктор явно не визначено, *компілятор автоматично створює порожній деструктор*.

✓ *По завершенні програми об'єкти усіх класів знищуються* з пам'яті комп'ютера навіть тоді, коли деструктор явно не визначено.

У попередніх програмах об'єкти створювалися після їхнього оголошення. По завершенні програми C++ автоматично викликав деструктор для кожного об'єкта, хоча його не було явно визначено у програмі.

Деструктор, окрім деініціалізації об'єктів, може виконувати деякі дії, наприклад, виведення остаточних значень елементів даних класу, що буває зручно при налагодженні програми. Як приклад, визначимо клас `worker` з деструктором та конструктором:

```
class worker
{ private:                    // Приховані поля класу
  char name[64] ;
  long worker_id;
  float salary;
 public:                     // Загальнодоступні методи класу
worker(char *iname, long iworker_id, float isalary); // конструктор
  void ~worker(void);        // Деструктор
  void show_worker(void);  // Функція виведення інформації
};
```

Реалізація деструктора для класу `worker` (поза класом):

```
void worker::~worker(void)
{ cout << "Знищення об'єкта для " << name << endl; }
```

Цей деструктор у консольному режимі виведе на екран значення елемента класу `name` та повідомлення про те, що C++ знищує цей об'єкт. Наприклад, створимо об'єкт для визначеного вище класу:

```
void main(void)
{  worker worker("Василенко", 777, 10101.0);
   worker.show_worker();
}
```

У результаті виконання цієї програми на екрані побачимо:

```
Прізвище: Василенко
Код робітника: 777
Зарплатня: 10101.00
Знищення об'єкта для Василенко
```

Явно розміщувати деструктор у класі треба у разі, якщо об'єкт містить вказівники на пам'ять, виділену динамічно, – інакше при знищуванні об'єкта пам'ять, на яку посилались його поля-вказівники, не буде позначено як звільнену.

```
class CMyClass
{ public:
    CMyClass();       // Конструктор класу CMyClass
    ~CMyClass();      // Деструктор класу CMyClass
  private:
   int MyInt;         // Змінна типу int (ціле число)
   int *point;        // Змінна типу вказівник на int (ціле число)
};
```

```
CMyClass::CMyClass() // Конструктор
{ MyInt = 10;        // На етапі ініціалізації об'єкта класу CMyClass
                     // присвоюється змінній цього об'єкта MyInt значення 10
  point = new int;   // Виділення пам'яті під ціле число для вказівника
  *point = 20;       // Записування у виділену пам'ять числа 20
}

CMyClass::~CMyClass() // Деструктор
{  MyInt = 0;       // Об'єкт класу CMyClass фактично припинив своє існування,
                    // але надамо змінній класу MyInt значення 0
   delete point;    // Використаємо вказівник на число для того,
                    // щоби звільнити пам'ять, яку виділено під це число
                    // (Автоматично деструктор не виконує цього!)
}
```

Приклад 14.4 Створимо клас – "два резистори, які з'єднані паралельно" з полями номіналів *r*1 та *r*2 (див. приклад 4.7). Крім того, явно визначимо деструктор в оголошенні класу та обчислимо сумісний опір цих резисторів.

Розв'язок. У програмі використовуватимемо різні способи визначення конструкторів: без параметрів (конструктор 1), з одним (конструктор 2) та з двома параметрами (конструктор 3). Програму розмістимо в двох файлах `Unit1.h` та `Unit1.cpp`, призначення операторів пояснюватимемо коментарями.

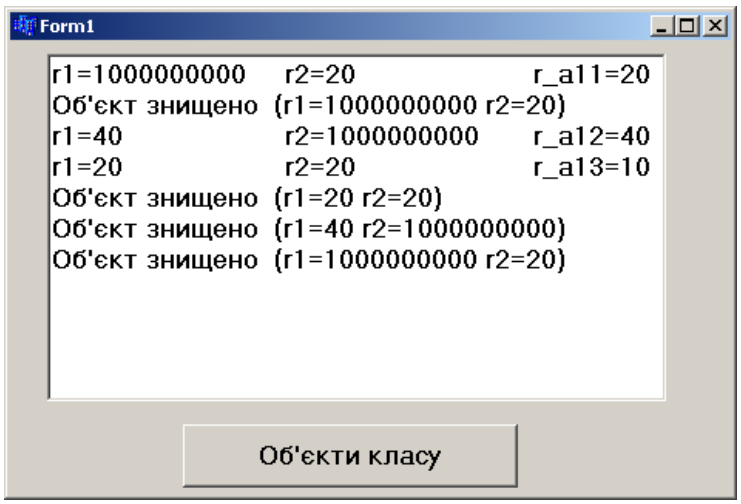
Файл Unit1.h

```
// Оголошення класу
class resistor    // Клас – два паралельних резистора
{ private:
   float r1,r2;   // Поля класу – значення номіналів резисторів
  public:
   resistor(float iR1):r1(iR1),r2(1e9) {}  // Конструктор 1
   resistor():r1(1e9),r2(20) {}            // Конструктор 2
   resistor(float iR1,float iR2):r1(iR1),r2(iR2){};//Конструктор 3
   float comr(); // Метод класу – обчислення сумісного опору двох резисторів
   AnsiString resistor::Info();// Метод класу – формування рядка
                               // зі значеннями номіналів резисторів
~ resistor(void); // Деструктор
};

// Опис реалізації функцій (методів) класу
float resistor::comr()
{ return r1*r2/(r1+r2); }

AnsiString resistor::Info()
{ return "r1="+FloatToStr(r1)+" r2="+FloatToStr(r2); }

resistor::~ resistor(void) // Реалізація деструктора
{ Form1->Memo1->Lines->Add("Об'єкт знищено  ("+ Info()+")"); }
```

Файл Unit1. cpp

```
//---------  Відгук на кнопку «Об'єкти класу»  --------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ resistor a11;          // Створення об'єкта класу a11 конструктором 1
  float ra1=a11.comr(); // Обчислення сумісного опору для об'єкта a11
  // Виведення результатів для об'єкта a11
  Memo1->Lines->Add(a11.Info()+"    r_a11="+FloatToStr(ra1));
  a11.~ resistor();      // Знищення об'єкта a11 деструктором
  resistor a12(40);      // Створення об'єкта класу a12 конструктором 2
  float ra2=a12.comr(); // Обчислення сумісного опору для об'єкта a12
  // Виведення результатів для об'єкта a12
  Memo1->Lines->Add(a12.Info()+"    r_a12="+FloatToStr(ra2));
  resistor a13(20,20);  // Створення об'єкта класу a13 конструктором 3
  float ra3=a13.comr(); // Обчислення сумісного опору для  об'єкта a13
  // Виведення результатів для об'єкта a13
  Memo1->Lines->Add(a13.Info()+"    r_a13="+FloatToStr(ra3));
}
```

Результати роботи програми – значення загального опору двох паралельних резисторів для трьох об'єктів, створених за різними конструкторами та з явним використанням деструктора, – наведено на зображенні форми проекту:



Як бачимо, після роботи з першим об'єктом (a11) викликається функція-деструктор, яка знищує об'єкт та виводить повідомлення про цю дію. Наприкінці програми деструктор автоматично звільнює пам'ять від усіх об'єктів програми (у зворотному порядку), у тому числі повторює повідомлення про об'єкт a11.

14.7 Успадкування

Успадкування (inheritance) – це механізм, за допомогою якого один об'єкт може набувати властивості іншого. Точніше, об'єкт може успадковувати основні властивості іншого об'єкта та набувати нових рис, які характерні лише для нього. Успадкування є дуже важливим, оскільки воно дозволяє підтримувати принцип *ієрархії класів* (hierarchical classification). Наприклад, клас мобільних

телефонів є підкласом класу “Телефон”, який, у свою чергу, входить до ще більшого класу “Електрозв’язок”. Разом з тим, клас “Електрозв’язок” є підкласом класу “Способи зв’язку”, до складу якого, крім електрозв’язку, входять супутниковий зв’язок, радіозв’язок, поштовий зв’язок тощо (рис. 14.1). Застосування ієрархії класів дозволяє керувати великими потоками інформації. Прикладом подібної ієрархії є системи класифікації в ботаніці, зоології тощо.

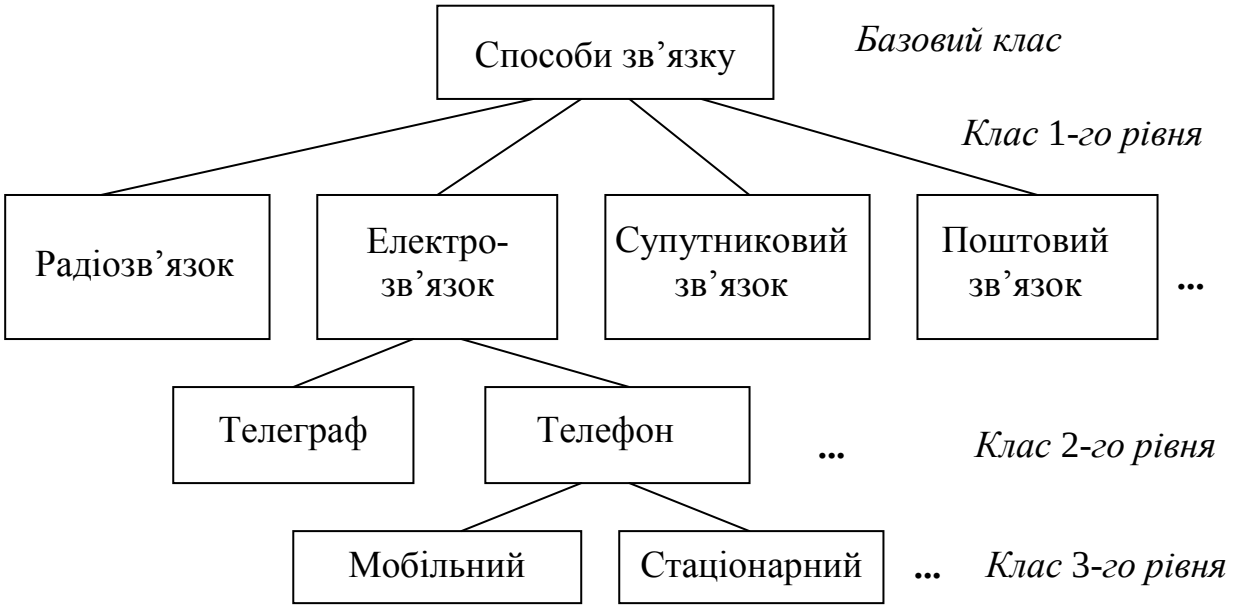


Рис. 14.1. Ієрархія класів “Способи зв’язку”

В об’єктно-орієнтованому програмуванні *успадкування* – це процес створювання нових класів, які називають *похідними класами* (*нащадками*) на базі вже існуючих *батьківських* (*базових*) *класів*. Похідний клас успадковує всі можливості батьківського (базового) класу, але може бути удосконаленим за рахунок *змінювання* існуючих методів і *долучення* нових власних полів та методів. Батьківський клас (чи клас вищого рівня) при цьому залишається незмінним. Похідний клас (клас нижчого рівня), у свою чергу, сам може слугувати за батьківський.

Найважливішою позитивною якістю успадкування в ООП є те, що воно дає можливість уникати повторювань програмного коду для кожного об’єкта, адже спільний для множини подібних класів код може бути винесено до методів їхнього спільного батьківського класу. Це є доволі зручний спосіб, оскільки програміст може використовувати класи, створені будь-ким іншим, без модифікації коду, лише створюючи похідні класи.

Успадкування буває простим і множинним. *Простим* називається успадкування, за якого похідний клас має один батьківський клас. *Множинне* успадкування означає, що клас має кілька батьківських класів, і застосовується для того, щоб забезпечити похідний клас їхніми властивостями. При застосовуванні множинного успадкування треба ретельно стежити за тим, щоби похідний клас не успадкував поля чи методи з однаковими іменами, але різні за змістом.

Для *створювання похідного класу* використовують ключове слово `class`, після якого записують ім'я нового класу, двокрапку (`:`), *ключ доступу класу* (`public`, `private`, `protected`), а потім зазначають ім'я батьківського класу:

```
class
<ім'я_похідного_класу>:[<ключ_доступу>]<ім'я_батьківського_класу>
        { <тіло_класу> };
```

Вище розглядались лише специфікатори доступу `private` (закритий) та `public` (відкритий), які застосовуються до полів класу. Однак члени базового класу, оголошені як `private`, у похідному класі є недоступні незалежно від ключа доступу. Звертання до них може відбуватися лише через методи базового класу. Для базових класів можливе використання ще одного специфікатора – `protected` (захищений), який для поодиноких класів, що не входять до ієрархії, означає те ж саме, що й `private`.

Ключ доступу `public` означає, що всі відкриті й захищені члени базового класу стають такими для похідного класу. Ключ доступу `private` означає, що всі відкриті й захищені члени базового класу стають закритими членами для похідного класу. Ключ доступу `protected` означає, що всі відкриті й захищені члени базового класу стають захищеними членами похідного класу.

Розглянемо успадкування класів на прикладі.

Приклад 14.5. Створити клас – "два дійсних числа", які можуть відображати значення висоти та ширини геометричної фігури. На базі цього класу створити два похідні класи: "прямокутник", який обчислює площу прямокутника, та "циліндр", який обчислює площу бокової поверхні циліндра. Для класу "прямокутник" створити похідний клас "паралелепіпед", який обчислює об'єм паралелепіпеда з використання параметра – значення висоти. Ієрархію цих класів наведено на рис 14.2.

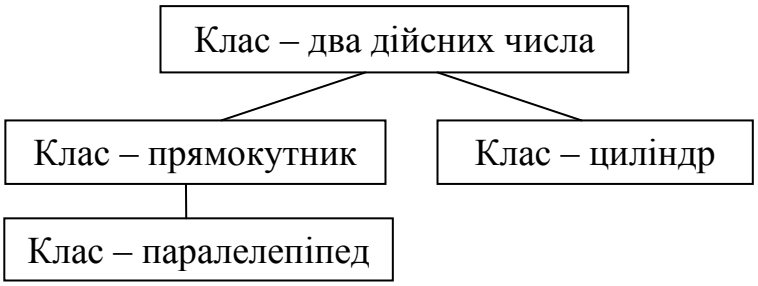


Рис. 14.2. Ієрархія класів для прикладу 14.5

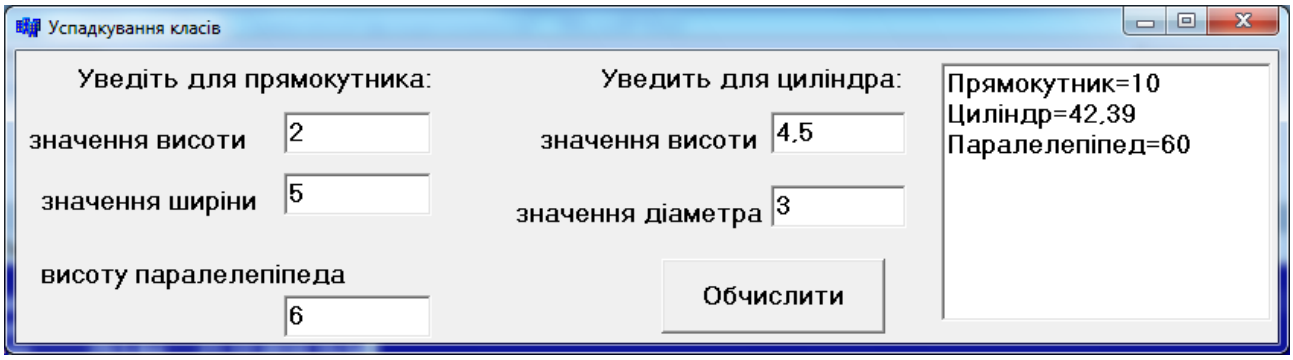
Код програми:

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------
class areal_r_c  // Оголошення батьківського класу «два дійсних числа»
{ protected:     // Забезпечення доступу до даних похідного класу
    float  heigth, width ;
};
// Оголошення похідного класу "прямокутник"
class rectangle : public areal_r_c
{ public:
   rectangle(float h, float w);
   float arear();
};
// Реалізація методів похідного класу "прямокутник"
rectangle::rectangle(float h, float w)   // Конструктор
{ heigth=h;  width=w;  }
float rectangle::arear()                 // Обчислення площі
{ return heigth*width; }

// Оголошення похідного класу "циліндр"
class cylinder : public areal_r_c
{ public:
   cylinder (float h, float w);
   float areac();
};
// Реалізація методів похідного класу "циліндр"
cylinder::cylinder(float h, float w)     // Конструктор
{ heigth=h;  width=w; }
float cylinder::areac() // Обчислення бокової поверхні циліндра
{ return 3.14*width*heigth; }

// Оголошення класу "паралелепіпед", похідного від класу "прямокутник"
class paral : public rectangle
{ public:
   float hv;
   paral(float h, float w, float hh);
   float areap();
};
// Реалізація методів похідного класу "паралелепіпед"
paral :: paral(float h, float w, float hh) : rectangle(h, w)
{ hv=hh; }                               // Конструктор
float paral::areap()  // Обчислення об'єму паралелепіпеда з використанням
{ return rectangle::arear()*hv; } // методу arear батьківського класу

//-----  Відгук на кнопку "Обчислити"--------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ // Створення об'єктів класів
 rectangle b(StrToFloat(Edit1->Text),StrToFloat(Edit2->Text));
 cylinder c(StrToFloat(Edit3->Text),StrToFloat(Edit4->Text));
```

```
 paral d(StrToFloat(Edit1->Text),StrToFloat(Edit2->Text),
                                 StrToFloat(Edit5->Text));
 // Виведення результатів обчислень об'єктів
 Memo1->Lines->Add("Прямокутник = "+FloatToStr(b.arear()));
 Memo1->Lines->Add("Циліндр = "+FloatToStr(c.areac()));
 Memo1->Lines->Add("Паралелепіпед = "+FloatToStr(d.areap()));
}
```

Форма з результатами обчислень має вигляд:



14.8 Поліморфізм

При успадкуванні деякі методи класу мають можливість бути замінені на інші. Так, батьківський клас способів зв'язку (див. рис. 14.1) матиме узагальнений метод – спосіб передавання інформації. У похідних класах цей метод буде уточнено: радіозв'язок – це передавання радіосигналів від радіостанції до радіоприймачів, поштовий зв'язок – це перевезення транспортом поштових відправлень (посилок, бандеролей, листів тощо), супутниковий зв'язок – це передавання сигналів на супутники та прийом цих сигналів супутниковими антенами. Отже, одне ім'я методу використовується для розв'язання декількох схожих, але технічно різних задач. Таке змінювання змісту методу називається поліморфізмом. Взагалі *поліморфізм* (polymorphism) (від грецької – polymorphos) – це здатність об'єкта змінювати форму (poly означає багато, а morphism має відношення до змінювання форми). Отже поліморфний об'єкт, являє собою об'єкт, який може набувати різноманітних форм.

Мета поліморфізму в об'єктно-орієнтованому програмуванні – використання одного імені для схожих дій (методів) класу. Виконання кожної конкретної дії буде визначено типом даних. У різних мовах програмування поліморфізм реалізовано різноманітними засобами. Наприклад, у Pascal та C++ його реалізовано за допомогою механізму віртуальних функцій. Разом з тим, у мові C++ поліморфізм підтримується недостатньо, наприклад, обчислення абсолютного значення змінної можна виконати трьома функціями: `abs()`, `labs()` та `fabs()`. Ці функції обчислюють та повертають абсолютне значення змінних цілого, довгого цілого та дійсного типів відповідно. У Pascal така задача виконується однією функцією `abs()`. У мові C++ вибір конкретної функції для цієї задачі здійснює програміст відповідно до типу даних.

Поліморфізм може також застосовуватись до операторів. Фактично в усіх мовах програмування обмежено використовується поліморфізм в арифметичних операціях. Так, у мові C++, символ плюс (+) використовується для додавання цілих, довгих цілих, дійсних чисел, а також для символьних змінних та рядків. У такому разі компілятор автоматично визначає, який тип арифметики слід застосувати.

При роботі з типом даних `class` у C++ є можливість застосовувати механізм поліморфізму, тобто одне ім'я методу класу використовувати для множини різноманітних дій. Перевагою поліморфізму є те, що він допомагає зменшити складність програм. Вибір конкретної дії, відповідно до ситуації, покладено на компілятор, і програмістові не треба вибирати самому. Необхідно лише пам'ятати та використовувати загальний інтерфейс. Приклад з функцією обчислення абсолютного значення змінної, показує, як за наявності трьох імен у мові C++ замість одного, будь-яка задача стає більш складною, аніж це дійсно потрібно.

Щоб використовувати поліморфізм, треба забезпечити виконання певних умов. По-перше, всі похідні класи мають бути нащадками одного й того самого базового класу. По-друге, методи, які забезпечують поліморфізм (назвемо їх поліморфними методами), мають бути оголошені в батьківському класі як віртуальні. Віртуальний (virtual) – означає видимий, але неіснуючий у реальності. *Віртуальна функція* – це функція базового класу, перед прототипом якої стоїть ключове слово `virtual` за форматом:

`virtual` <*тип*> <*ім'я_функції*> (<*список_параметрів*>);

Віртуальні функції не визначаються у батьківському класі, до оголошення цього класу записують лише прототипи цих функцій із ключовим словом `virtual` перед ними. Похідний клас перевизначає ці функції, пристосовуючи їх для власних потреб. І, якщо функції було оголошено як віртуальні, вони залишатимуться віртуальними й в усіх похідних класах. Проте, зазвичай, надають перевагу зберіганню цього специфікатора і в класах-нащадках для більшої зрозумілості суті цих класів.

Базовий клас здебільшого буває *абстрактним класом*, об'єкти якого ніколи не буде реалізовано. Такий клас існує з єдиною метою – бути батьківським відносно похідних класів, об'єкти яких буде реалізовано, для створювання ієрархічної структури. Наприклад, клас "Способи зв'язку" має метод "Засоби передавання інформації", який не може бути реалізовано для об'єктів цього класу, а в похідних класах цей метод має конкретне значення: для класу "Радіозв'язок" – це радіосигнал, для класу "Поштовий зв'язок" – це транспорт для перевезення поштових відправлень тощо. Але, якщо об'єкти батьківського (абстрактного) класу не призначені для реалізації, то в який спосіб захистити базовий клас від використання не за призначенням? – Захистити його треба програмно. Для цього достатньо ввести у клас хоча б одну *суто віртуальну функцію* (pure virtual function).

Для суто віртуальної функції використовується формат:

`virtual` <*тип*> <*ім'я_функції*> (<*список_параметрі в* >) **= 0;**

Ключовою частиною цього оголошення є присвоєння суто віртуальній функції значення нуль. Це повідомляє компіляторові, що в батьківському класі немає тіла функції. Якщо функцію задано як суто віртуальну, то це означає, що вона обов'язково повинна бути заміненою в кожному похідному класі, інакше при компіляції виникне помилка. Отже, створення суто віртуальних функцій – це гарантія того, що похідні класи забезпечать їхнє перевизначення. Якщо клас містить хоча б одну суто віртуальну функцію, то його називають *абстрактним класом*. Оскільки в абстрактному класі є хоча б одна функція, в якої відсутнє тіло функції, технічно такий клас не є повністю визначений, і для нього неможливо створити жодного об'єкта. Тому абстрактні класи можуть бути лише похідними.

Основні концепції поліморфізму в мові програмування С++:

✓ поліморфізм – це властивість об'єкта змінювати форму під час виконання програми;

✓ для створення методів, які є поліморфними, у програмі необхідно використовувати віртуальні (`virtual`) функції;

✓ якщо об'єкти батьківського (абстрактного) класу не призначені для реалізації, необхідно ввести у клас хоча б одну суто віртуальну функцію;

✓ будь-який клас, похідний від батьківського, має можливість використовувати чи перевантажувати віртуальні функції;

✓ для створення поліморфного об'єкта в С++ слід використовувати вказівник на об'єкт батьківського класу;

✓ поліморфізм спрощує програмування та створення поліморфних об'єктів і зменшує складність програм.

Розглянемо механізм поліморфізму на прикладах.

Приклад 14.6 Створити поліморфний клас – телефон, який відображує (імітує) телефонні операції: набирання номера, дзвінок, роз'єднання та індикацію зайнятості лінії зв'язку. Передбачити у програмі можливість імітувати, за бажанням, роботу дискового, кнопкового чи платного телефону (25 копійок, щоб подзвонити). Тобто об'єкти класу мають бути поліморфними – від одного дзвінка до іншого об'єкт-телефон має змінювати форму (своє призначення).

Розв'язок. Створимо батьківський клас `phone` – дисковий телефон з полем `number` – номером телефону і методами: `dial` – набирання номера, `ring` – дзвінок, `answer` – очікування відповіді, `hangup` – роз'єднання. Далі створимо два похідні класи: `touch_tone` – кнопковий телефон та `pay_phone` – платний телефон, причому в цих класах будуть визначені свої власні методи `dial`.

Код програми з виведенням результатів у консольному режимі має такий вигляд:

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
```

```
class phone       // Оголошення батьківського класу – дисковий телефон
{ protected:
   char number[13] ;
  public:
   virtual void dial (char *number)    // Віртуальна функція
   { cout<<"Набирання номера "<<number<< endl;; }
   void answer(void)
   { cout << "Очікування відповіді" << endl; }
   void hangup(void)
   { cout << "Дзвінок виконано – покладіть слухавку " << endl; }
   void ring(void)
   { cout << " Дзвінок, дзвінок, дзвінок " <<endl; }
   phone(char *number)
   { strcpy(phone::number, number); };         // Конструктор
};
// Оголошення похідного класу – кнопковий телефон
class touch_tone : phone
{ public:
   void dial(char * number)
   { cout << "Пік, пік. Набирання номера" << number << endl; }
   touch_tone(char *number):phone(number){ } // Конструктор
};
// Оголошення похідного класу – платний телефон
class pay_phone: phone
{ private:
   int amount;
  public:
    void dial(char *number)
    { cout<<"Будь ласка, сплатіть "<<amount<<" копійок"<<endl;
      cout<<"Набирання номера "<<number<<endl;
    }
    pay_phone(char *number, int amount):     // Конструктор
    phone(number) { pay_phone::amount = amount; }
};

// Головна програма – створення та робота з об'єктами класу
void main(void)
{ // Створення об'єкта  rotary – дисковий телефон
  phone rotary("201-555-1212");
  rotary.dial("602-555-1212");
  // Створення  об'єкта  home_phone – кнопковий телефон
  touch_tone home_phone("555-1212");
  home_phone.dial("555-1212");
  // Створення об'єкта  city_phone – платний телефон
  pay_phone city_phone("702-555-1212", 25);
  city_phone.dial("212-555-1212");
  cout << "      Поліморфні об'єкти: " << endl;
  // Створення об'єкта-вказівника на  батьківський клас – дисковий телефон
  phone *poly_phone = &rotary;
```

```
  poly_phone->dial("818-555-1212");
  // Змінення форми об'єкта на кнопковий телефон
  poly_phone = (phone *) &home_phone;
  poly_phone->dial("303-555-1212");
  // Змінювання форми об'єкта на платний телефон
  poly_phone = (phone *) &city_phone;
  poly_phone->dial("212-555-1212");
  getch();
  return ;
}
```

Результати виконання програми:

```
Набирання номера 602-555-1212
Пік, пік. Набирання номера 212-555-1212
Будь ласка, сплатіть 25 копійок
Набирання номера 212-555-1212
   Поліморфні об'єкти:
Набирання номера 818-555-1212
Пік, пік. Набирання номера 303-555-1212
Будь ласка, сплатіть 25 копійок
Набирання номера 212-555-1212
```

Наведені класи телефону мають однакову назву, але різну за виконанням функцію `dial()`. Цю функцію було визначено як віртуальну (з ключовим словом `virtual`) у батьківському класі `phone`.

Для створення поліморфного об'єкта у програмі використано вказівник на об'єкт батьківського класу (`phone *poly_phone`). Для змінювання форми об'єкта цьому вказівникові надано адресу об'єкта похідного класу:

```
poly_phone = (phone *) &home_phone;
```

Вираз у круглих дужках (`phone *`), записаний за оператором присвоєння, є операцією зведення типів, яка сповіщає компілятору C++, що вказівнику на змінну одного типу (`phone`) треба надати адресу змінної іншого типу (`home_phone`). Оскільки програма присвоює вказівнику об'єкта `poly_phone` адреси різних об'єктів, то цей об'єкт може змінювати свою форму, а, отже, він є поліморфним. Згідно з програмою, об'єкт `poly_phone` змінює форму з дискового телефону на кнопковий, а після цього – на платний.

Приклад 14.7 Побудувати батьківський клас `Tvirob` з полями: назва виробу, кількість випробувань виробу на якість, кількість виробів, які успішно пройшли випробування, кількість таких з них, які виявилися нестандартними. Метод класу визначає якість за формулою:

Q = *кількість випробувань виробу* / *кількість успішних випробувань.*

Побудувати похідний клас `TPvirob`, який крім батьківських полів має додаткове поле (***P***) – кількість виробів, які не відповідають стандарту (за розміром, кольором тощо) та визначає якість виробу за новою формулою:

Qp = ***Q*** – 2* ***P*** / *кількість випробувань виробу на якість*.

Оголосити та визначити елементи класів в окремих файлах. У головному модулі створити об'єкти з вказівником та без нього і вивести інформацію у вікно форми. При написанні програми застосувати поняття поліморфізму класів.

Розв'язок. У батьківському класі `Tvirob` оголосимо поля: `NAME` – назва виробу, `kolp` – кількість випробувань виробу на якість (у розділі `protected`), `kolpr` – кількість виробів, які успішно пройшли випробування. Методи батьківського класу – конструктор, `rachest()` – віртуальна функція розрахунку якості (у розділі `protected`), `Info()` – функція формування рядка інформації (не віртуальна).

Клас-нащадок `TPvirob` має одне, додаткове до батьківських, поле `kolV` – кількість таких виробів, які успішно пройшли випробування, але виявилися нестандартними. Методи класу-нащадка – конструктор та віртуальна функція `rachest(void)` для розрахунку якості за новою формулою, яка перекриває функцію якості батьківського класу.

Текст програми:

Файл Unit2.h – оголошення батьківського класу

```
#ifndef Unit2H
#define Unit2H
#include <vcl.h>
//---------------------------------------------------------------
class Tvirob
{ private:
   int kolpr;
   AnsiString NAME;
 protected:
   virtual float rachest(void); // Віртуальна функція
   int kolp;
 public:
   Tvirob (AnsiString iNAME,int ikolp, int ikolpr); // Конструктор
   AnsiString Info(void);       // Без перекриття
};
#endif
```

Файл Unit2.cpp – визначення методів батьківського класу

```
#pragma hdrstop
#include "Unit2.h"
//---------------------------------------------------------------
#pragma package(smart_init)
Tvirob::Tvirob(AnsiString iNAME,int ikolp,int ikolpr)//Конструктор
{ NAME=iNAME;kolp=ikolp; kolpr=ikolpr;}
float Tvirob::rachest()          // Функція розрахунку якості Q
{ return  1.0*kolpr/kolp; };
AnsiString  Tvirob::Info()       // Функція формування рядка-інформації s
{ AnsiString s=NAME+" Q = "+FloatToStrF(rachest(),ffFixed,6,2);
 return s;  }
```

Файл Unit3.h – оголошення похідного класу

```
#ifndef Unit3H
#define Unit3H
#include "Unit2.h"
//---------------------------------------------------------
class TPvirob : public Tvirob
{ private:
    int kolV;    // Додаткове поле нащадка
  protected:
    float rachest(void);
  public:
    TPvirob(AnsiString iNAME,int ikolp, int ikolpr, int ikolV);
};
#endif
```

Файл Unit3.cpp – визначення методів похідного класу

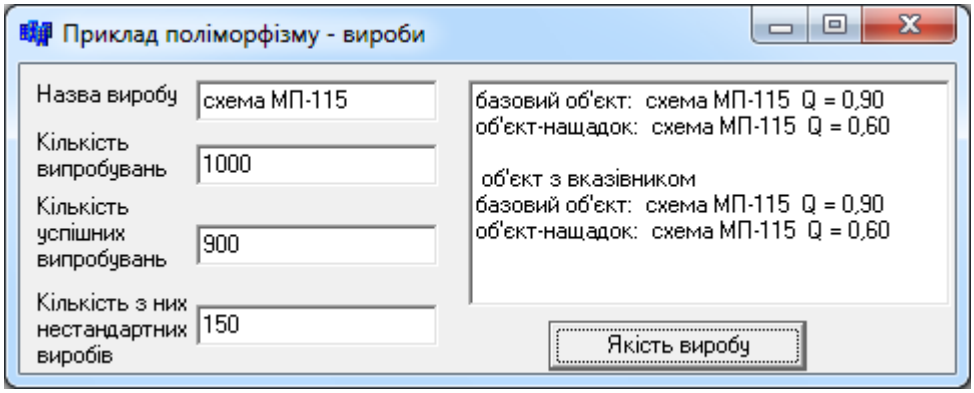
```
#pragma hdrstop
#include "Unit3.h"
//---------------------------------------------------------
#pragma package(smart_init)
// Конструктор нащадка
TPvirob::TPvirob(AnsiString iNAME,int ikolp,int ikolpr,int ikolV)
               : Tvirob (iNAME,ikolp,ikolpr)
{ kolV=ikolV;  }
float TPvirob::rachest()
{ float Q=Tvirob::rachest();
return Q-2.0*kolV/kolp; }
```

Файл Unit1.cpp – головний модуль проекту

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#include "Unit2.h"
#include "Unit3.h"
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString NAME = Edit1->Text;
  int kp = StrToInt(Edit2->Text);
```

```
  int kpr = StrToInt(Edit3->Text);
  Tvirob vr(NAME,kp,kpr);   // Об'єкт батьківського класу
  Memo1->Lines->Add("базовий об'єкт:  " + vr.Info());
  int kol = StrToInt(Edit4->Text);
  TPvirob vg(NAME,kp,kpr,kol); // Об'єкт класу-нащадка
  Memo1->Lines->Add("об'єкт-нащадок:  "+ vg.Info());
  Memo1->Lines->Add("  об'єкт з вказівником  ");
  Tvirob *p0,*p1; // Оголошення змінних об'єктів з вказівником
  p0 = new Tvirob(NAME,kp,kpr); // Об'єкт батьківського класу
  Memo1->Lines->Add("базовий об'єкт:  "+p0->Info());
  p1 = new TPvirob(NAME,kp,kpr,kol); // Об'єкт класу-нащадка
  Memo1->Lines->Add("об'єкт-нащадок : "+ p1->Info());
}
//---------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{ Memo1->Clear();
}
```

У головному модулі проекту вводитимемо дані через компоненти `Edit` та виводити інформацію до `Memo`. Створимо об'єкти батьківського класу як змінні без вказівника та з вказівником. Форма проекту з результатами обчислень для двох варіантів введених даних:



14.9 Класи та об'єкти бібліотеки компонентів

14.9.1 Ієрархія класів бібліотеки візуальних компонентів

У середовищі C++ Builder *компоненти* (наприклад: `Button`, `StringGrid`, `Edit` тощо) є спеціальними класами, які організовано у бібліотеці Visual Components Library (VCL). Класи компонентів створюються розробниками програмного забезпечення і використовуються програмістами при роботі в інтегрованому середовищі програмування C++ Builder. Кожен клас компонентів складається з таких елементів: властивості (properties), методи (methods) та події (events). *Властивості компонентів* (наприклад: `Caption`, `Top`, `Left`, `Font` тощо) – це розширення поняття поля класу, вони забезпечують доступ до даних класу не лише в програмі, а й під час створення форми проекту. *Методи* класу (на-

приклад: `Show()`, `Hide()`, `SetFocus()`, `GetParentComponent()` тощо) реалізують певну стандартну поведінку компонентів. *Події* – це спеціальні методи класу, за допомогою яких компонент повідомляє користувача про те, що над ним виконали якусь дію (наприклад: `Click`, `MouseDown`, `KeyPress` тощо) і буде виконано програмний код, передбачений програмістом. Перелік усіх елементів компонентів можна переглянути за допомогою довідкової системи Help.

Класи компонентів мають ієрархічну структуру. Наведемо фрагмент дерева класів бібліотеки візуальних компонентів C++ Builder (рис. 14.3).

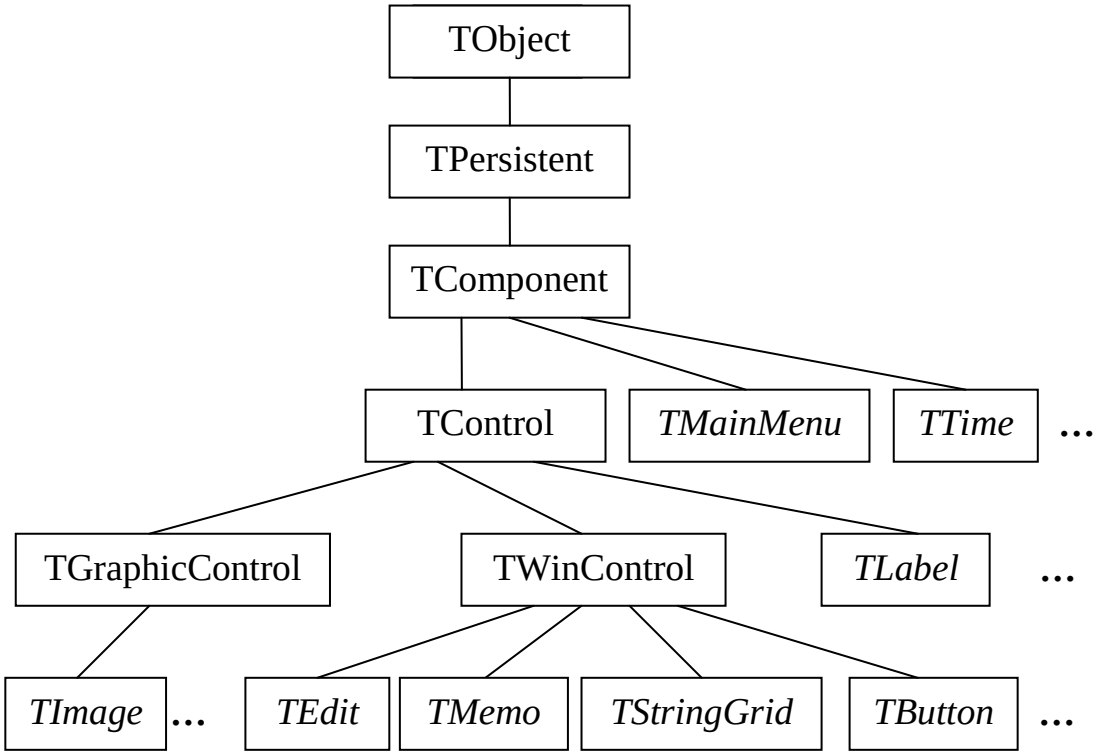


Рис. 14.3. Ієрархія класів бібліотеки VCL
(*курсивом позначено класи компонентів*)

Клас `TObject` – це базовий клас, який є спільним предком усіх класів. Він забезпечує такі методи: здібність конструктора створювати, а деструктора знищувати об'єкт класу в динамічній пам'яті, опрацювання повідомлень Windows тощо. До методів цього класу не слід звертатися зі своїх програм, оскільки вони можуть бути перевизначеними у похідних класах.

Клас `TPersistent` – похідний клас від `TObject`. Це абстрактний клас, який визначає низку методів копіювання об'єктів подібних класів, повернення власника даного об'єкта, завантаження та збереження спеціальної інформації.

Клас `TComponent` – похідний клас від `TPersistent` та батьківський клас для усіх компонентів. Цей клас забезпечує відображення компонентів та маніпуляцію з ними в редакторі форм. Не слід створювати об'єкти цього класу у своїх програмах, але можна використовувати цей клас для створювання похідних класів. Похідними від цього класу є класи невізуальних компонентів (наприклад: `TMainMenu`, `TTimer`, `OpenDialog` тощо), які є видимі як піктограми на формі під час проектування, але їх не видно під час виконання.

Клас TControl – базовий клас усіх візуальних компонентів. Цей клас встановлює властивості компонента: розташування, розміри, видимість, доступність, колір, шрифт тощо. У табл. 14.1 наведено заголовки та призначення деяких методів, притаманних об'єктам класу TControl, а отже, усім нащадкам цього класу.

Таблиця 14.1

Деякі методи класу TControl

Прототип методу	Призначення
DYNAMIC void __fastcall **Click**(void);	Викликає функцію відгуку на подію OnClick, яка відбувається при клацанні лівою кнопкою миші
DYNAMIC void __fastcall **DblClick**(void);	Викликає функцію відгуку на подію OnDblClick, яка відбувається у разі подвійного клацання лівою кнопкою миші
DYNAMIC void __fastcall **MouseDown**(TMouseButton Button, Classes::TShiftState Shift, int X, int Y);	Викликає функцію відгуку на подію OnMouseDown, яка відбувається при клацанні кнопки миші. Параметр Button визначає кнопку: ліва, середня чи права. Параметр Shift зазначає, чи було натиснуто якусь з кнопок клавіатури: Shift, Ctrl, Alt. Параметри X, Y – координати миші
DYNAMIC void __fastcall **MouseMove** (Classes::TShiftState Shift, int X, int Y);	Викликає функцію відгуку на подію OnMouseMove, яка відбувається при переміщуванні курсора миші. Параметри є аналогічні до наведених вище
DYNAMIC void __fastcall **Resize**(void);	Повідомляє про те, що розміри об'єкта змінилися
DYNAMIC void __fastcall **MouseUp** (TMouseButton Button, Classes::TShiftState Shift, int X, int Y);	Викликає функцію відгуку на подію OnMouseUp, яка відбувається при відпусканні кнопки миші. Параметри є аналогічні до параметрів методу MouseDown

Усі ці методи у класі TControl вміщено до секції protected. Здебільшого наведені методи не викликаються безпосередньо, тобто за явними звертаннями у програмах. Іноді їхні прямі виклики навіть заборонено (наприклад для методу Resize()). Проте всі вони є доступні для перекриття, на що вказує ключове слово DYNAMIC – аналог слова virtual для компонентів.

Клас TWinControl – базовий клас усіх віконних компонентів. У цьому класі введено віконний дескриптор (window handle), реалізовано здібність приймати фокус введення інформації та обслуговувати події клавіатури. У табл. 14.2 наведено деякі з методів класу TWinControl. Слід звернути увагу на використання слова virtual в оголошенні функції SetFocus замість слова DYNAMIC у переважної більшості компонентів.

Таблиця 14.2

Деякі методи класу TWinControl

Прототип методу	Призначення
`virtual void __fastcall` **`SetFocus`**`(void);`	Передає об'єктові "фокус", після чого натискання клавіш клавіатури сприйматимуться саме цим об'єктом
`DYNAMIC void __fastcall` **`KeyDown`**`(Word &Key, Classes:: TShiftState Shift);`	Викликає функцію відгуку на подію `OnKeyDown`, яка відбувається за натискання будь-якої клавіші на клавіатурі. Параметр `Key` визначає клавішу, яку натиснуто. Параметр `Shift` є такий самий, як у `MouseDown`
`DYNAMIC void __fastcall` **`KeyPress`**`( char&Key);`	Викликає функції відгуку на подію `OnKeyPress`, яка відбувається за натискання будь-якої символьної клавіші
`DYNAMIC void __fastcall` **`KeyUp`**`(Word &Key,Classes:: TShiftState Shift);`	Викликає функції відгуку на подію `OnKeyUp`, яка відбувається за відпускання будь-якої клавіші

Клас `TGraphicControl` – це абстрактний клас, похідний від `TControl`, але який, на відміну від класу `TWinControl`, не має віконного дескриптора. Цей клас використовують для графічних зображень на формі без звертань до системних ресурсів Windows. Важлива властивість цього класу – `Canvas`, яка забезпечує доступ до поверхні малювання (див. підрозд. 12.4). Похідні від цього класу також можуть обслуговувати події маніпулювання з мишею.

Завдяки важливій властивості класів – успадкуванню – можна створювати нові компоненти не з нуля, а як клас, який успадковує властивості, методи чи події батьківського класу, тобто класу, на базі якого створюється цей нащадок. Наприклад, при створюванні нової кнопки можна взяти за основу один з уже розроблених класів кнопок, наприклад клас `TButton`, й лише долучити до нього певні нові властивості чи то відкинути певні властивості та методи батьківського класу.

14.9.2 Побудова компонента-нащадка

Яскравим прикладом застосування успадкування є створення нового компонента у візуальному середовищі програмування як нащадка існуючого компонента. При цьому компоненту можна долучити нові властивості чи нові події.

Процес створювання компонента складається з послідовності етапів:

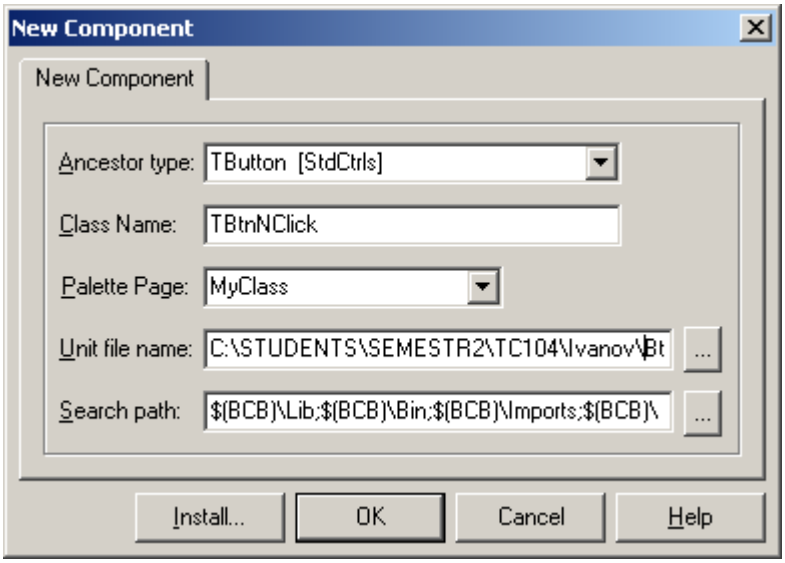
1) обирання батьківського компонента;
2) створення модуля нового компонента;
3) тестування компонента;
4) інсталяція (долучення) компонента до пакета компонентів.

Розглянемо цей процес на прикладі створення нового компонента, який запам'ятовує кількість натискань на нього та відображує цю інформацію.

Обирання батьківського компонента. Створимо власний компонент як нащадок класу `TButton`, долучивши до нього нову властивість, яка буде обчислювати кількість натискань. Назвемо новий компонент `TBtnNClick`. До компонента, окрім властивостей, звичних для `TButton`, долучимо нову властивість у вигляді додаткового цілого типу – `int NClick` – лічильник натискань.

Створення модуля нового компонента. Роботу над створенням нового компонента розпочинаємо з побудови шаблона нового класу. Для цього слід виконати такі дії.

Виконати команду **File / New / Application**. У меню Component обрати опцію New Component і в однойменному діалоговому вікні заповнити такі поля з інформацією про компонент, який створюється:



`Ancestor type` – батьківський клас нащадка, що будується (обираємо для нашого прикладу клас кнопки `TButton`);

`Class Name` – ім'я нового класу, що будується (тут напишемо `TBtnNClick`);

`Palette Page` – сторінка палітри компонентів, на яку буде розміщено новий компонент. За замовчуванням це сторінка `Samples`. Можна зазначити одну з існуючих сторінок, або ж задати нове ім'я – відповідну нову сторінку палітри буде сформовано. Задамо ім'я `MyClass`;

`Unit file name` – слід зазначити ім'я нового модуля (`unit`) та шлях до нього. В цей `unit` буде розміщено код мовою C++, потрібний для побудови нового класу. *За замовчуванням імена файлів модуля збігаються з ім'ям класу, – за винятком першої літери `T`,* – отже, якщо клас називається `TBtnNClick`, файли буде названо `BtnNClick.cpp` та `BtnNClick.h`. Щодо шляху до файла, то за замовчуванням C++ Builder встановлює шлях до власного бібліотечного каталогу LIB. *Рекомендується змінити цей шлях* і розмістити `unit` нового класу у власному каталозі, наприклад:

```
C:\STUDENTS\SEMESTR2\TC104\Ivanov\BtnNClick.cpp
```

`Search path` – тут перелічено каталоги, в яких C++ Builder шукатиме `unit` нового класу. Якщо у попередньому пункті було зазначено повний шлях

до власного каталогу, C++ Builder автоматично допише його до Search path; тоді залишиться лише перевірити, чи правильно це було зроблено.

Тепер слід натиснути кнопку ОК, – і буде утворено модуль нового компонента, який складається з двох файлів: заголовного файла BtnNClick.h та файла реалізації BtnNClick.cpp. При цьому буде відкрито вікно файла BtnClick.cpp; проте оголошення нового класу буде сформовано у заголовному файлі BtnNClick.h.

До шаблона компонента, який утворено, треба ввести певні доповнення: оголошення нових полів даних, функції доступу до них, властивості та методи. Для нашого компонента створимо нове поле NClick типу int та допишемо прототип методу Click.

Файл BtnNClick.h:

```
#ifndef BtnNClickH
#define BtnNClickH
#include <SysUtils.hpp>
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
class PACKAGE TBtnNClick : public TButton
{ private:
    int NClick; // Змінна для підрахунку кількості натискань
  protected:
  public:       // Прототип конструктора нового компонента
     __fastcall TBtnNClick(TComponent* Owner);
                // Прототип події Click
     DYNAMIC void __fastcall Click(void);
  __published:
};
#endif
```

У цьому файлі рядок

```
class PACKAGE TBtnNClk : public TButton
```

повідомляє про те, що клас TBtnNClk є нащадком класу TButton. Слово public означає, що поля й методи батьківського класу будуть доступними з екземплярів його нащадка. Поле NClick вміщено до секції private, щоб користувач не мав до нього доступу. Тепер слід долучити до класу TBtnNClick додаткові методи. У секції public нового класу вже вміщено прототип методу:

```
__fastcall TBtnNClick (TComponent* Owner);
```

Це так званий метод-конструктор об'єктів нового класу. Далі у секції public йде прототип методу Click, який перекриває метод Click класу TButton. Цей прототип має вигляд:

```
DYNAMIC void __fastcall Click(void);
```

Тепер перейдемо до файла BtnNClick.cpp. У цьому файлі розмістимо реалізацію оновленого конструктора та методу Click.

Файл BtnNClick.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include "BtnNClick.h"
#pragma package(smart_init)
//---------------------------------------------------------------------
static inline void ValidCtrCheck(TBtnNClick *)
{  new TBtnNClick(NULL);
}
//---------------------------------------------------------------------
__fastcall TBtnNClick::TBtnNClick(TComponent* Owner)
        : TButton(Owner)
{  NClick=0; // Початкове значення змінної кількості натискань
}
//---------------------------------------------------------------------
void __fastcall TBtnNClick::Click(void)
{ // Збільшення значення змінної кількості натискань на 1
      NClick++;
  // Виведення значення кількості натискань на кнопці
      Caption=Name+" ("+IntToStr(NClick)+") клацань";
  // Подальше успадкування роботи методу Click() від класу TButton
      TButton::Click();
}
//---------------------------------------------------------------------
namespace Btnnclick
{ void __fastcall PACKAGE Register()
  { TComponentClass classes[1]= {__classid(TBtnNClick)};
    RegisterComponents("MyClasses", classes, 0);
  }
}
```

Першою у цьому файлі стоїть функція `ValidCtrCheck()`, яка запобігає вміщенню до нового класу віртуальних функцій. Далі йде конструктор класу, який ініціалізує поле `NClick` значенням 0. Новий конструктор тепер виконуватиме, окрім усіх дій батьківського конструктора, ще й додаткову інструкцію `NClick=0`. У такий спосіб він *перекриває* батьківський метод. Параметр конструктора `Owner` має тип `TComponent*`, тобто є вказівником на об'єкт класу `TComponent`. Це означає, що "володарем" кнопки типу `TBtnNClick` може бути лише якийсь з компонентів C++ Builder.

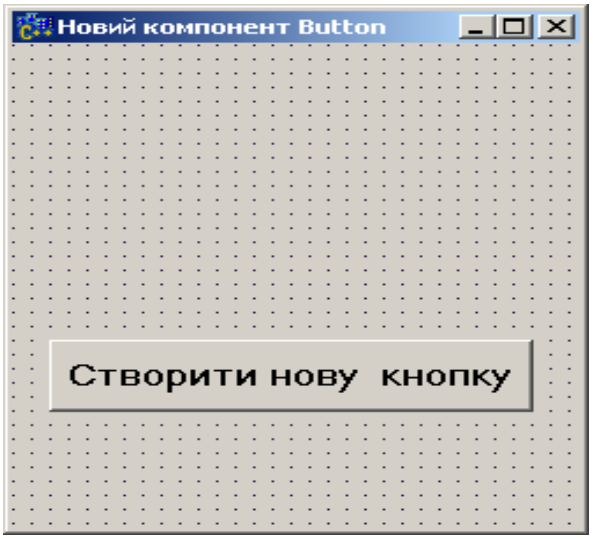
Далі треба перекрити ще один метод класу `TButton` – метод `Click`. Функція `Click()` збільшує на одиницю лічильник натискань кнопки; формує властивість кнопки `Caption` як рядок, у якому до її імені `Name` дописано кількість натискань, і потім викликає успадкований метод `TButton::Click()`. Останнє зроблено для того, щоби, за винятком відображення кількості натискань, "поводження" кнопок нового типу нічим не відрізнялось від типу `TButton`.

Останньою є директива `namespace` і функція `Register()`, потрібна для реєстрації нового класу як компонента C++ Builder. Ця функція викликає біблі-

отечну підпрограму `RegisterComponents`, перший параметр якої визначає сторінку, на яку буде вміщено новий компонент, другий – масив імен класів, які реєструються, а останній параметр має бути на 1 меншим за кількість класів у масиві. Директива `namespace` визначає константу `BtnNClick`, яка збігається з ім'ям модуля з точністю до регістра літер. У подальшому ця константа дозволяє викликати "правильну" функцію `Register` для реєстрації різних класів.

Тестування компонента. Після створення модуля нового компонента його бажано протестувати перед інсталяцією, щоби переконатися, що новий компонент працює правильно. Для цього розроблені модулі зберігають та створюють проект, який використовує модуль нового компонента. Компонент буде створено як динамічну змінну, тому що на палітрі компонентів його піктограми ще нема.

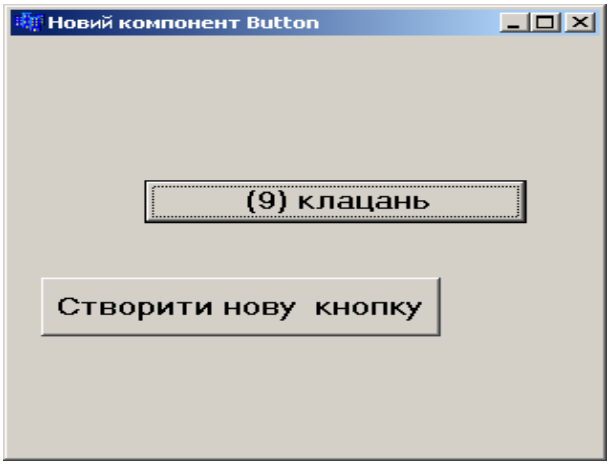
На форму проекту нанесемо тільки стандартну кнопку `Button1` для керування роботою проекту. Варіант форми для тестування:




```
#include <vcl.h>
#pragma hdrstop
#include "BtnNClick.cpp"      // Приєднуємо модуль компонента
#include "Unit1.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------
TBtnNClick *Btn;      // Оголошення динамічної змінної – об'єкта класу
//---------Відгук для кнопки "Створити нову кнопку"---------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ Btn = new TBtnNClick (Form1);
```

```
  Btn->Parent=Form1;
  // Координати розміщення нового компонента на формі
  Btn->Left=70;     Btn->Top=100;
  Btn->Width=200;   Btn->Height=30;
}
```

Власником об'єкта, що тестується, є вікно форми `Form1` (власник показано як параметр конструктора `TBtnNClick`). Властивість `Parent` створеного екземпляра класу (її задають обов'язково) також дорівнює `Form1`, а це означає, що новий компонент `Btn` буде розміщено у вікні `Form1`. Після запуску проекту та натискання на стандартну кнопку "Створити нову кнопку" на формі з'явиться новий компонент. При клацанні по цій кнопці, на ній з'явиться повідомлення про кількість клацань, як це показано на формі з результатами роботи.



Наведемо ще один приклад створення та тестування нового компонента.

Приклад 14.8. Необхідно створити новий компонент – клас `TSG` як нащадок класу `TStringGrid`. Цей компонент повинен мати такі додаткові можливості:

✓ при натисканні на клавішу <F11> його комірки заповнюватимуться випадковими цілими числами від 0 до 20;

✓ при натисканні на клавішу <Esc> комірки компонента очищуватимуся;

✓ при клацанні лівою кнопкою миші на будь-яку комірку розмір шрифту її тексту збільшуватиметься вдвічі, а при повторному клацанні – повертатиметься до початкового стану.

Розв'язок. Для реалізації останньої з названих можливостей введемо додаткове поле `flag` логічного типу `bool` – прапорець, який змінюватиме власне значення після змінювання розміру шрифту з `true` на `false` та навпаки. Для опрацювання подій клацання лівою кнопкою миші до оголошення класу внесемо прототип `Click` (див. табл. 14.1). Також до оголошення класу впишемо прототип функції `KeyPress` (див. табл. 14.2) з метою опрацювання події натискання клавіші <F11>. Обидві функції оголосимо як `DYNAMIC`, що й дозволить перекрити зміст методу базового компонента.

Наведемо коди файлів SG.h та SG.cpp.

Файл SG.h:

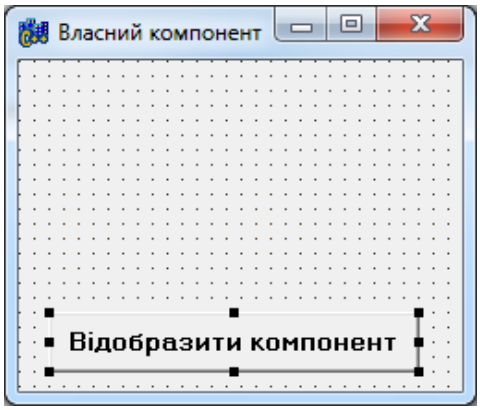
```
#ifndef SGH
#define SGH
#include <SysUtils.hpp>
#include <Classes.hpp>
#include <Controls.hpp>
#include <Grids.hpp>
//---------------------------------------------------------
class PACKAGE TSG : public TStringGrid
{private:
        bool flag; // Оголошення змінної логічного типу
 protected:
 public: // Прототип конструктора нового компонента
        __fastcall TSG(TComponent* Owner);
         // Прототип події KeyDown
        DYNAMIC void __fastcall KeyDown(Word &Key,
                     Classes::TShiftState Shift);
         // Прототип події Click
        DYNAMIC void __fastcall Click(void);
 __published:
};
#endif
```

Файл SG.cpp:

```
#include <vcl.h>
#pragma hdrstop
#include "SG.h"
#pragma package(smart_init)
static inline void ValidCtrCheck(TSG *)
{  new TSG(NULL); }
//---------------------------------------------------------
__fastcall TSG::TSG(TComponent* Owner)
        : TStringGrid(Owner)
{ flag=true;            // Надання початкового значення змінній flag
}
// Новий код функції на подію Click
void __fastcall TSG::Click()
{ if(flag) Font->Size=Font->Size*2;    // Збільшення шрифту удвічі
  else Font->Size=Font->Size/2;        // Зменшення шрифту удвічі
  flag=!flag;             // Змінювання значення flag на протилежне
  // Подальше успадкування роботи методу Click() від класу TStringGrid
 TStringGrid::Click();
}
//---------------------------------------------------------
// Новий код функції на подію KeyDown
void __fastcall TSG::KeyDown(Word &Key,
```

```
                                Classes::TShiftState Shift)
{ if(Key==VK_ESCAPE)          // Перевірка натискання клавіші <Esc?
   for(int i=0; i<RowCount; i++)
   for(int j=0; j<ColCount; j++) Cells[j][i]="";
   if(Key==VK_F11)            // Перевірка натискання клавіші <F11>
    for(int i=0; i<RowCount;i++)          // Заповнення компонента
    for(int j=0; j<ColCount; j++)         // StringGrid
      Cells[j][i]=IntToStr(random(20)); // випадковими числами
  // Подальше  успадкування роботи методу KeyDown() від класу TStringGrid
  TStringGrid::KeyDown(Key, Shift);
}
//---------------------------------------------------------------
namespace Sg
{ void __fastcall PACKAGE Register()
   { TComponentClass classes[1] = {__classid(TSG)};
     RegisterComponents("Samples", classes, 0);
   }
}
```

Протестуємо модуль нового компонента. Варіант форми проекту для тестування:



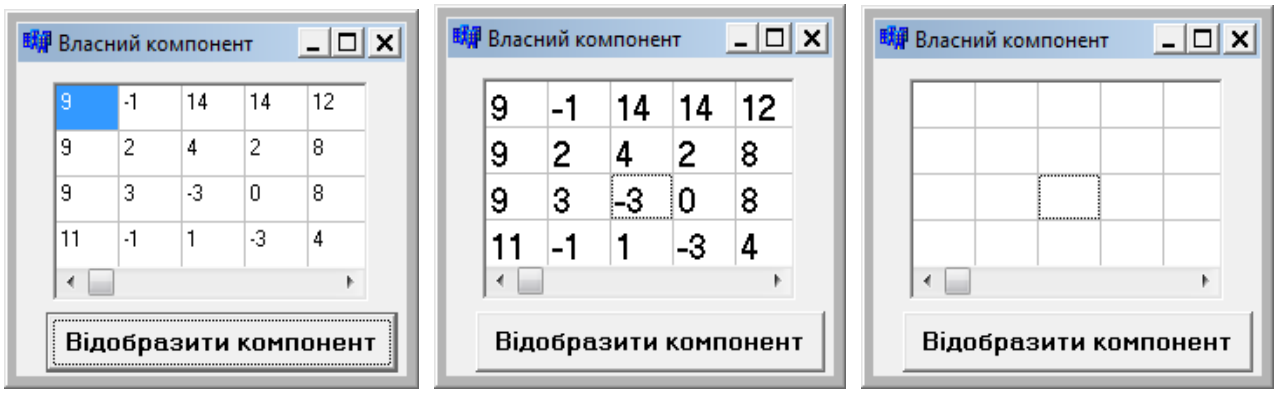
Створимо новий компонент як динамічну зміну `SGt`, задамо координати розташування та деякі властивості компонента.

Файл Unit1.cpp

```
#include <vcl.h>
#pragma hdrstop
#include "SG.cpp"  // Долучення модуля нового компонента
#include "Unit1.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
```

```
//---------------------------------------------------------
TSG *SGt;     // Оголошення динамічної змінної класу  нового компонента
//---- Відгук на кнопку "Відобразити компонент"-------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ SGt=new TSG(Form1);
  SGt->Parent=Form1;
  // Координати розміщення нового компонента на формі
  SGt->Left=20;    SGt->Top=10;
  // Значення властивостей нового компонента
  SGt->FixedRows=0;
  SGt->FixedCols=0;
  SGt->RowCount=4;
  SGt->ColCount=5;
  SGt->Options<<goEditing<<goTabs;
}
```

Власником об'єкта, що тестується, є вікно форми `Form1` (власник показано як параметр конструктора `TSG`). Властивість `Parent` створеного екземпляра класу також має значення `Form1`, а це означає, що новий компонент `SGt` буде розміщено у вікні `Form1`. Після запуску проекту та натискання на стандартну кнопку "Відобразити компонент" на формі з'явиться новий компонент. Коли курсор знаходитиметься у будь-якій комірці компонента `SGt` стануть доступними такі можливості: при натисканні клавіші <F11>, компонент заповнюватиметься випадковими числами (рис. 14.4, а), при клацанні лівою клавішею миші шрифт символів комірок збільшуватиметься удвічі (рис. 14.4, б), при натисканні клавіші <Esc> всі комірки автоматично очищуватимуться (рис. 14.4, в).



а **б** **в**

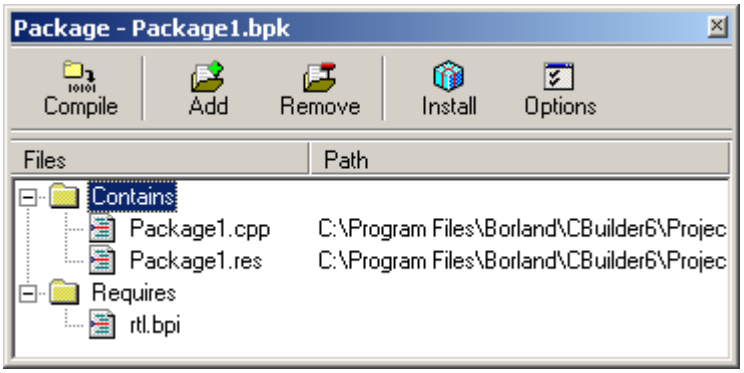
Рис. 14.4. Результати роботи з компонентом `SGt`:
а) при натисканні клавіші <F11> ;
б) при клацанні лівою клавішею;
в) натисканні клавіші <Esc>.

14.9.3 Інсталяція компонента

Компоненти у C++ Builder компілюються у пакети. Тому, для того, щоб ярлик створеного компонента відобразився на палітрі компонентів C++ Builder,

треба створити пакет і бажано розмістити його у власному каталозі. Для створення пакета слід виконати такі дії:

✓ виконати команду **File / New** і в діалоговому вікні New Items обрати піктограму Package. Після чого у вікні Package буде відображено склад пакета:

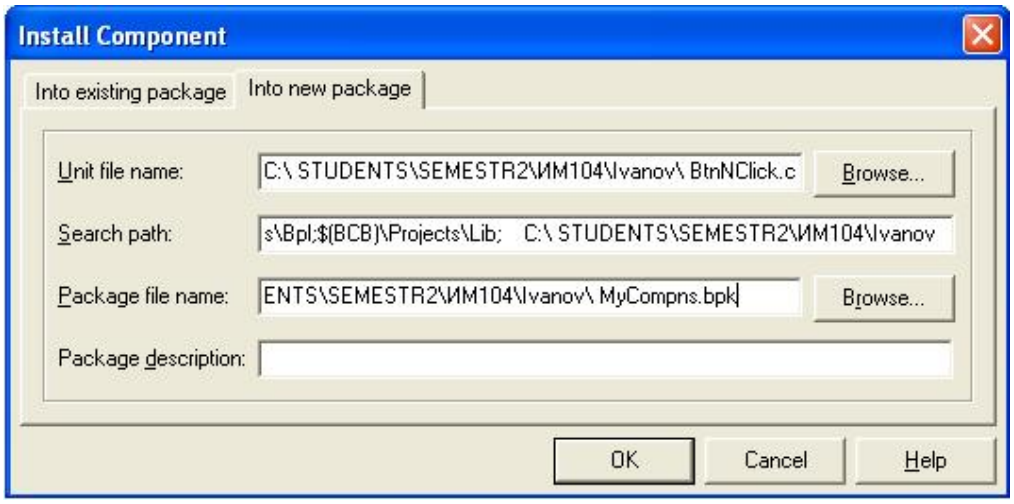


✓ виконати команду **File / Save As** для зберігання у власному каталозі. Ім'я `Package1` можна змінити, наприклад, на `MyCompn`.

Створений пакет складається з двох файлів: файл з розширенням `.bpk` та файл з розширенням `.cpp`, який містить інструкції щодо розміщення побудованих класів в особливій динамічній бібліотеці компонентів C++ Builder. Після компіляції цих файлів новим компонентом можна буде користуватися.

Як приклад розглянемо послідовність дій при компіляції компонента `BtnNClick`, розглянутого на початку п. 14.9.2. Для цього виконаємо дії у такій послідовності.

У головному меню оберемо опцію **Component / Install Component**, що спричинить появу вікна Install Component.



Перейдемо на сторінку Into new package цього вікна і задамо такі параметри нового пакета:

`Unit file name` – шлях (місцеперебування файла, яке записується у вигляді послідовності імен каталогів, розпочинаючи з кореневого) та ім'я бібліотеки з визначенням нового класу. Наприклад, при створенні нового компонента класу `TBtnNClick` (див. п. 14.9.2) визначимо шлях:

```
C:\STUDENTS\SEMESTR2\TC104\Ivanov\BtnNClick.cpp
```

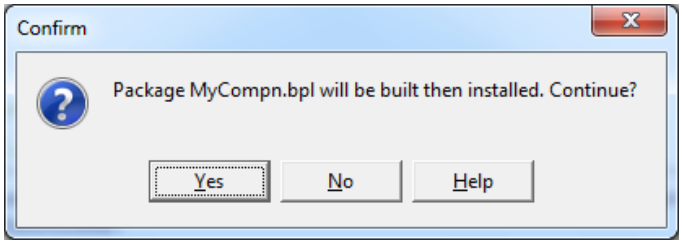
`Search path` – каталоги, в яких C++ Builder шукатиме `unit` нового класу. Якщо у попередньому пункті було зазначено повний шлях до власного каталогу, C++ Builder автоматично допише його до `Search path`; тому залишиться лише перевірити, чи правильно це було зроблено;

`Package file name` – шлях та ім'я файла пакета. Для компонента `TBtnNClick` можна задати:

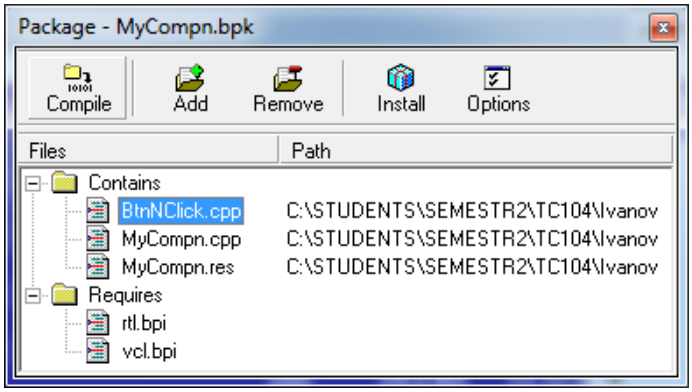
```
C:\STUDENTS\SEMESTR2\TC104\Ivanov\MyCompn.bpk
```

`Package description` – коментар до пакета. Цей параметр заповнювати необов'язково.

Після зазначення усіх параметрів і натискання кнопки "ОК" буде виконано спробу трансляції бібліотек, які входять до складу пакета, з виведенням відповідного вікна:



Незалежно від успішності виконання трансляції та реєстрації пакет буде створено і з'явиться відповідне діалогове вікно з переліком бібліотек:



Головне меню цього діалогового вікна має такі опції:

`Compile` – трансляція компонентів;

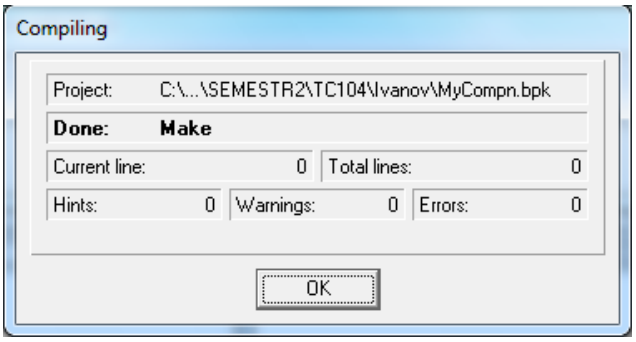
`Add` – долучення компонентів до пакета;

`Remove` – вилучення компонентів;

`Install` – реєстрація компонентів у палітрі;

`Options` – параметри пакета.

Якщо за трансляції чи реєстрації було припущено помилок, їх слід виправити й виконати команду `Compile` (натиснути відповідну кнопку). Після виправлення усіх помилок та успішної компіляції виведеться вікно з відповідним повідомленням.



При цьому новий компонент буде розміщено на сторінку `MyClasses` палітри компонентів, зазначену у функції `Register` (див. п. 14.9.2). Залишається позакривати усі діалогові вікна компіляції та перевірити правильність роботи новоствореного компонента.

Для розміщення нового компонента на формі слід перейти до вкладки `MyClasses` палітри компонентів, на якій розміщено піктограму цього компонента. Зовнішній вигляд цієї піктограми нічим не відрізняється від батьківської кнопки `Button`, але при наведенні на неї курсора можна побачити її ім'я `BtnNClick`. Ім'ям та надписом першого такого компонента, розташованого на формі, буде `BtnNClick1`. Після запуску програми при кожному натисканні цієї кнопки на ній буде дописуватись інформація про кількість натискань (див. форму на стор. 534).

Описана послідовність дій буде подібною при інсталяції будь-якого новоствореного компонента.

Питання та завдання для самоконтролю

1) Чим подібні і чим відрізняються у C++ поняття структури та класу?
2) Що таке інкапсуляція?
3) Що називається класом?
4) Чим відрізняється оголошення класу від визначення класу?
5) Чи можливо визначити метод класу поза класом?
6) Як співвідносяться поміж собою поняття об'єкта та класу?
7) Для чого потрібна секція `private` в оголошенні класу? Наведіть приклади полів чи методів, які доцільно вміщувати у цю секцію.
8) Для чого потрібна секція `public` оголошення класу?
9) Що таке конструктор та коли він викликається?
10) Чи може клас мати декілька конструкторів?
11) Що таке деструктор і коли він викликається?
12) Чи може в класі бути декілька деструкторів?
13) Що таке конструктор за замовчуванням та коли він викликається?
14) Що таке конструктор копіювання та коли він викликається?
15) Що таке успадкування? Наведіть приклади.
16) Що таке ієрархія класів?
17) Як оголошується похідний клас?

18) Що означає специфікатор `protected`?

19) Чи буде правильним твердження: створення похідного класу потребує докорінних змін у базовому класі?

20) Чи буде правильним твердження: якщо конструктор похідного класу не визначено, то об’єкти цього класу використовуватимуть конструктори базового класу?

21) Що таке поліморфізм? Наведіть приклади.

22) Що означає специфікатор `virtual`?

23) Яка функція називається суто віртуальною?

24) Що таке абстрактний клас?

25) Що означає специфікатор `DYNAMIC`?

26) Які елементі є основними у класі компонентів?

27) Який клас є базовим для усіх візуальних компонентів?

28) Який клас є базовим для усіх віконних компонентів?

29) Який клас є базовим для графічних компонентів?

30) Назвіть методи класу `TControl` та їхнє призначення.

31) Назвіть методи класу `TWinControl` та їхнє призначення.

32) Назвіть послідовність процесу створення нового компонента?

33) Для чого потрібна інсталяція компонента?

Розділ 15

Налагодження програм

Створювання нової програми на практиці доволі рідко відбувається без помилок. Розрізнюють помилки компіляції, які називають також помилками першого рівня (див. підрозд. 15.1), помилки, які виникають на етапі компонування (див. підрозд. 15.3), і помилки етапу виконування, які називають також помилками другого рівня (див. підрозд. 15.4). Окрім помилок, компілятор виводить зауваження і попередження щодо виявлення неточностей у програмі (див. підрозд. 15.2).

Виправлення помилок потребує від програміста глибоких знань мови та середовища програмування. Слід пам'ятати, що помилки можуть проявлятися не в тому місці програми, де їх було припущено. Складність усунення помилок полягає також у тому, що різні помилки можуть призводити до однакових проявів.

15.1 Помилки компіляції

Помилки компіляції (помилки першого рівня) пов'язані з помилковим записом команд, тобто це орфографічні й синтаксичні помилки. При виявленні такої помилки компілятор C++ Builder зупиняється на рядку з командою, в якій знайдено помилку. У нижній частині екрана з'являється текстове вікно, яке містить відомості про всі знайдені у проекті помилки (`Error`), зауваження (`Hints`) й попередження (`Warnings`). Кожен рядок цього вікна містить таку інформацію: ім'я файла, в якому знайдено помилку, номер рядка з помилкою і пояснювальний опис характеру помилки чи зауваження. Для швидкого переходу до конкретного рядка з помилкою слід двічі клацнути у рядку з її описом. Зауважимо, що одна помилка може спричинити за собою інші помилки, котрі зникнуть після її виправлення. Тому помилки слід виправляти послідовно, зверху донизу, і після виправлення кожної помилки зберігати і компілювати програму знов.

Без виправлення помилок (`Error`) запустити програму на виконання не вдасться, оскільки йдеться про синтаксичні помилки і компілятор не може перекласти текст програми на машинну мову. Зауваження (`Warning`) не перешкоджають запуску програми, але допомагають виправляти логічні помилки (помилки другого рівня), якщо програміст зверне на них увагу.

Розглянемо спочатку *синтаксичні помилки*. Повідомлення про такі помилки розпочинається текстом [`C++ Error`], після чого йде назва файла (`Unit1.cpp`) і у круглих дужках номер рядка, в якому знайдено помилку, тоді стоїть двокрапка і код помилки (наприклад `E2379`) з пояснювальним описом помилки. Деякі описи допомагають однозначно діагностувати помилку і виправити її. На жаль, більшість помилок насправді виникає не в тому рядку, на який вказує компілятор. Далі спочатку в табл. 15.1, а потім докладно з прикладами рисунків наведено найбільш поширені помилки компіляції, їхні причини й рекомендації щодо виправлення.

Таблиця 15.1

Помилки компіляції

№	Форма повідомлення про помилку	Характер помилки
1	`[C++ Error] Unit1.cpp(N): E2379 Statement missing;`	Пропущено крапку з комою в `N`-му чи у попередньому (`N-1`)-му рядку
2	`[C++ Error] Unit1.cpp(N): E2268 Call to undefined function '...'`	Неправильний запис імені функції чи не долучено заголовний файл, у якому оголошено цю функцію
3	`[C++ Error] Unit1.cpp(N): E2235 Member function must be called or its address taken`	Пропущено дужки у виклику функції, яка міститься в `N`-му рядку
4	`[C++ Error] Unit1.cpp(N): E2316 '_fastcall TForm::Button1Click (TObject*)'is not a member of 'TForm1`	Неправильно створено заголовок відгуку на подію `Button1Click`
5	`[C++ Error] Unit1.cpp(N): E2089 Identifier 'Button2Click' cannot have a type qualifier`	Неможливо створити відгук на подію `Button2Click`, оскільки не всі фігурні дужки закрито у попередній функції
6	`[C++ Error] Unit1.cpp(N): E2451 Undefined symbol 'x'`	Змінну `x` не було оголошено до використання чи то ім'я змінної написано з помилкою
7	`[C++ Error] Unit1.cpp(N): E2238 Multiple declaration for 'x'`	Багаторазове оголошення змінної `x` в одній області видимості
8	`[C++ Error] Unit1.cpp(N): E2134 Compound statement missing }`	Кількість фігурних дужок, що відкриваються й закриваються, не збігається, а саме не вистачає закритої фігурної дужки у тексті програми, який розміщено вище
9	`[C++ Error] Unit1.cpp(N): E2121 Function call missing )`	Пропущено (чи помилково поставлено) круглу дужку, яка закриває список параметрів при викликанні функції
10	`[C++ Error] Unit1.cpp(N): E2227 Extra parameter in call to pow(double, double)`	Кількість параметрів при викликанні функції перевищує їхню кількість за синтаксисом
11	`[C++ Error] Unit1.cpp(N): E2193 Too few parameters in call to 'pow(double, double)'`	Кількість параметрів є менша, аніж цього вимагає синтаксис функції
12	`[C++ Error] Unit1.cpp(N): E2030 Misplaced break`	Присутність `break` у цьому місці програми не має сенсу
13	`[C++ Error] Unit1.cpp(N): E2054 Misplaced else`	Неможливо встановити до якого `if` належить `else`
14	`[C++ Error] Unit1.cpp(N): E2034 Cannot convert 'int' to 'char'`	Суперечка типів: не можна присвоїти змінній типу `char` значення типу `int`
15	`[C++ Error] Unit2.cpp(N): E2356 Type mismatch in redeclaration of 'sum(int, int)'`	Невідповідність типу прототипу функції та її визначення (реалізації)

Закінчення табл. 15.1

№	Форма повідомлення про помилку	Характер помилки
16	`[C++ Error] Unit1.cpp(N): E2040 Declaration terminated incorrectly`	Оголошення припинено некоректно. Помилка внаслідок неправильного розташування операторних дужок чи то дійсно некоректного оголошення
17	`[C++ Error] Unit2.cpp(N): E2308 do statement must have while`	Помилка виникає внаслідок неправильного розташування фігурних дужок в операторі `do-while`
18	`[C++ Error] Unit2.cpp(N) E2308 Unterminated string or character constant`	Рядок не завершено, тобто в кінці рядка пропущено лапки (")
19	`[C++ Error] Unit1.cpp(N): E2376 If statement missing (`	Пропущено одну чи обидві дужки в умові оператора `if` чи умову написано з синтаксичними помилками
20	`[C++ Error] Unit1.cpp(N): E2378 For statement missing ;`	Відсутня крапка з комою як розділювач складових частин оператора `for`
21	`[C++ Error] Unit1.cpp(N): E2060 Illegal use of floating point`	Неприпустима операція над дійсним числом

Розглянемо більш докладно причини, які можуть призвести до наведених в таблиці помилок.

1) **`[C++ Error] Unit1.cpp(N): E2379 Statement missing;`**

Таке повідомлення вказує на те, що пропущено крапку з комою в **N**-му чи то у попередньому (**N-1**)-му рядку (рис. 15.1). Для усунення помилки в кінці попереднього рядка слід поставити крапку з комою (якщо вона є відсутня).

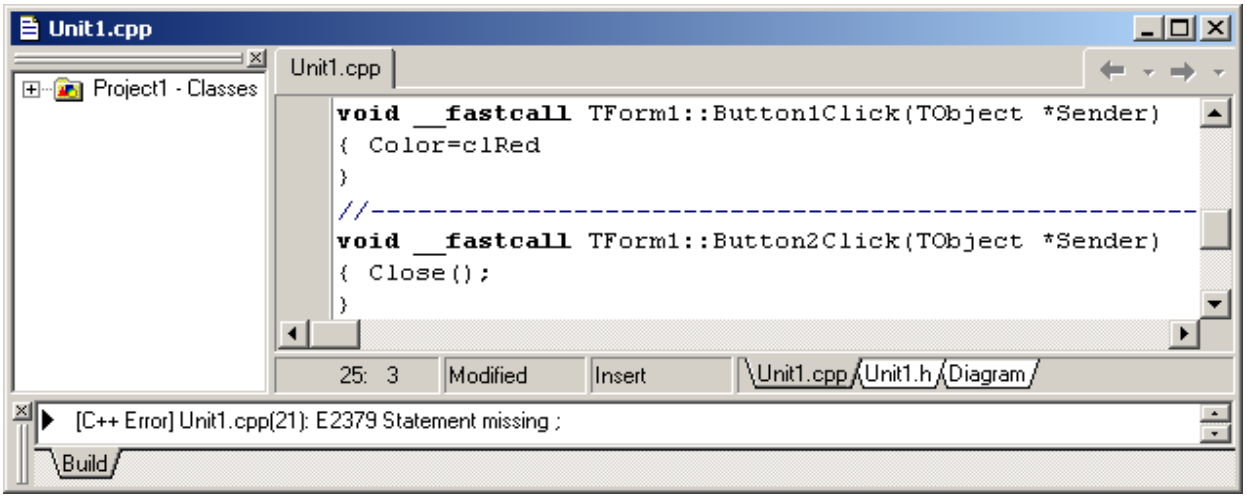


Рис. 15.1. Помилка E2379: пропущено крапку з комою

Окрім того, ця помилка виникає, коли компілятор сподівається на крапку з комою, а зустрічає інший символ, наприклад, якщо у математичному виразі стоїть зайва дужка (рис. 15.2). Для усунення помилки слід перевірити правиль-

ність формули, кількість та відповідність дужок, що відкриваються й закриваються.

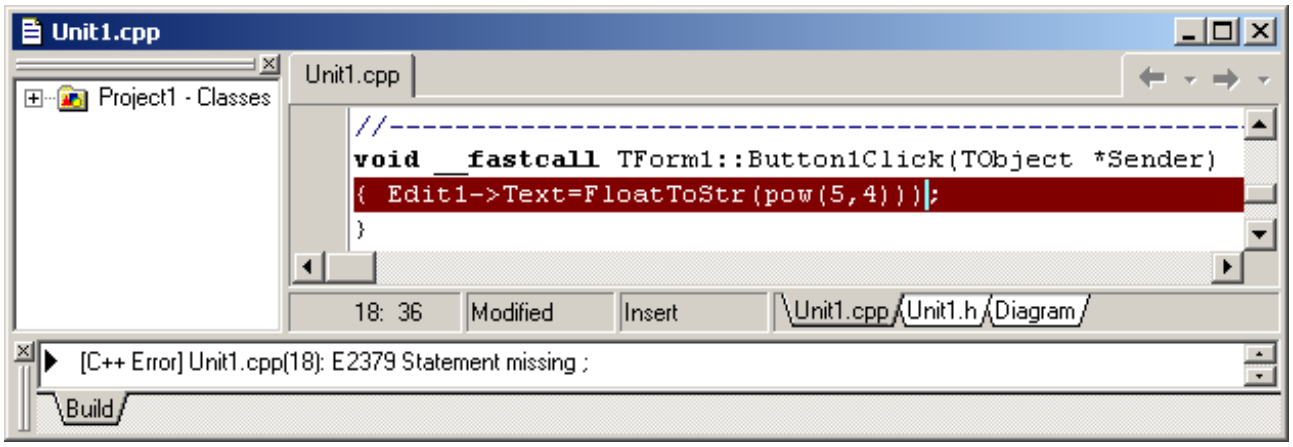


Рис. 15.2. Помилка E2379: зайва дужка у математичному виразі

2) **[C++ Error] Unit1.cpp(N): E2268 Call to undefined function 'close'**

Причиною такої помилки може бути:

✓ неправильний запис імені функції. Так, у прикладі на рис. 15.3 функцію `Close()` написано з маленької літери;

✓ ім'я функції написано правильно, але не долучено заголовний файл, у якому оголошено цю функцію (рис. 15.7). Наприклад, така помилка виникне, якщо використовувати у програмі математичні функції і не написати на початку програми:

```
#include <math.h>
```

В такому разі для усунення помилки слід долучити необхідний заголовний файл.

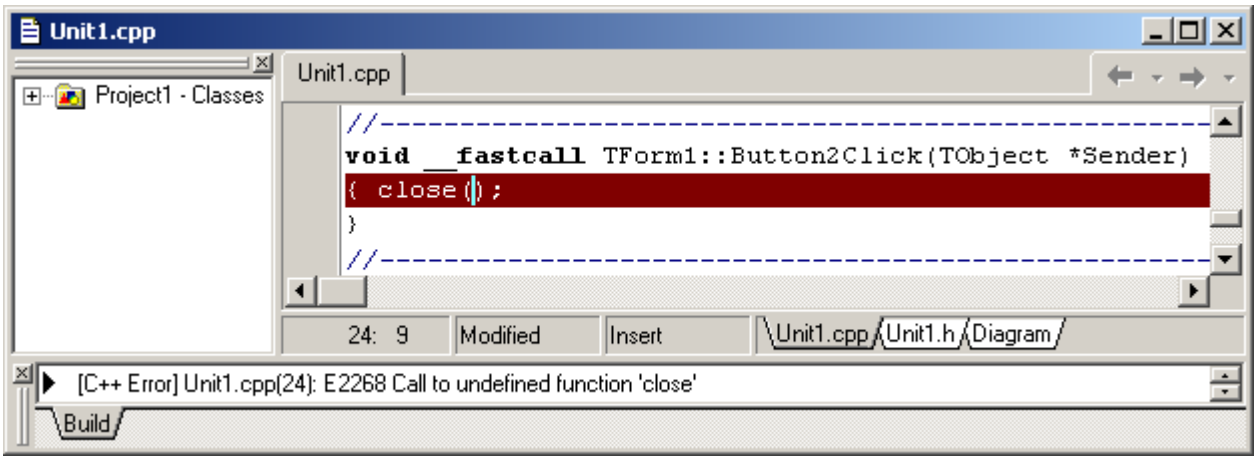


Рис. 15.3. Помилка E2268: ім'я функції написано неправильно

3) **[C++ Error] Unit1.cpp(N): E2235 Member function must be called or its address taken**

Пропущено порожні дужки у виклику функції `Close()` (рис. 15.4). Для усунення помилки слід поставити дужки.

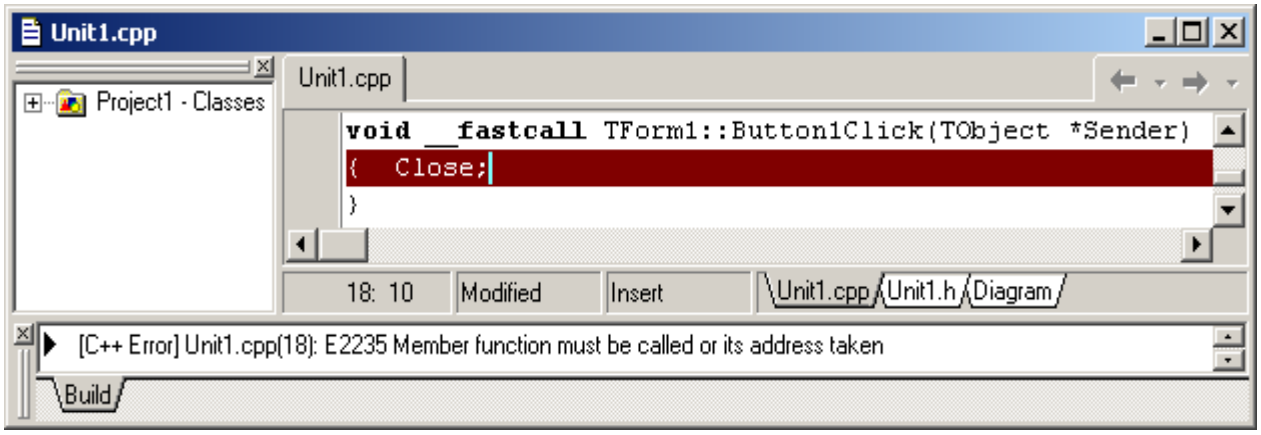


Рис. 15.4. Помилка E2235: пропущено порожні дужки у виклику функції

4) **[C++ Error] Unit1.cpp(N): E2316 '_fastcall TForm::Button2Click(TObject*)' is not a member of 'TForm1'**

Така помилка (рис. 15.5) виникає, якщо заголовок відгуку на подію `Button2Click` написати власноруч (як відомо, система сама має створити цей заголовок). Слід вилучити написаний заголовок, повернутися на форму і створити відгук на подію коректно.

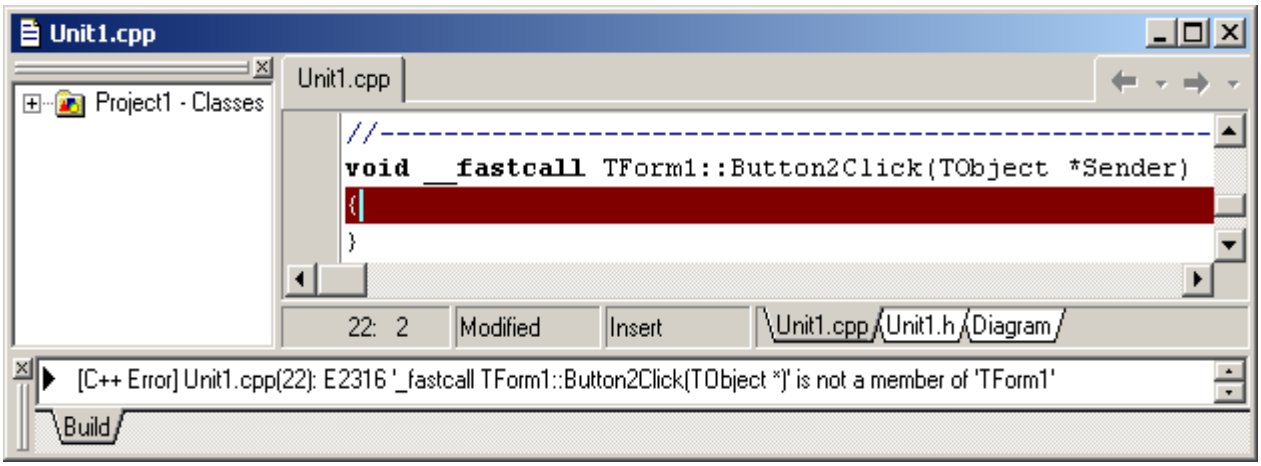


Рис. 15.5. Помилка E2316: заголовок відгуку на подію написано власноруч

5) **[C++ Error] Unit1.cpp(N): E2089 Identifier 'Button2Click' cannot have a type qualifier**

Така помилка належить до прихованих помилок. Складається враження, що все написано правильно (рис. 15.6). Але за уважного розглядання виявляється, що за кілька рядків до помилки відсутня закрита фігурна дужка і компіляторові "здається", що функція відгуку на подію `Button2Click` міститься всередині функції відгуку події `Button1Click`, а вкладення функцій одна в одну в C++ не дозволяється. Слід поставити пропущену дужку – і помилка зникне.

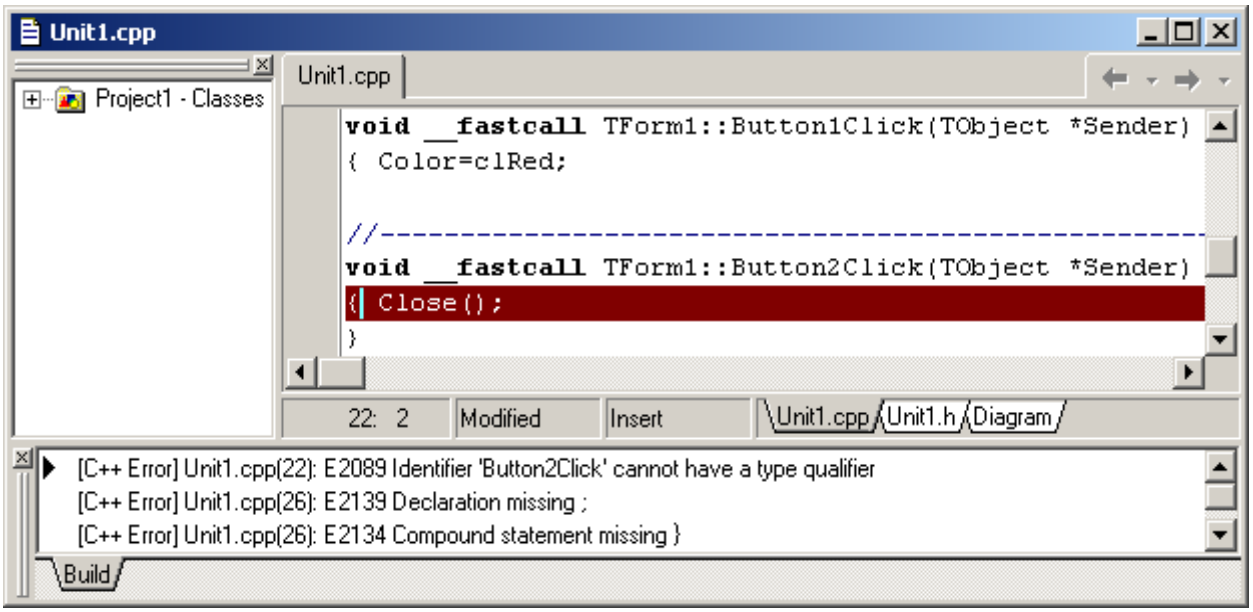


Рис. 15.6. Помилка E2089: пропущено закриту дужку у попередній функції

6) **`[C++ Error] Unit1.cpp(N): E2451 Undefined symbol 'x'`**

Така помилка виникає з двох причин: змінну не було оголошено перед використанням (рис. 15.7) або ім'я змінної написано з помилкою (наприклад, не збігається регістр літер). Для усунення помилки слід оголосити змінну або, якщо її вже було оголошено, перевірити правильність імені.

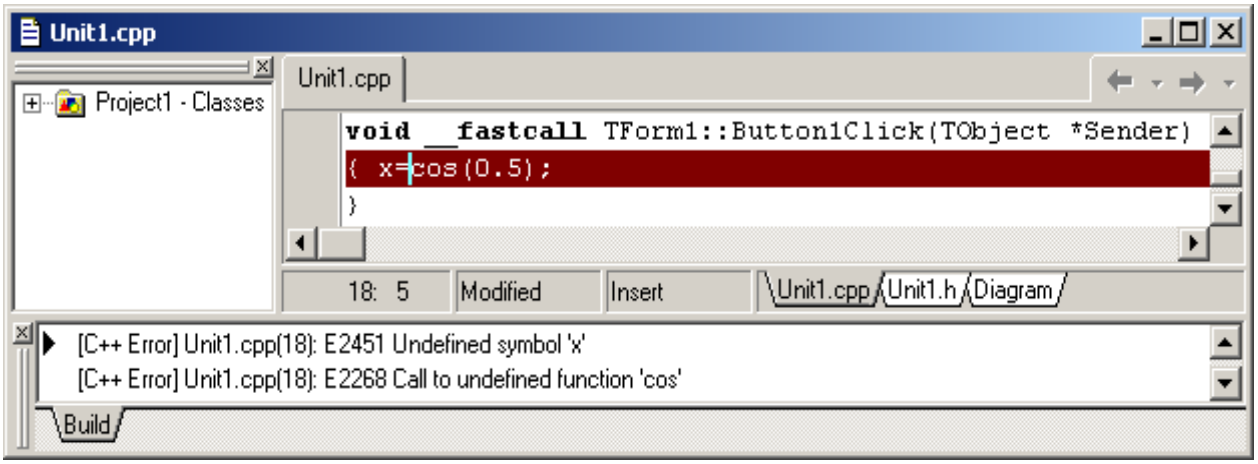


Рис. 15.7. Помилка E2451: не оголошено змінну `x`

Зауважимо, що змінну може бути оголошено в іншій області видимості. У прикладі на рис. 15.8 змінну `i` оголошено усередині циклу та вважається за невідому поза циклом. У такому разі слід оголосити змінну разом зі змінною `s` поза циклом або використовувати іншу змінну.

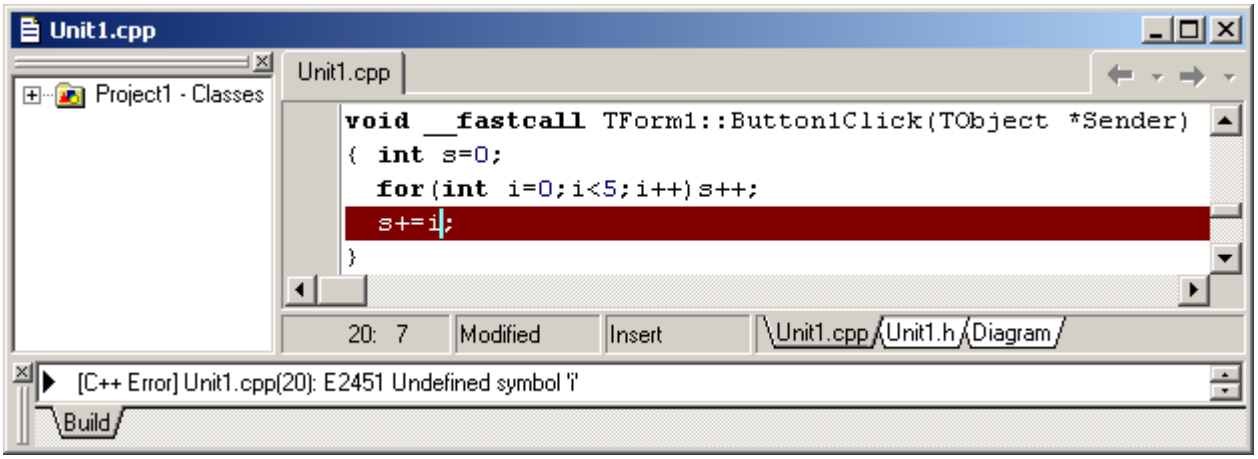


Рис. 15.8. Помилка E2451: змінну `i` оголошено в іншій області видимості

7) **`[C++ Error] Unit1.cpp(N): E2238 Multiple declaration for 'x'`**

Така помилка виникає, коли змінна оголошена двічі в одній області видимості (рис. 15.9). Слід вилучити повторне оголошення (у рядку 19 вилучити `double` і залишити `x=-2.5;`)

Разом з цією помилкою йде помилка

`[C++ Error] Unit1.cpp(N): E2344 Earlier declaration of 'x'`

Вона вказує на рядок, у якому відбулося попереднє оголошення змінної `x`.

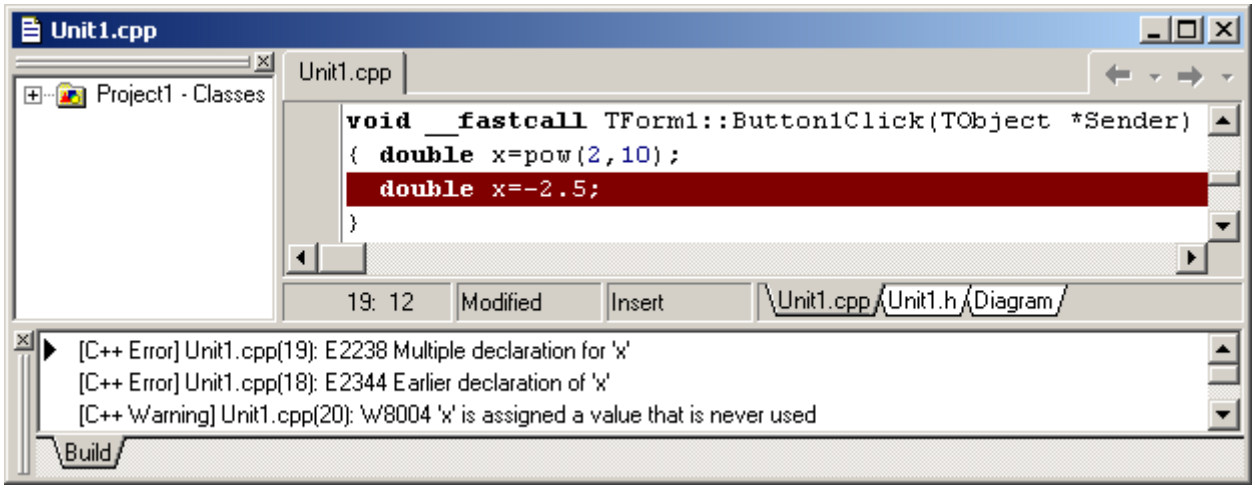


Рис. 15.9. Помилка E2238: змінну `x` оголошено двічі

Окрім помилок, на рис. 15.9 можна бачити попередження:

`[C++ Warning] Unit1.cpp W8004 'x' is assigned a value that is never used`

Детальніше про попередження див. далі підрозд. 15.2.

8) **`[C++ Error] Unit1.cpp(N): E2134 Compound statement missing }`**

Таке повідомлення видається, якщо кількість фігурних дужок, що відкриваються й закриваються, не збігається (рис. 15.10). Воно не означає, що фігурну дужку слід поставити саме в цьому рядку. Слід уважно передивитись попередній текст програми і визначити рядок, у якому було пропущено цю дужку.

Таке ж саме повідомлення видається, коли фігурну дужку закрито несвоєчасно (див. рис. 15.6).

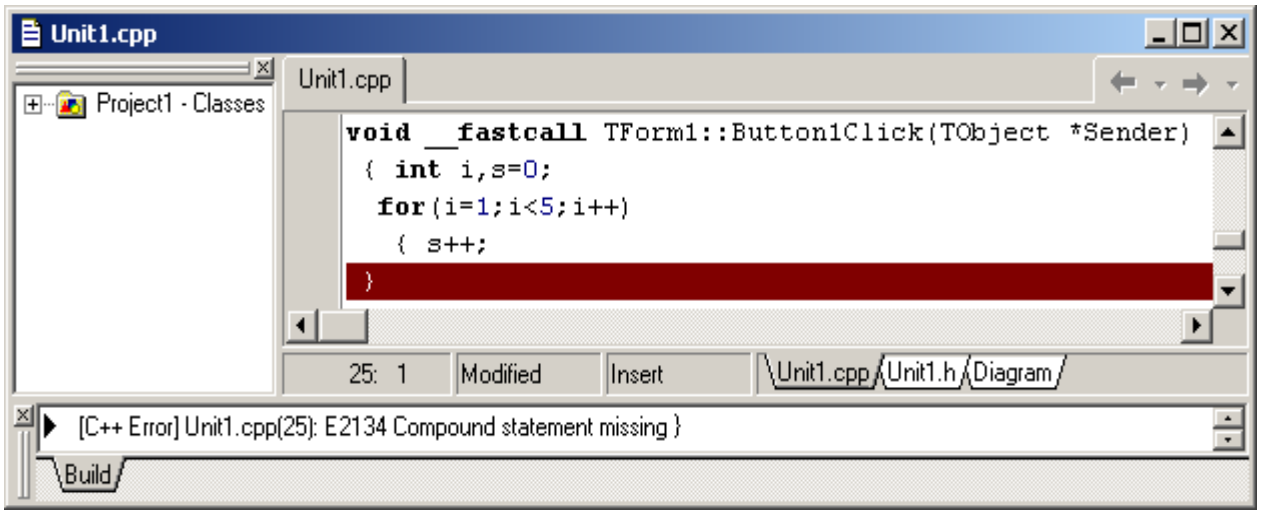


Рис. 15.10. Помилка E2134: пропущено закриту фігурну дужку

9) **[C++ Error] Unit1.cpp(N): E2121 Function call missing)**

Це повідомлення свідчить про те, що було пропущено (або поставлено у невідповідному місці) круглу дужку, що закриває список параметрів при викликанні функції. Найчастіш це буває, коли поряд йде кілька дужок (рис. 15.11). Слід уважно перевірити кількість круглих дужок у рядку і правильність їхнього розташування.

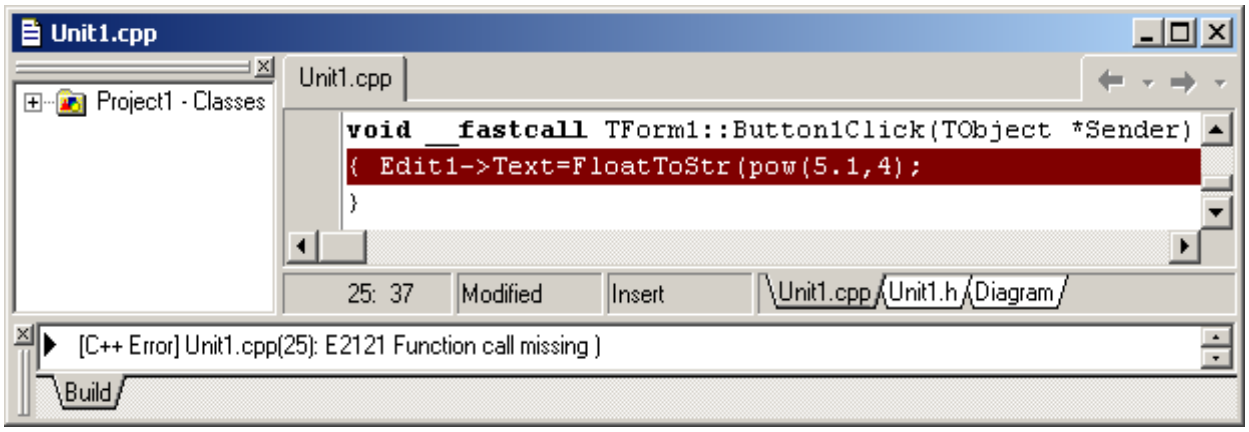


Рис. 15.11. Помилка E2121: пропущено круглу дужку у виклику функції

10) **[C++ Error] Unit1.cpp(N): E2227 Extra parameter in call to pow(double, double)**

Така помилка виникає, коли кількість параметрів при викликанні функції перевищує їхню кількість за синтаксисом. У прикладі на рис. 15.12 замість десяткової крапки у числі 5.1 (або 1.4) було поставлено кому, і компіляторові "здалося", що викликається функція `pow()` з трьома параметрами замість двох. Слід перевірити правильність написання чисел, відповідність параметрів і правильність розташування круглої дужки, яка закриває виклик функції (можливо, її було пропущено чи помилково поставлено в інше місце).

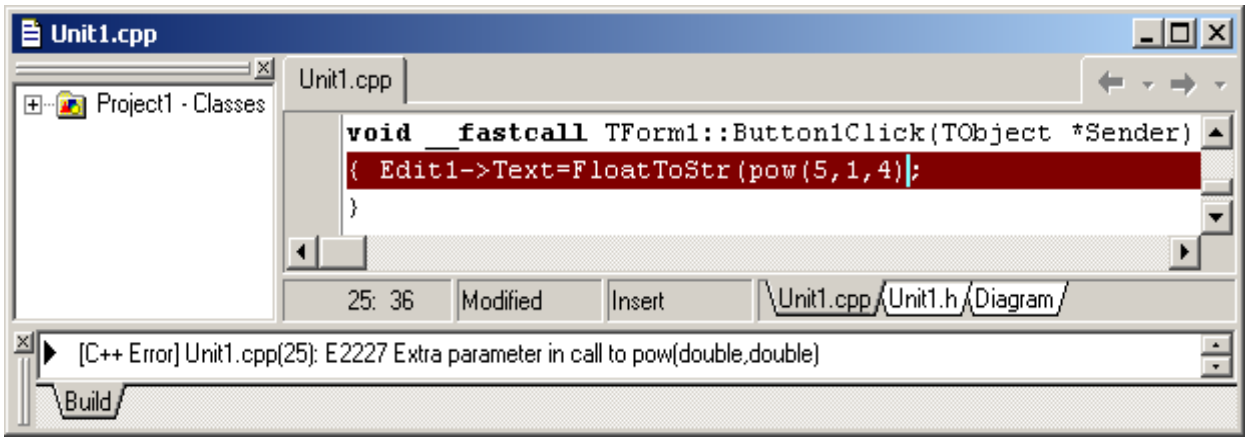


Рис. 15.12. Помилка E2227: виклик функції `pow()` з трьома параметрами замість двох

11) **`[C++ Error] Unit1.cpp(N): E2193 Too few parameters in call to 'pow(double, double)'`**

Така помилка виникає, коли кількість параметрів є менша, аніж вимагає синтаксис. У прикладі на рис. 15.13, можливо, десяткова крапка поставлена випадково замість коми, яка розділяє параметри, або пропущено один з параметрів. Слід уважно перевірити список фактичних параметрів при викликанні функції.

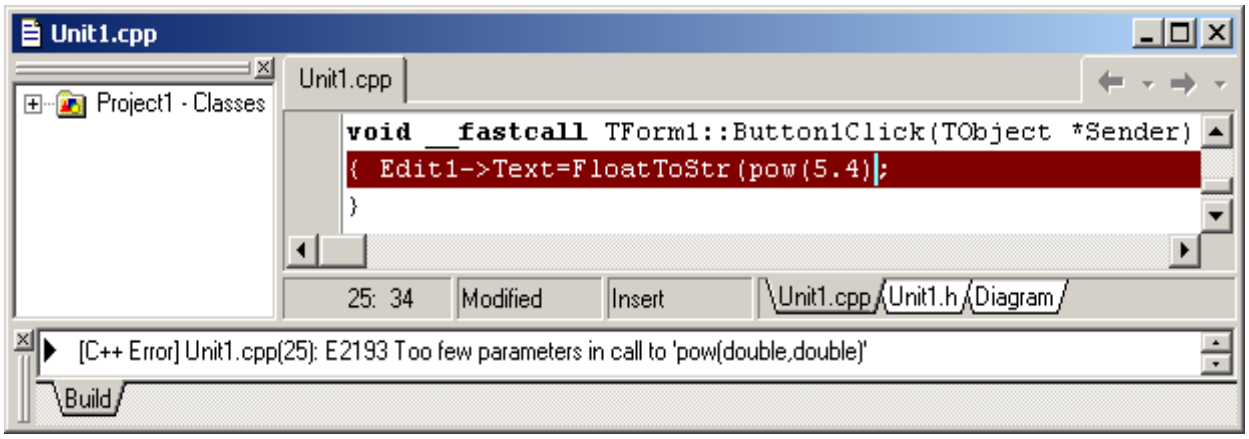


Рис. 15.13. Помилка E2193: виклик функції `pow()` з одним параметром замість двох

12) **`[C++ Error] Unit1.cpp(N): E2030 Misplaced break`**

Оскільки, зазвичай, `break` припиняє виконування циклу, то якщо його розташовано поза циклом, видається така помилка. Вона свідчить про те, що присутність `break` в цьому місці програми не має сенсу. Слід уважно перевірити, чому так сталося, що `break` опинився поза циклом. Можливо, пропущено фігурні дужки (рис. 15.14)? Чи ці дужки закрито завчасно? Чи дійсно вживати `break` не було сенсу?

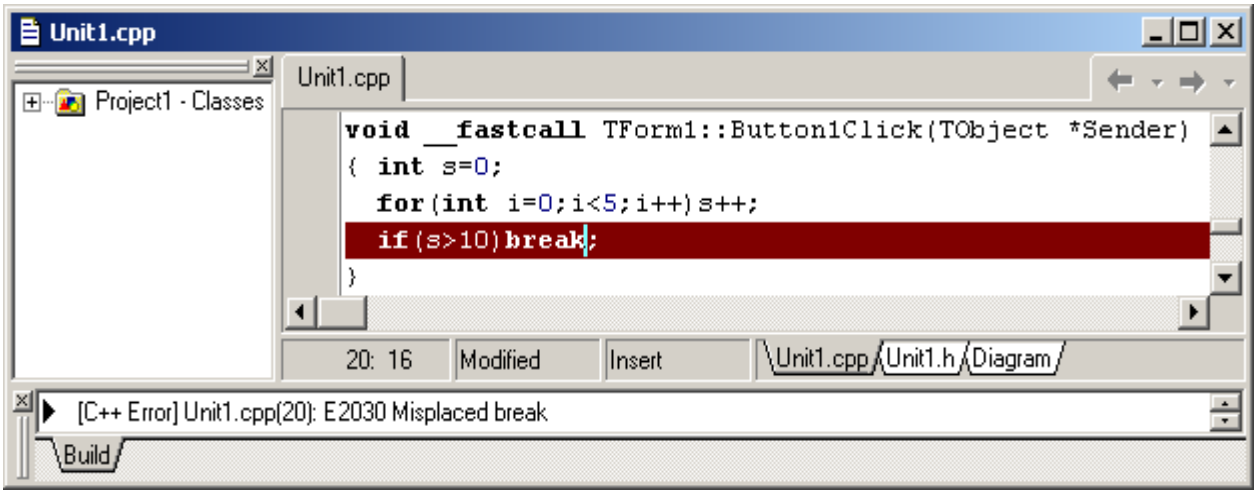


Рис. 15.14. Помилка E2030: пропущено фігурні дужки і вживання `break` немає сенсу

13) **`[C++ Error] Unit1.cpp(N): E2054 Misplaced else`**

Інструкція `else` міститься не на власному місці. Така помилка виникає, коли неможливо встановити, до якого `if` належить `else`. У прикладі на рис. 15.15 ця помилка виникла внаслідок того, що було помилково поставлено крапку з комою одразу після умови і вважається, що умовний оператор вже завершився і `else` до нього не належить. У прикладі на рис. 15.16 ця помилка виникла внаслідок того, що пропущено операторні (фігурні) дужки після умови і перед `else`.

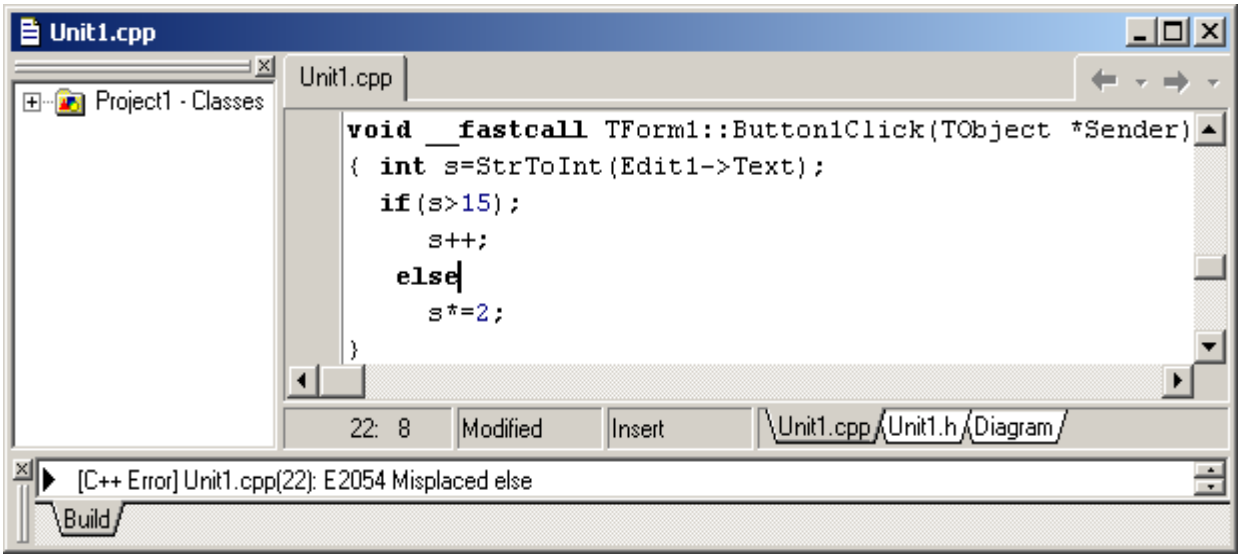


Рис. 15.15. Помилка E2054: неможливо встановити, до якого `if` належить `else`, через зайву крапку з комою після умови

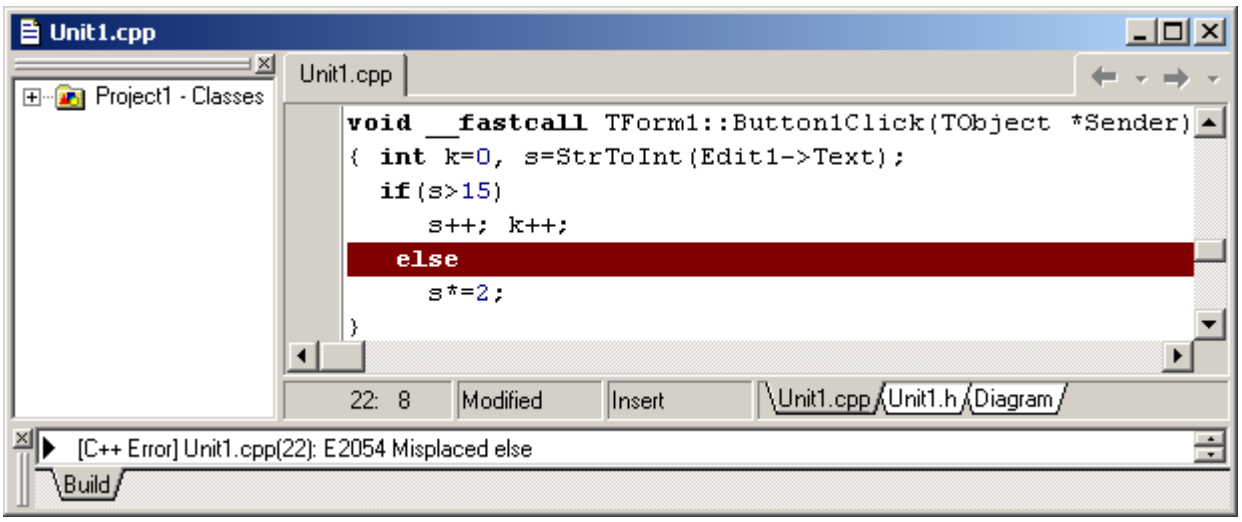


Рис. 15.16. Помилка E2054: неможливо встановити, до якого `if` належить `else`, через те, що пропущено фігурні дужки

14) **`[C++ Error] Unit1.cpp(N): E2034 Cannot convert 'int' to 'char'`**

Така помилка виникає за спроби присвоїти змінній значення, яке не може бути неявно перетвореним на тип цієї змінної, тобто виникає суперечка типів. На рис. 15.17 показано дві такі помилки: в рядку 23 число 15 не може бути неявно перетвореним у тип `char*`, а в наступному – 24-му рядку послідовність символів не можна записати в один символ. Така сама помилка виникає за спроби задати фактичні параметри функції зі значенням, яке не може бути неявно перетвореним на тип, зазначений при оголошенні функції.

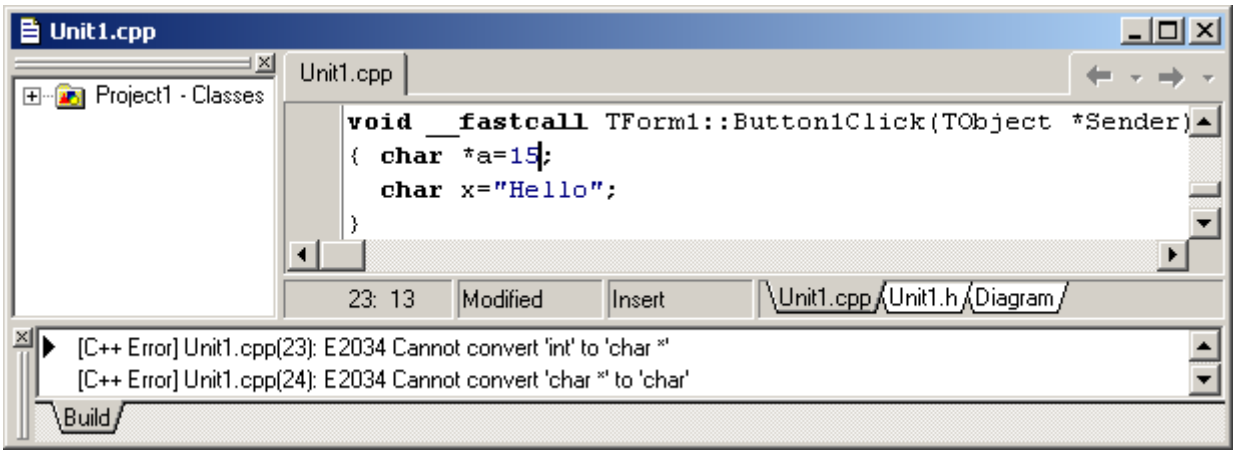


Рис. 15.17. Помилка E2034: суперечка типів

15) **`[C++ Error] Unit1.cpp(N): E2040 Declaration terminated incorrectly`**

Помилка виникає внаслідок неправильного розташування операторних дужок. У прикладі на рис. 15.18 дужка після `s++;` закрита завчасно, компілятор інтерпретує її як завершення тексту функції `Button1Click`, тому вважається, що наступні рядки є некоректні (можливо, зайві). Слід уважно перевірити відповідність і правильність розташування усіх фігурних дужок.

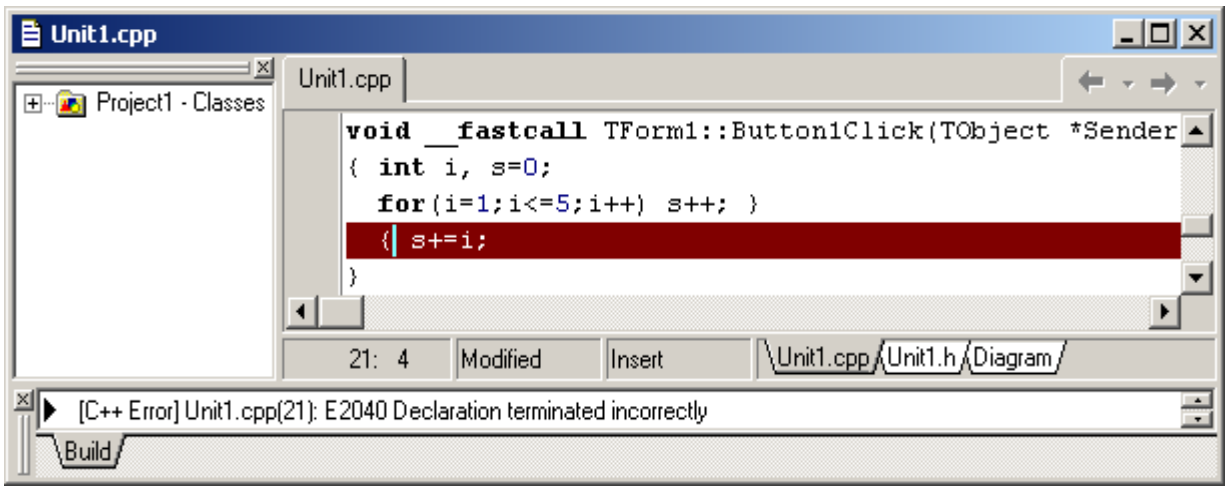


Рис. 15.18. Помилка E2040: фігурна дужка закрита завчасно

16) **[C++ Error] Unit2.cpp(N): E2356 Type mismatch in redeclaration of 'sum(int, int)'**

Найчастіше така помилка виникає у визначенні функції, коли у заголовку функції було зазначено інший тип результату (рис. 15.19). Слід уважно простежити, щоб прототип функції та її визначення (реалізація) мали однаковий тип результату.

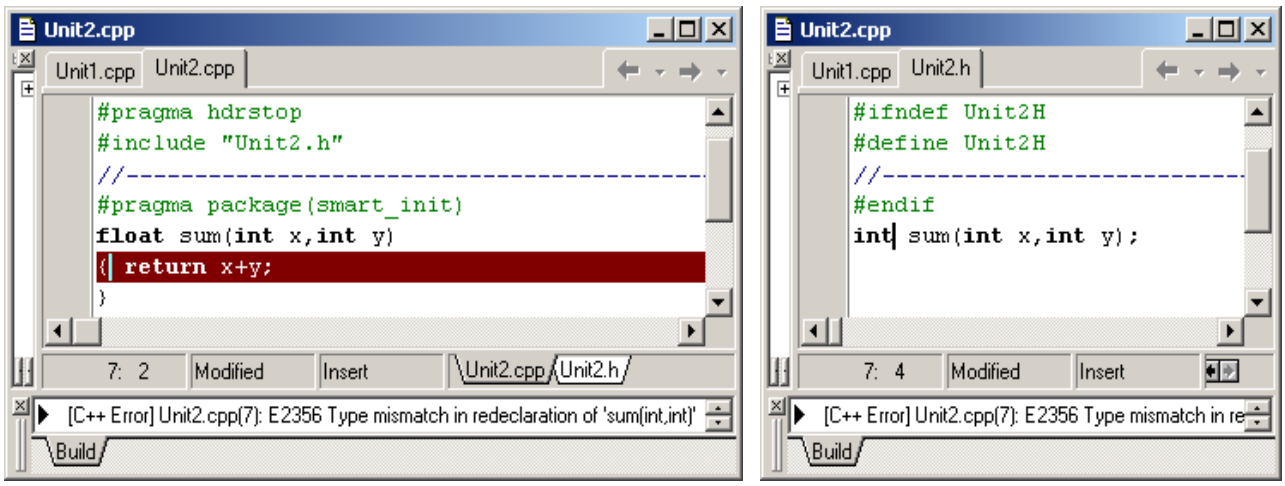


Рис. 15.19. Помилка E2356: типи функції в оголошенні та визначенні не збігаються

17) **[C++ Error] Unit2.cpp(N): E2308 do statement must have while**

Помилка виникає внаслідок неправильного розташування операторних дужок (компілятор чекає на `while` після `s++;}`, але не знаходить цієї інструкції (рис. 15.20)).

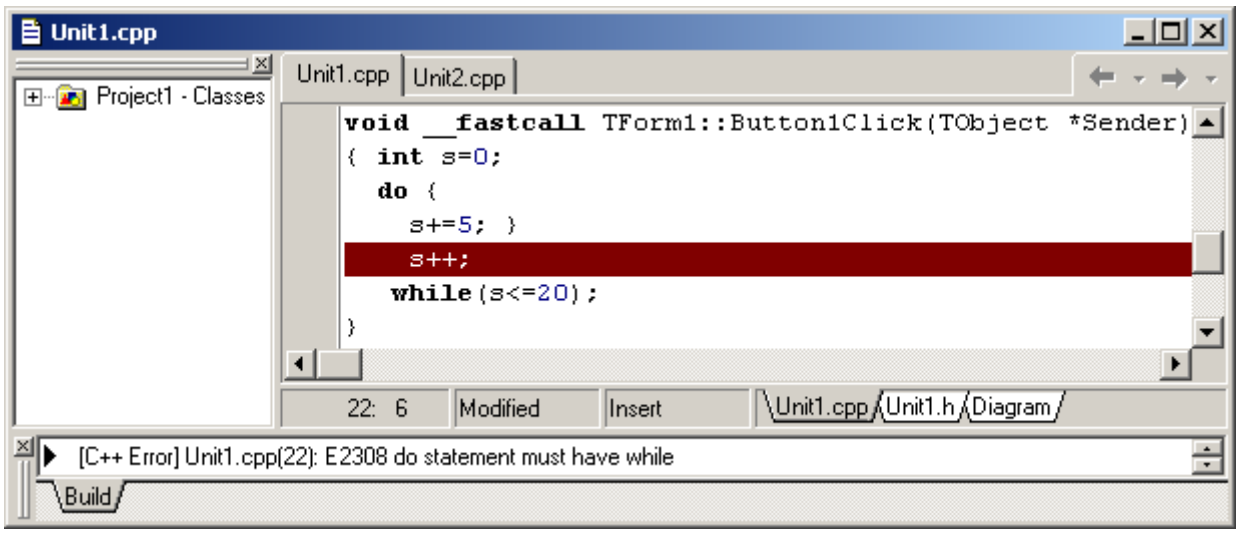


Рис. 15.20. Помилка E2308: завчасно закрито фігурну дужку

18) **[C++ Error] Unit2.cpp(N) E2308 Unterminated string or character constant**

Помилка виникає, коли в кінці рядка чи після символу пропущено лапки, тобто рядок не завершено (рис. 15.21).

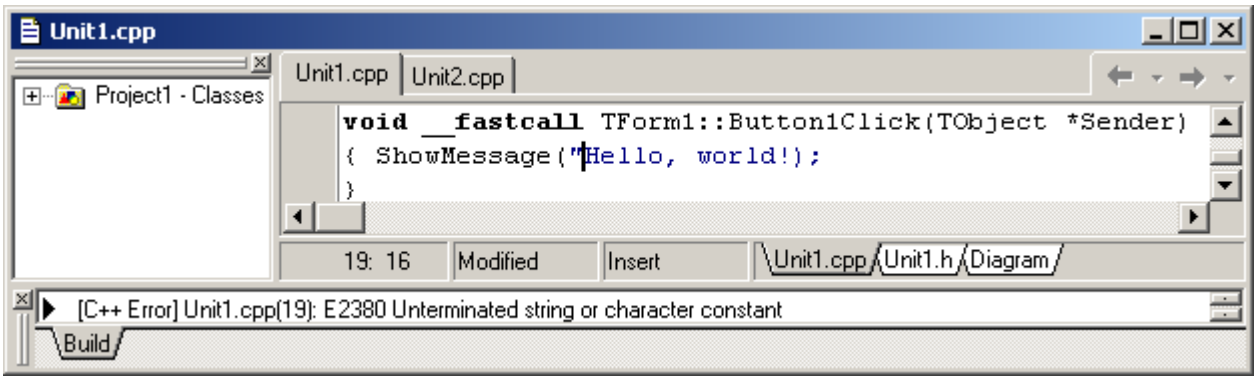


Рис. 15.21. Помилка E2380: пропущено лапки

19) **[C++ Error] Unit1.cpp(N): E2376 If statement missing (**

Помилка виникає, коли пропущено одну чи обидві дужки в умові оператора `if` (рис. 15.22).

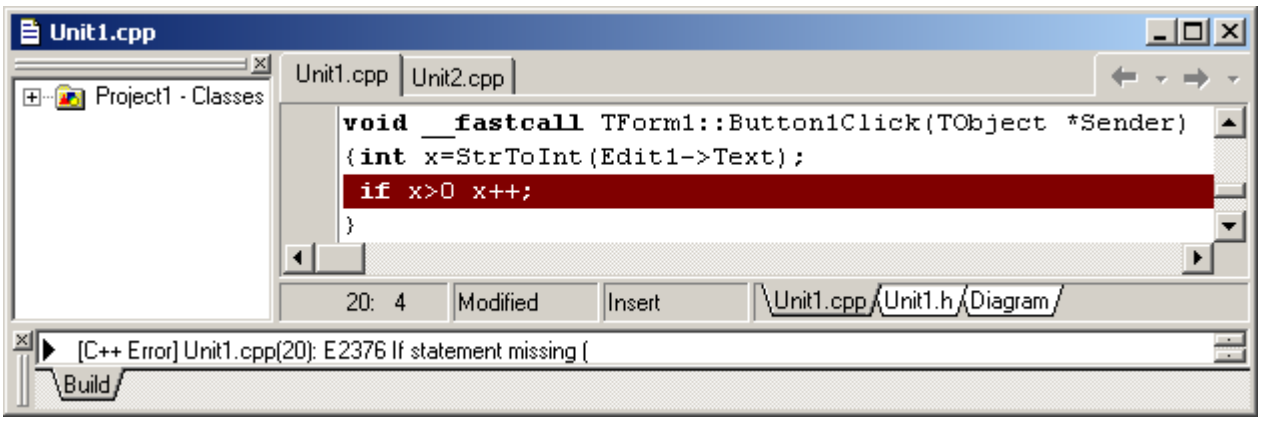


Рис. 15.22. Помилка E2376: пропущено дужки в умові оператора `if`

20) **[C++ Error] Unit1.cpp(N): E2378 For statement missing ;**

Помилка виникає, коли замість крапки з комою стоїть кома (рис. 15.23) чи інший символ або крапка з комою взагалі відсутня (рис. 15.24). Перевірте наявність складових частин оператора `for` та їх розділювачів – крапок з комами.

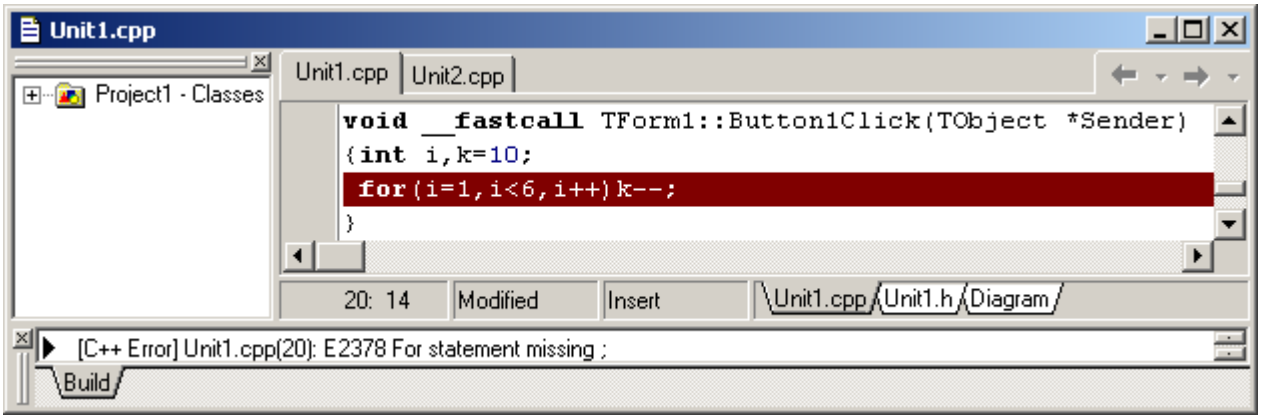


Рис. 15.23. Помилка E2378: розділювач у операторі `for` – кома замість крапки з комою

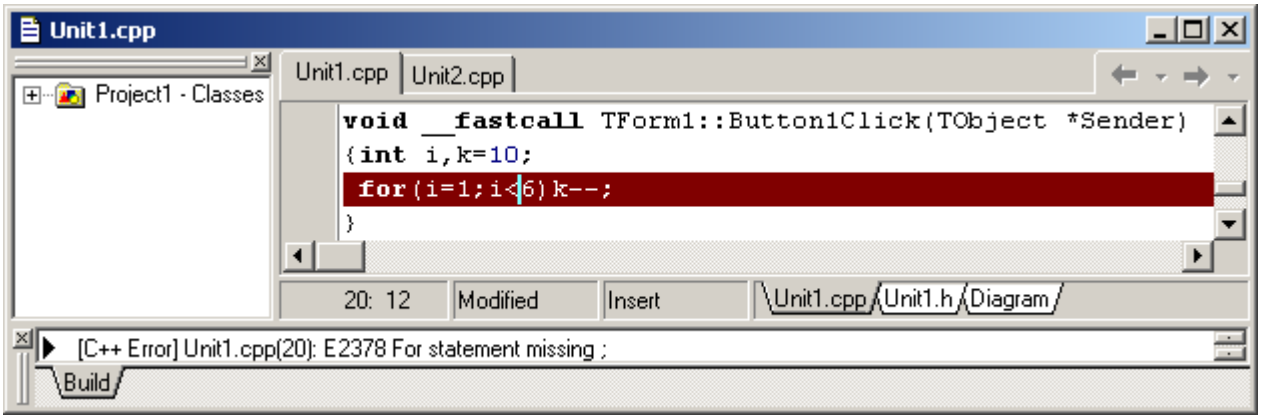


Рис. 15.24. Помилка E2378: друга крапка з комою у `for` є відсутня

21) **[C++ Error] Unit1.cpp(N): E2060 Illegal use of floating point**

Помилка виникає за спроби виконати операцію, для дійсних чисел не визначену, наприклад обчислення остачі від ділення (рис. 15.25). Перевірте правильність виразу. Можливо, слід змінити тип операнда (з `float` на `int`) чи операцію.

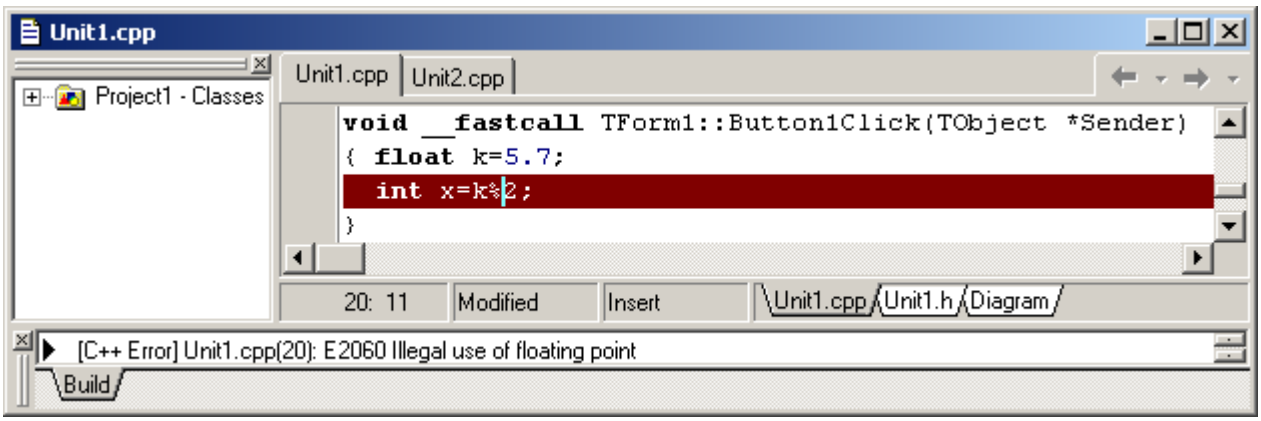


Рис. 15.25. Помилка E2378: операцію обчислення остачі від ділення дійсних чисел не визначено

22) **[C++ Error] Unit1.cpp(N): E2015 Ambiguity between '_fastcall Sysutils::IntToStr(__int64)' and '_fastcall Sysutils::IntToStr(int)'**

Помилка виникає за спроби задати фактичний параметр іншого типу, аніж формальний (на рис. 15.26 функція `IntToStr()` чекає на ціле число, а зустрічає дійсне).

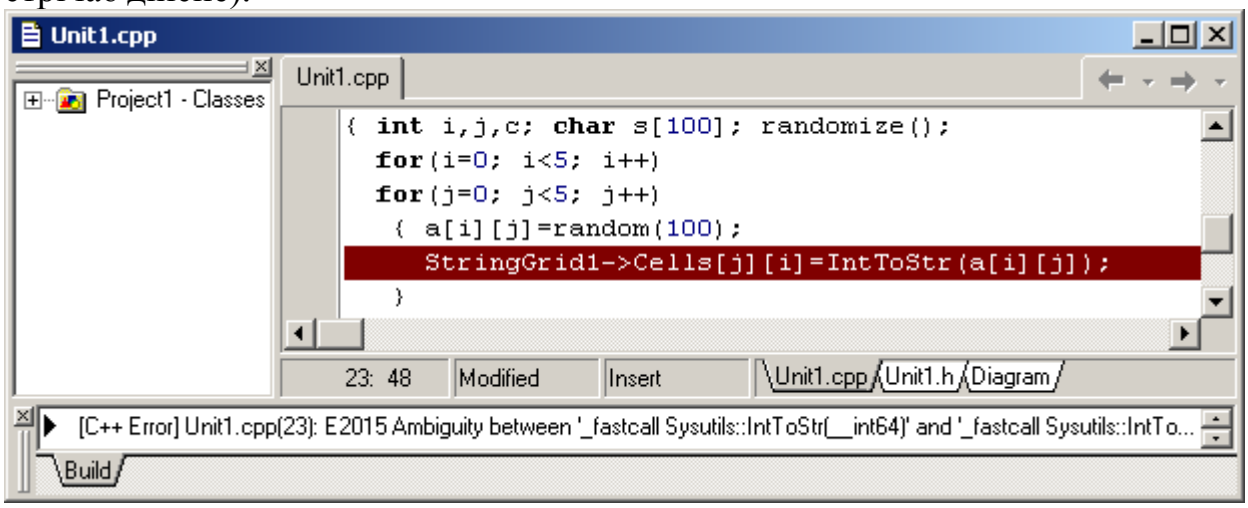


Рис. 15.26. Помилка E2378: фактичний параметр у функції `IntToStr()` дійсний замість цілого

15.2 Попередження і підказки

За виявляння у програмі неточностей, які не є помилками, компілятор виводить підказки (Hints) і попередження (Warnings). Доволі часто вони допомагають програмістові уникнути логічних помилок у програмі.

Розглянемо деякі з найбільш поширених попереджень і підказок.

1) **[C++ Warning] Unit1.cpp(N): W8060 Possibly incorrect assignment**

Попередження виводиться, коли в умові замість операції порівняння "==" використовується оператор присвоювання "=" (рис. 15.27).

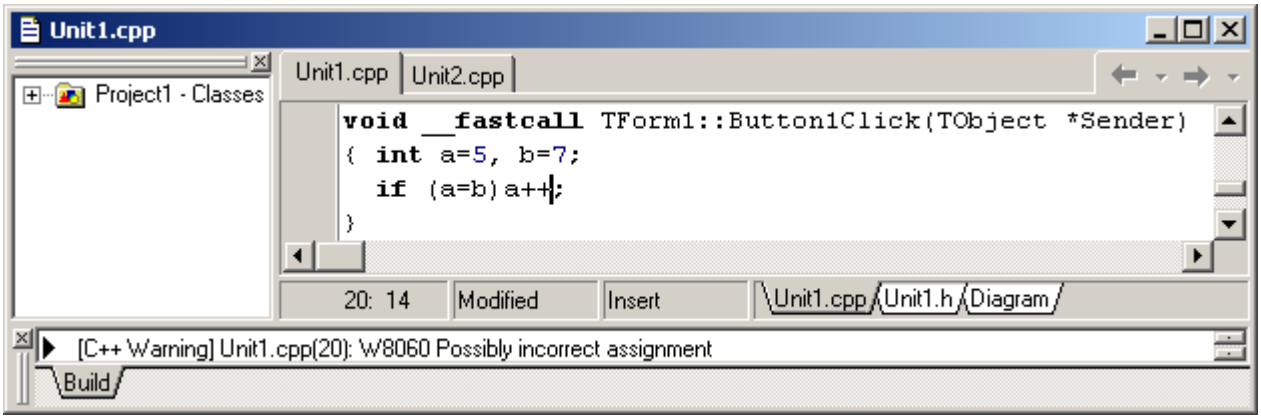


Рис. 15.27. Попередження W8060: замість операції порівняння використано оператор присвоювання

2) **[C++ Warning] Unit1.cpp(N) W8004 'x' is assigned a value that is never used**

Змінній `x` присвоєно значення, яке ніде не використовується. Це зауваження може сигналізувати про те, що значення виразу обчислено, але не виведено. Або про те, що значення обчислено, але далі замість нього використовується інша змінна, що призводить до логічної помилки і помилкового результату (чи взагалі нездобуття результату).

3) **[C++ Warning] Unit1.cpp(N) W8004 Possible use of x before definition**

Змінній `x` не присвоєно початкове значення.

15.3 Компонування

Якщо у програмі немає синтаксичних помилок, можна виконувати компонування. Для цього слід у меню **Project** обрати команду **Make** чи **Build**. Різниця поміж командами Make й Build полягає в такому. Команда Make забезпечує компонування файлів проекту, а команда Build – примусову перекомпіляцію, а потім компонування.

На етапі компонування також можуть виникати помилки. Найчастіше причиною помилок при компонуванні є недоступність файлів бібліотек чи інших раніше відкомпільованих модулів.

Наприклад:

[Linker Error] Unresolved external 'sum(int, int)' referenced from ... *(далі йде повне ім'я файла)*

Найчастіше така помилка виникає у багатофайловому проекті, коли в заголовному файлі прототип функції (ім'я чи щось інше) не збігається з іменем функції при виклику чи то заголовком функції в її реалізації. Наприклад, помилка на рис. 15.28 виникла внаслідок того, що в заголовному файлі `Unit2.h` ім'я функції `sum` написано з малої літери, а в `Unit2.cpp` – з великої (тобто не збігається регістр і імена вважаються за різні). Слід уважно простежити, щоб прототип функції, її визначення (реалізація) і виклик мали однакове ім'я.

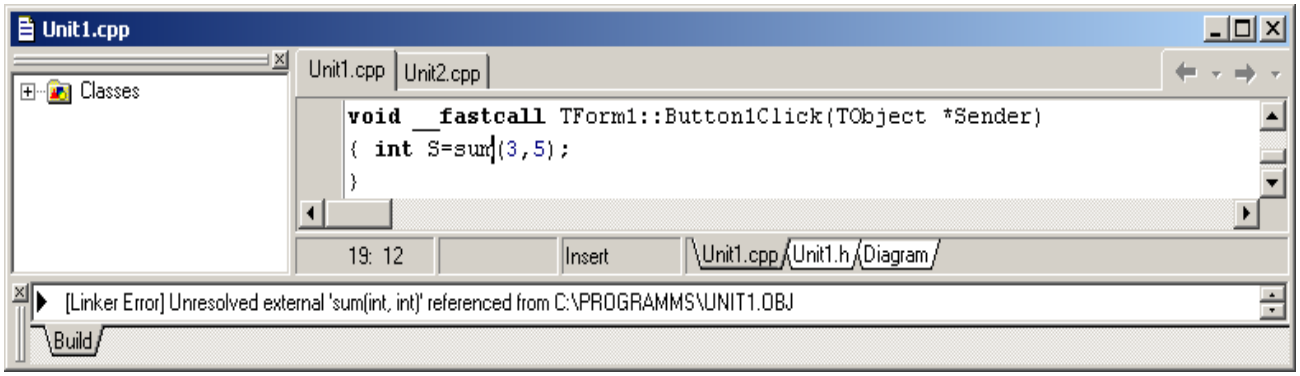


Рис. 15.28. Помилка на етапі компонування: імена функції в оголошенні (у `.h` файлі) та визначенні (у `.cpp` файлі) не збігаються

15.4 Помилки етапу виконання

Помилки другого рівня (помилки етапу виконання – run time errors) або *виняткові ситуації* (exceptions) зазвичай пов'язані з некоректними початковими значеннями, помилками обраного алгоритму чи неправильною програмною реалізацією алгоритму. Ці помилки виявляються в тому, що результат обчислень є неправильним або відбувається переповнення, ділення на нуль, зависання програми тощо. Тому вже налагоджену програму обов'язково слід протестувати, тобто зробити розрахунки за таких комбінацій початкових даних, для котрих заздалегідь є відомий результат. Якщо тестові розрахунки вказують на помилку, для її пошуку слід використовувати вбудовані способи налагодження середовища C++ Builder.

Найпоширеніші причини помилок етапу виконання:

1) неправильне визначення вихідних даних;
2) логічні помилки;
3) накопичування похибок результатів обчислень.

У простому випадку для локалізації місця помилки рекомендовано чинити в такий спосіб. У вікні редактора програмного коду встановити курсор у рядку перед підозрілою ділянкою і натиснути клавішу <F4> (виконання до курсора). Виконання програми буде призупинено на рядку, в якому перебуває курсор. Тепер можна переглянути значення всіх змінних, обчислених до цього рядка. Для цього слід помістити чи підвести до змінної курсор миші (на екрані поряд зі змінною висвітиться її значення) або натиснути <Ctrl>+<F7> і у діалоговому вікні, яке з'явиться, вказати змінну (за допомогою даного вікна також можна змінювати значення змінної у перебігу виконування програми). Натискаючи клавішу <F7> (покрокове виконування), можна виконувати програму по рядках, контролюючи змінювання тих чи інших змінних і правильність обчислень. Якщо курсор перебуває усередині циклу, після натискання <F4> розрахунок зупиняється після одного виконання операторів тіла циклу. Для продовження обчислень слід натиснути кнопку запуску програми на виконання.

Наведемо приклади типових помилок етапу виконання програми.

1) У програмі міститься код

```
void __fastcall TForm1::Button1Click
                                (TObject *Sender)
{ float x=5.7, y;
  Edit1->Text=x/y;
}
```

Результат програми є помилковим. Причина полягає в тому, що при діленні використовується змінна y, значення якої до ділення не було визначено, тобто воно є випадкове. Щоб уникнути цієї помилки, слід попередньо ініціалізувати змінну y.

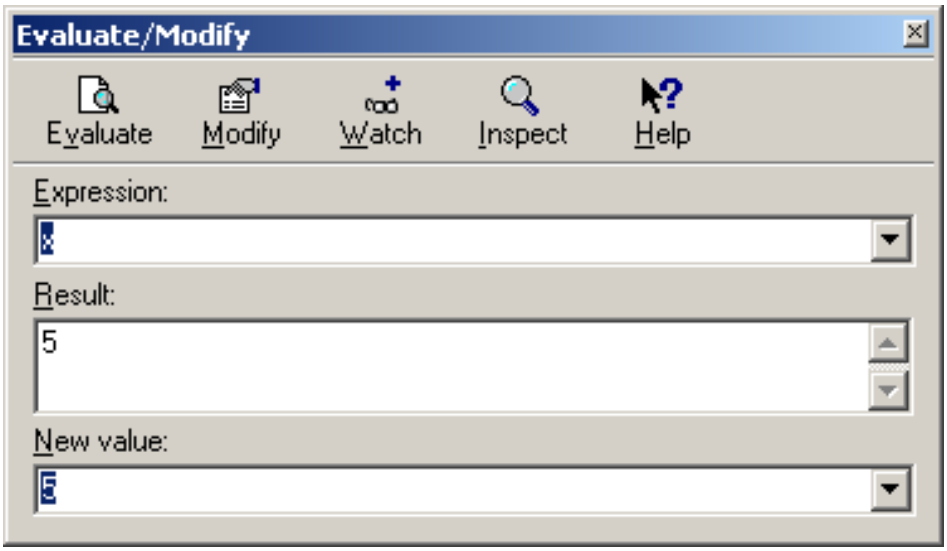
2) Віднайти й виправити помилки у програмі з кодом

```
void __fastcall TForm1::Button1Click
                                (TObject *Sender)
{ int x=5;
```

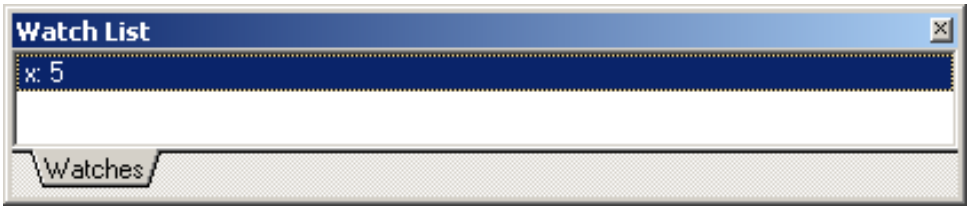
```
  while(x<15)
    x-=2;
  Edit1->Text=IntToStr(x);
}
```

При компіляції цього коду жодних повідомлень про помилки чи попередження не виникає. При виконуванні ця програма на певний час “зависає”, а потім виводить до `Edit1` число 2147483647, яке є некоректним результатом, оскільки сам програмний код є некоректний.

Установимо курсор усередину циклу (наприклад на рядок `x-=2;`) і натиснемо <F4>. На екрані з’явиться форма, клацнемо на `Button1`. Після цього повертаємось до вікна редагування коду. Якщо клацнути на змінну `x`, можна побачити її значення (`x=5`). Натиснемо <Ctrl>+<F7> і побачимо вікно



Натиснемо кнопку `Watch`, після чого з’явиться вікно



Якщо багаторазово натискати <F7>, можна спостерігати значення змінної `x`. Воно зменшується на 2 і ніколи не стане більше за 15. Коли змінна `x` стане менше за нижню межу типу `int`, вона набуде довільного значення (можливо, великого додатного). Якщо воно більше за 15, – програма зупиниться і до `Edit1` буде виведено це значення.

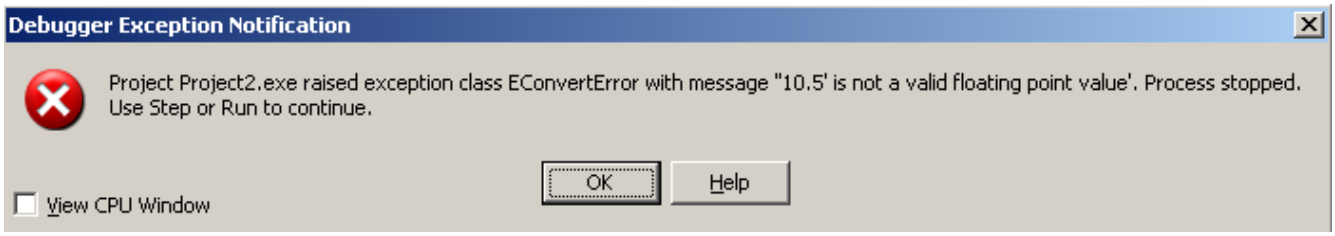
Приклад 15.1 Обчислити силу струму за формулою $I = U / R$, де U – напруга джерела (В), R – опір (Ом).

Текст програми:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float u, r, i;
  u=StrToFloat(Edit1->Text);
```

```
    r=StrToFloat(Edit2->Text);
    i=u/r;
    Edit3->Text=FloatToStr(i);
  }
```

Якщо при виконуванні програми до `Edit1` ввести число `10.5` замість `10,5`, тобто розділили цілу і дробову частини крапкою, після клацання на `Button1` на екрані з'явиться повідомлення



Причина виникнення цієї помилки полягає в такому. У тексті програми дробова частина завжди відокремлюється від цілої десятковою крапкою. При введенні даних до `Edit1` розділювач залежить від налагодження операційної системи. Для України зазвичай у Windows встановлено розділювачем є кома, Тому за використання десяткової крапки у вікні `Edit1` при виконуванні команди

```
u =StrToFloat(Edit1->Text);
```

виникне виняткова ситуація, оскільки аргументом функції `StrToFloat` не є зображення дробового числа.

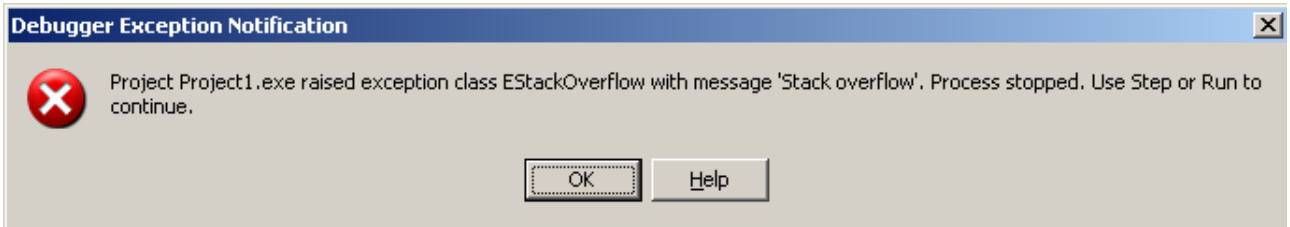
Після виникнення такої виняткової ситуації і клацання на кнопці `OK` у діалоговому вікні `Debugger Exception Notification` виконування програми можна перервати чи, незважаючи на помилку, продовжити. Щоб перервати виконування програми, слід у меню `Run` обрати команду `Program Reset`, а щоб продовжити – `Step Over`.

Найбільш поширені помилки на етапі виконування програми:

1) Переповнення (нескінченний цикл) `stack overflow`.

Наприклад:

```
void f(int n)
{ f(n+1);
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ f(5);
}
```



Причиною виникнення помилки є нескінченна рекурсія, унаслідок якої відбувається переповнення стека. Щоб уникнути цієї помилки, слід зазначати умову зупинки рекурсії, наприклад:

```
void f(int n)
{ if (n>=10) return;
  f(n+1);
}
```

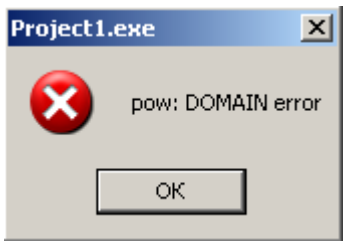
2) Надто велике/мале число чи то неприпустиме значення для використання у якості аргументу функції спричинюють помилки:

```
pow: domain error
log: SING error.
```

Приміром, команда програми

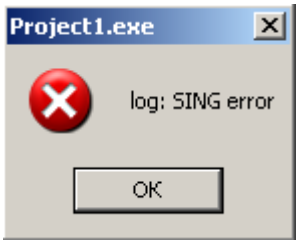
```
x=pow(-5, 0.3);
```

спричинить помилку



Наведемо ще один приклад програмного коду, який призводить до помилки на етапі виконування:

```
float x=0;
Edit1->Text=FloatToStr(log(x));
```



Причина помилки: логарифм нуля не існує, оскільки значення `x<=0` є неприпустиме для цієї функції.

3) Ділення на нуль: `Division by zero`.

Наприклад:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  float x=0;
  for(int i=0; i<5; i++)
    x+=1./i;
  Edit1->Text=FloatToStr(x);
}
```

При запусканні програми на виконання і клацанні на `Button1` з'являється повідомлення

Це пов'язано з тим, що на першій ітерації циклу за `i=0` виконується ділення `1./0`. Щоб уникнути цієї помилки, слід змінити початкове значення `i` (наприклад на `i=1`) або долучити оператор перевірки:

```
if(i!=0) x+=1./i;
```

Питання та завдання для самоконтролю

1) Які типи помилок можуть виникати у програмах на C++ Builder?

2) Що спричинює помилки при компіляції програми?

3) Які помилки є помилками другого рівня?

4) Які помилки можуть виникнути на етапі компонування?

5) Чим синтаксичні помилки відрізняються від логічних?

6) Чи запуститься на виконання програма, у якій знайдено синтаксичну помилку?

7) Чи можна виконати програму, у якій є логічна помилка?

8) Що може статися унаслідок логічної помилки?

9) Чи можна виконати програму, якщо при компіляції було видано попередження?

10) Про що свідчить повідомлення про помилку "`Statement missing ;`"? Як виправити таку помилку?

11) Про що може свідчити повідомлення про помилку "`Compound statement missing }`"? Як виправити таку помилку?

12) Унаслідок чого може з'явитися повідомлення про помилку "`Misplaced else`"?

13) Яких помилок припущено у такому фрагменті програми? Як їх виправити?

```
double x, y, z;
double y=sqrt(pow(3,x);
```

14) Про що може свідчити повідомлення про помилку "`log: SING error`" при виконуванні програми? Як її уникнути?

15) Яке повідомлення виведено, якщо у програмі із компонента `Edit` зчитується число з неправильним розділювачем (крапкою замість коми)?

16) Яким чином реагує компілятор, якщо у програмі оголошено змінну, але вона не використовується?

Бібліографічний список

1 **Архангельский А. Я.** Приемы программирования в C++ Builder 6 и 2006 / А. Я. Архангельский. – М. : ООО Бином-Пресс, 2006. – 992 с.

2 **Архангельский А. Я.** Программирование в C++ Builder 6 и 2006 / А. Я. Архангельский, М. А. Тагин. – М. : ООО Бином-Пресс, 2007. – 1184 с.

3 **Боровский А. Н.** C++ и Borland C++ Builder. Самоучитель / А. Н. Боровский. – 1-е изд. – СПб. : Питер, 2005. – 256 с.

4 **Бобровский С. И.** Самоучитель программирования на языке C++ в системе Borland C++ Builder 5.0 / С.И. Бобровский. – М. : ДЕСС КОМ, 2001. – 272 с.

5 **Бобровский С. И.** Технологии C++ Builder. Разработка приложений для бизнеса. Учебный курс / С.И. Бобровский. – СПб. : Питер, 2007. – 560 с.

6 **Глушаков С. В.** Программирование в среде C++ Builder 6 / С. В. Глушаков, В. Н. Зорянский, С. Н. Хоменко. – Харьков : Фолио, 2003. – 508 с.

7 **Глушаков С. В.** Практикум по C++ / С. В. Глушаков, С. В. Смирнов, А. В. Коваль. – Харьков : Фолио, 2006. – 526 с.

8 **Глушков С. В.** Язык программирования C++ : учебн. курс / С. В. Глушаков, С. В. Смирнов, А. В. Коваль.– Харьков : Фолио, 2001. – 500 с.

9 **Дейтел Х.М.** Как программировать на C++ / Х. М. Дейтел, П. Дж. Дейтел; пер. с англ. – 5-е изд. – М. : ООО Бином-Пресс, 2008. – 1456 с.

10 **Динман М. И.** C++. Освой на примерах / М. И. Динман. – СПб. : БХВ-Петербург, 2006. – 384 с.

11 **Джосьютис Н.** C++. Стандартная библиотека / Н. Джосьютис. – СПб. : Питер, 2004. – 730 с.

12 **Златопольский Д. М.** Сборник задач по программированию / Д. М. Златопольский. – 2-е изд. – СПб. : БХВ-Петербург, 2007. – 240 с.

13 **Ковалюк Т. В.** Основи програмування / Т. В. Ковалюк. – К. : BHV, 2005. – 384 с.

14 **Крупник А. Б.** Изучаем C++ / А. Б. Крупник. – СПб. : Питер, 2004. – 251 с.

15 **Крячков А. В.** Программирование на C и C++. Практикум : учебник / А. В. Крячков, И. В. Сухинина, В. К. Томшин. – 2-е изд. – М. : Горячая линия − Телеком, 2000. – 344 с.

16 **Культин Н. Б.** Самоучитель C++ Builder / Н. Б. Культин. – СПб. : БХВ-Петербург, 2004. – 320 с.

17 **Культин Н. Б.** C/C++ в задачах и примерах / Н. Б. Культин. – СПб. : БХВ-Петербург, 2007. – 336 с.

18 **Лафоре Р.** Объектно-ориентированное программирование в C++ / Р. Лафоре. – 4-е изд. – СПб.: Питер, 2003. – 928 с.

19 **Либерти Дж.** Освой самостоятельно C++ за 21 день / Джесс Либерти, Брэдли Джонс. – 5-е изд. – М. : Вильямс, 2009. – 784 с.

20 **Липпман С. Б.** Язык программирования C++. Вводный курс / С. Б. Липпман, Ж. Лажойе, Б. Э. Му. – СПб.: Невский диалект, 2001. – 1088 с.

21 **Лупал А. М.** Теория автоматов : учеб. пособие / А. М. Лупал. – СПб. : СПбГУАП, 2000. – 119 с.

22 **Мозговой М. В.** Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический поход / М. В. Мозговой. – СПб. : Наука и техника, 2006. – 320 с.

23 **Павловская Т. А.** C/C++. Программирование на языке высокого уровня : / Т. А. Павловская. – СПб. : Питер, 2002. – 464 с.

24 **Павловская Т. А.** C/C++. Структурное программирование: практикум : учебник / Т. А. Павловская, Ю.А. Щупак. – СПб. : Питер, 2003. – 240 с.

25 **Прата Ст.** Язык программирования C++. Лекции и упражнения: учебник / Стивен Прата; пер. с англ. – СПб. : ООО ДиаСофтЮП, 2005. – 1104 с.

26 **C++**. Основи програмування. Теорія та практика: підручник / [О. Г. Трофименко, Ю. В. Прокоп, І. Г. Швайко, Л. М. Буката та ін.] ; за ред. О. Г. Трофименко. – Одеса : Фенікс, 2010. – 544 с.

27 **Страуструп Б.** Дизайн и эволюция C++ / Бьерн Страуструп; пер. с англ. – М. : ДМК Пресс; СПб. : Питер, 2006. – 448 с.

28 **Страуструп Б.** Язык программирования C++. Специальное издание / Бьерн Страуструп; пер. с англ. – М. : ООО Бином-Пресс, 2007. – 1104 с.

29 **Шамис В. А.** C++ Builder. Техника визуального программирования / В. А. Шамис. – 3-е изд. – М. : Нолидж, 2001. – 688 с.

30 **Шилдт Г.** C++: базовый курс / Герберт Шилдт. – 3-е изд. – М. : Вильямс, 2009. – 624 с.

31 **Шилдт Г.** Справочник программиста по C/C++ / Герберт Шилдт. – М. : Вильямс, 2006. – 432 с.

Додаток А

Таблиці кодів ASCII

Таблиця А.1

Символи ASCII

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

Таблиця А.2

Керувальні символи ASCII

Код	Ім'я	Ctrl-код	Призначення
0	NUL (Null – порожньо, дані є відсутні)	^@	Використовується для передавання у разі відсутності даних. Вживається у багатьох мовах програмування як кінець рядка. В деяких операційних системах NUL – останній символ кожного текстового файла
1	SOH (Start of Heading – початок заголовка)	^A	Використовується для зазначення початку заголовка, який може містити інформацію про маршрутизацію чи адресу
2	STX (Start of Text – початок тексту)	^B	Вказує на початок тексту і водночас на кінець заголовка
3	ETX (End of Text – кінець тексту)	^C	Використовується при завершенні тексту, який розпочинався символом STX
4	EOT (End of Transmission – кінець передачі)	^D	У системі UNIX <Ctrl>+<D> означає кінець файла при введенні даних з клавіатури
5	ENQ (Enquiry – запит)	^E	Запит ідентифікаційних даних від віддаленої станції

Продовження табл. А.2

Код	Ім'я	Ctrl-код	Призначення
6	ACK (Acknowledge – потвердження)	^F	Приймальний пристрій передає цей символ відправникові як потвердження успішного прийому даних
7	BEL (Bell – дзвоник, звуковий сигнал)	^G	Використовується для керування пристроями сигналізації. У мовах програмування С та С++ позначається `\a`
8	BS (BackSpace – повертання на один символ)	^H	Повертання на одну позицію назад, відбувається вилучання попереднього символу
9	HT (Horizontal Tabulation – горизонтальне табулювання), чи TAB	^I	Відбувається переміщення курсора до наступної позиції табуляції. У багатьох мовах програмування позначається `\t`
10	LF (Line Feed – переведення рядка)	^J	Відбувається переміщення курсора до початку наступного рядка (на один рядок вниз). У кінці кожного рядка текстового файлу ставиться чи то цей символ, чи CR, чи вони обидва разом (CR, а потім LF), залежно від операційної системи. У багатьох мовах програмування позначається `\n` і при виведенні тексту призводить до переведення рядка
11	VT (Vertical Tabulation – вертикальне табулювання)	^K	Відбувається переміщення курсора до наступної групи рядків
12	FF (Form Feed – переведення сторінки)	^L	Відбувається переміщення курсора до початку наступної сторінки, форми чи екрана
13	CR (Carriage Return – переведення каретки)	^M	Відбувається переміщення курсора до початку (крайньої лівої) позиції поточного рядка. У багатьох мовах програмування позначається `\r` і використовується для повертання на початок рядка без переведення рядка. У деяких операційних системах цей символ позначається <Ctrl>+<M> і ставиться наприкінці кожного рядка текстового файла перед LF
14	SO (Shift Out)	^N	Указує, що всі наступні кодові комбінації мають інтерпретуватися згідно із зовнішнім набором символів до приходу символу SI
15	SI (Shift In)	^O	Указує, що наступні кодові комбінації мають інтерпретуватися згідно зі стандартним набором символів

Продовження табл. А.2

Код	Ім'я	Ctrl-код	Призначення
16	DLE (Data Link Escape – перемикання)	^P	Використовується для змінювання значення наступних символів для додаткового контролю чи для передавання довільної комбінації бітів
17	DC1/XON	^Q	(Device Controls – контроль пристрою) символи для керування допоміжними пристроями (спеціальними функціями)
18	DC2	^R	
19	DC3/XOFF	^S	
20	DC4	^T	
21	NAK (Negative Acknowledgement – непотвердження)	^U	Приймальний пристрій передає цей символ відправнику у разі відмови прийому даних
22	SYN (Synchronous/Idle – синхронізація)	^V	Використовується у синхронізованих системах передавання. У моменти відсутності передавання даних система безперервно надсилає символи SYN для забезпечення синхронізації
23	ETB (End of Transmission Block – кінець блока передачі)	^W	Указує на кінець блока даних для комунікаційних цілей. Використовується для розбиття на окремі блоки великих обсягів даних
24	CAN (Cancel – відміна)	^X	Зазначає, що даними, які передували цьому символові у повідомленні чи блоці, слід знехтувати (зазвичай у разі виявлення помилки)
25	EM (End of Medium – кінець носія)	^Y	Указує на фізичний кінець носія інформації
26	SUB (Substitute – замінник)	^Z	Використовується для заміни помилкового чи неприпустимого символу. <CTRL>+<Z> використовується як кінець файла при введенні даних з клавіатури у системах DOS та Windows
27	ESC (Escape – розширення)	^[	Використовується для розширення коду, зазначаючи те, що наступний символ має альтернативне значення
28	FS (File Separator – розділювач файлів)	^\	Набір керувальних символів, що використовувались у минулому як розділювачі меж порцій інформації
29	GS (Group Separator – розділювач груп)	^]	
30	RS (Record Separtator – розділювач записів)	^^	
31	US (Unit Separator – розділювач юнітів)	^_	

Закінчення табл. А.2

Код	Ім'я	Ctrl-код	Призначення
32	(sp) (Space – пробіл)		Недрукований символ для розділювання слів чи переміщування курсора на екрані вперед на одну позицію
127	DEL (Delete – вилучання)	^?	Використовується для вилучання (стирання) символу з поточної позиції курсора

Таблиця А.3

Кодування другої половини ASCII

Кодування ср866

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	А 128	Б 129	В 130	Г 131	Д 132	Е 133	Ж 134	З 135	И 136	Й 137	К 138	Л 139	М 140	Н 141	О 142	П 143
9	Р 144	С 145	Т 146	У 147	Ф 148	Х 149	Ц 150	Ч 151	Ш 152	Щ 153	Ъ 154	Ы 155	Ь 156	Э 157	Ю 158	Я 159
A	а 160	б 161	в 162	г 163	д 164	е 165	ж 166	з 167	и 168	й 169	к 170	л 171	м 172	н 173	о 174	п 175
B	▒ 176	▒ 177	▓ 178	│ 179	┤ 180	╡ 181	╢ 182	╖ 183	╕ 184	╣ 185	║ 186	╗ 187	╝ 188	╜ 189	╛ 190	┐ 191
C	└ 192	┴ 193	┬ 194	├ 195	─ 196	┼ 197	╞ 198	╟ 199	╚ 200	╔ 201	╩ 202	╦ 203	╠ 204	═ 205	╬ 206	╧ 207
D	╨ 208	╤ 209	╥ 210	╙ 211	╘ 212	╒ 213	╓ 214	╫ 215	216	┘ 217	┌ 218	▉ 219	▄ 220	▌ 221	▐ 222	▀ 223
E	р 224	с 225	т 226	у 227	ф 228	х 229	ц 230	ч 231	ш 232	щ 233	ъ 234	ы 235	ь 236	э 237	ю 238	я 239
F	Ё 240	ё 241	≥ 242	≤ 243	⌠ 244	⌡ 245	÷ 246	≈ 247	° 248	∙ 249	· 250	√ 251	ⁿ 252	² 253	■ 254	□ 255

Кодування ср1251 (Windows-1251)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	Ђ 128	Ѓ 129	‚ 130	ѓ 131	„ 132	… 133	† 134	‡ 135	€ 136	‰ 137	Љ 138	‹ 139	Њ 140	Ќ 141	Ћ 142	Џ 143
9	ђ 144	‘ 145	’ 146	" 147	" 148	• 149	– 150	— 151	152	™ 153	љ 154	› 155	њ 156	ќ 157	ћ 158	џ 159
A	160	Ў 161	ў 162	Ј 163	¤ 164	Ґ 165	¦ 166	§ 167	Ё 168	© 169	Є 170	« 171	¬ 172	- 173	® 174	Ї 175
B	° 176	± 177	І 178	і 179	ґ 180	µ 181	¶ 182	· 183	ё 184	№ 185	є 186	» 187	ј 188	S 189	s 190	ї 191
C	А 192	Б 193	В 194	Г 195	Д 196	Е 197	Ж 198	З 199	И 200	Й 201	К 202	Л 203	М 204	Н 205	О 206	П 207
D	Р 208	С 209	Т 210	У 211	Ф 212	Х 213	Ц 214	Ч 215	Ш 216	Щ 217	Ъ 218	Ы 219	Ь 220	Э 221	Ю 222	Я 223
E	а 224	б 225	в 226	г 227	д 228	е 229	ж 230	з 231	и 232	й 233	к 234	л 235	м 236	н 237	о 238	п 239
F	р 240	с 241	т 242	у 243	ф 244	х 245	ц 246	ч 247	ш 248	щ 249	ъ 250	ы 251	ь 252	э 253	ю 254	я 255

Додаток Б

Операції мови С++

Таблиця Б.1

Операція	Опис	Напрямок
Унарні операції		
`::`	доступ до області видимості	⇒
`.`	вибір елемента через об'єкт	
`->`	вибір елемента через вказівник	
`[ ]`	індекс масиву	
`( )`	дужки, виклик функції	
`<`*тип*`>()`	конструювання	⇐
`++`	постфіксний інкремент	
`--`	постфіксний декремент	
`typeid`	ідентифікація типу часу виконання	
`dynamic_cast`	перетворення типу з перевіркою на етапі виконування	
`static_cast`	перетворення типу з перевіркою на етапі компіляції	
`reinterpret_cast`	перетворення типу без перевірки	
`const_cast`	константне перетворення типу	
`sizeof`	розмір об'єкта чи типу в байтах	
`--`	префіксний декремент	
`++`	префіксний інкремент	
`~`	порозрядне НЕ (заперечення, інверсія)	
`!`	логічне НЕ (заперечення, інверсія)	
`-`	унарний мінус	
`+`	унарний плюс	
`&`	адреса	
`*`	розадресація	
`new`	динамічне виділення пам'яті	
`delete`	динамічне звільнення пам'яті	
`(<`*тип*`>)`	зведення типів	
`.*`	вибір на елемент через об'єкт	
`->*`	вибір на елемент через вказівник	
Бінарні й тернарна операції		
`*`	множення	⇒
`/`	ділення	
`%`	остача від ділення	
`+`	додавання	⇒
`-`	віднімання	
`<<`	порозрядний зсув ліворуч	⇒
`>>`	порозрядний зсув праворуч	
`<`	менше	⇒
`<=`	менше чи дорівнює	
`>`	більше	
`>=`	більше чи дорівнює	

Закінчення табл. А.2

Операція	**Опис**	**Напрямок**
`==` `!=`	дорівнює не дорівнює	⇒
`&`	порозрядна кон'юнкція (І)	⇒
`^`	порозрядне виключне АБО	⇒
`|`	порозрядна диз'юнкція (АБО)	⇒
`&&`	логічне І	⇒
`||`	логічне АБО	⇒
`? :`	умовна операція (тернарна)	⇐

`=`
`*=`
`/=`
`%=`
`+=`
`-=`
`<<=`
`>>=`
`&=`
`|=`
`^=`
`throw`
	присвоювання
	множення з присвоюванням
	ділення з присвоюванням
	остача від ділення з присвоюванням
	додавання з присвоюванням
	віднімання з присвоюванням
	зсув ліворуч з присвоюванням
	зсув праворуч з присвоюванням
	порозрядне І з присвоюванням
	порозрядне АБО з присвоюванням
	порозрядне виключне АБО з присвоюванням
	генерування виняткової ситуації
		⇐

`,`	кома, послідовне обчислювання	⇒
Операції наведено у порядку зменшення пріоритету. Операції з різними пріоритетами розділені рискою		

Додаток В

Функції стандартної бібліотеки C++. Вміст заголовних файлів

Таблиця В.1

Заголовний файл <ctype.h> (<cctype>) – функції класифікації і перетворення символів

Функція	Опис
isalnum	Перевіряє, є символ літерою чи цифрою
isalpha	Перевіряє, чи є символ літерою
iscntrl	Перевіряє, чи є символ керувальним
isdigit	Перевіряє, чи є символ цифрою
isgraph	Перевіряє, чи є символ видимим
islower	Перевіряє, чи є символ літерою нижнього регістру
isprint	Перевіряє, чи є символ друкованим
ispunct	Перевіряє, чи є символ пунктуаційним
isspace	Перевіряє, чи є символ розділювальним
isupper	Перевіряє, чи є символ літерою верхнього регістру
iswalnum	Перевіряє, є символ літерою чи цифрою
iswalpha	Перевіряє, чи є символ літерою
iswcntrl	Перевіряє, чи є символ керувальним
iswctype	Перевіряє багатобайтовий символ
iswdigit	Перевіряє, чи є символ цифрою
iswgraph	Перевіряє, чи є символ видимим
iswlower	Перевіряє, чи є символ літерою нижнього регістру
iswprint	Перевіряє, чи є символ друкованим
iswpunct	Перевіряє, чи є символ пунктуаційним
iswspace	Перевіряє, чи є символ розділювальним
iswupper	Перевіряє, чи є символ літерою верхнього регістру
iswxdigit	Перевіряє, чи є символ шістнадцядковою цифрою
isxdigit	Перевіряє, чи є символ шістнадцядковою цифрою
tolower	Повертає символ у нижньому регістрі
toupper	Повертає символ у верхньому регістрі
towlower	Повертає символ у нижньому регістрі
towupper	Повертає символ у верхньому регістрі

Заголовний файл <string.h> (<cstring>) – функції локалізації

localeco	Повертає опис налагоджень локального середовища
setlocal	Встановлює нову локалізацію

Заголовний файл <math.h> (<cmath>) – математичні функції

Функція	Опис
acos	Повертає арккосинус аргументу
asin	Повертає арксинус аргументу
atan	Повертає арктангенс аргументу
atan2	Повертає арктангенс відношення аргументів
ceil	Округлює його до більшого цілого
cos	Обчислює косинус
cosh	Обчислює гіперболічний косинус
exp	Обчислює експоненту
fabs	Повертає модуль числа
floor	Округлює його до меншого цілого
fmod	Повертає остачу від ділення x на y
frexp	Виокремлює з числа мантису та експоненціальну частину
ldexp	Перетворює мантису та показник степеня на число
log	Обчислює натуральний логарифм
log10	Обчислює десятковий логарифм
modf	Розбиває число на цілу й дійсну частини
pow	Підносить число до степеня
sin	Обчислює синус
sinh	Обчислює гіперболічний синус
sqrt	Обчислює квадратний корінь
tan	Повертає тангенс аргументу
tanh	Повертає гіперболічний тангенс аргументу

Заголовний файл <signal.h> (<csignal>) – функції опрацювання сигналів

raise	Переривает виконування програми
signal	Реєструє опрацювання сигналів

Заголовний файл <stdio.h> (<cstdio>) – функції введення/виведення у стилі C

clearerr	Очищує прапорці помилок при роботі з потоком
fclose	Закриває потік введення-виведення
feof	Перевіряє досягання кінця файла
terror	Повертає код помилки при роботі з потоком
fflush	Записує дані з буфера
fgetc	Читає з потоку символ
fgetpos	Повертає поточну позицію у файлі
fgets	Читає з потоку в рядок n символів
fgetwc	Читає з потоку символ
fgetws	Читає з потоку в рядок n символів
fopen	Відкриває потік введення-виведення
fprintf	Записує дані до потоку

Заголовний файл <stdio.h> (<cstdio>) – функції введення/виведення у стилі C

Функція	Опис
fputc	Записує символ до потоку
fputs	Записує рядок символів до потоку
fputwc	Записує символ до потоку
fputws	Записує рядок символів до потоку
tread	Зчитує дані з потоку введення
freopen	Відкриває потік введення-виведення
fscanf	Виводить з файла форматовані дані
fseek	Переміщує позицію у файлі відносно поточної
fsetpos	Переміщує поточну позицію у файлі відносно його початку
ftell	Повертає поточну позицію у файлі
fwprintf	Записує дані до потоку
fwrite	Записує дані із заданого буфера до потоку
fwscanf	Зчитує з потоку дані й записує їх по заданих адресах
getc	Зчитує символ з потоку
getchar	Зчитує символ зі стандартного введення
gets	Зчитує символи з клавіатури до появи символу нового рядка
getwc	Зчитує символ з потоку
getwchar	Повертає черговий символ з клавіатури
perror	Виводить рядок вигляду «s: error повідомлення»
printf	Виводить рядок параметрів у певному форматі
putc	Записує символ до потоку
putchar	Виводить символ на стандартний пристрій виведення
puts	Виводить рядок на стандартний пристрій виведення
putwc	Записує символ до потоку
putwchar	Виводить символ на стандартний пристрій виведення
remove	Вилучає файл
rename	Перейменовує файл
rewind	Очищує прапорці помилок при роботі з потоком і переходить до початку файла
scanf	Вводить рядок параметрів у певному форматі
setbuf	Встановлює буферизацію потоку введення-виведення
setvbuf	Перетворює рядки на основі поточної локалізації
sprintf	Виводить рядок параметрів у певному форматі
sscanf	Вводить дані з рядка
swprintf	Виводить рядок параметрів у певному форматі
swscanf	Вводить дані з рядка
tmpfile	Відкриває потік двійкового введення-виведення до тимчасового файла
tmpnam	Створює унікальне ім’я файла
ungetc	Повертає символ до потоку

Заголовний файл <stdio.h> (<cstdio>) – функції введення/виведення у стилі С

Функція	Опис
ungetwc	Повертає символ до потоку
vfprintf	Надсилає відформатоване виведення до потоку
vfwprintf	Надсилає відформатоване виведення до потоку
vprintf	Надсилає відформатоване виведення до стандартного потоку виведення
vsprintf	Виводить рядок параметрів у певному форматі
vswprintf	Виводить рядок параметрів у певному форматі
vwprintf	Надсилає відформатоване виведення до стандартного потоку виведення
wprintf	Виводить рядок параметрів у певному форматі
wscanf	Вводить рядок параметрів у певному форматі

Заголовний файл <stdlib.h> (<cstdlib>) – різні функції у стилі С

abort	Перериває виконування програми
abs	Повертає модуль числа
atexit	Реєструє функцію, яка викликається по завершенні роботи програми
atof	Перетворює рядок на дійсне число
atoi	Перетворює рядок на ціле число
atol	Перетворює рядок на довге ціле число
bsearch	Відшукує елемент у відсортованому масиві
calloc	Виділяє блок пам’яті
div	Ділення з остачею
exit	Перериває виконування програми
free	Звільнює блок пам’яті
getenv	Повертає значення змінної оточення
labs	Повертає модуль числа
ldiv	Ділення з остачею
malloc	Виділяє блок пам’яті
mblen	Визначає розмір багатобайтових символів
mbstowcs	Перетворює рядок багатобайтових символів на масив з wchar_t
mbtowc	Перетворює багатобайтовий символ на wchar_t
qsort	Сортує заданий масив
rand	Генерує випадкові числа
realloc	Змінює розміри раніш виділеної пам’яті
srand	Встановлює початкове псевдовипадкове число
strtod	Перетворює рядок на число
strtol	Перетворює рядок на число з урахуванням системи числення
strtoul	Перетворює рядок на число з урахуванням системи числення
system	Передає рядок командному процесорові ОС
wcstod	Перетворює рядок на число

Заголовний файл <stdlib.h> (<cstdlib>) – різні функції у стилі C

Функція	Опис
wcstol	Перетворює рядок в число з урахуванням системи числення
wcstombs	Перетворює wchar_t на рядок багатобайтових символів
wcstoul	Перетворює рядок на число з урахуванням системи числення
wctomb	Перетворює wchar_t на багатобайтовий символ

Заголовний файл <string.h> (<cstring>) – роботи з рядками функції у стилі C

memchr	Відшукує перше входження символу до блока пам'яті
memcmp	Порівнює блоки пам'яті
memcpy	Копіює блок пам'яті
mewmove	Переносить блок пам'яті
memset	Заповнює блок пам'яті символом
strcat	Сполучає рядки
strchr	Відшукує символ у рядку
strcmp	Порівнює рядки
strcoll	Порівнює рядки з урахуванням встановленої локалізації
strcpy	Копіює один рядок до іншого
strcspn	Відшукує один із символів одного рядка в іншому
strerror	Повертає вказівник на рядок з описом помилки
strlen	Повертає довжину рядка
strncat	Сполучає один рядок з n символами іншого
strncmp	Порівнює один рядок з n символами іншого
strncpy	Копіює перші n символів одного рядка до іншого
strpbrk	Відшукує один із символів одного рядка в іншому
strrchr	Відшукує символ в рядку
strspn	Відшукує символ одного рядка, який є відсутній в іншому
strstr	Відшукує підрядок у рядку
strtok	Виокремлює з рядка лексеми
strxfrm	Перетворює рядки на основі поточної локалізації
wcscat	Сполучає рядки
wcschr	Відшукує символ у рядку
wcscmp	Порівнює рядки
wcscoll	Порівнює рядки з урахуванням встановленої локалізації
wcscpy	Копіює один рядок до іншого
wcscspn	Відшукує один із символів одного рядка в іншому
wcslen	Повертає довжину рядка
wcsncat	Сполучає один рядок з n символами іншого
wcsncmp	Порівнює один рядок з n символами іншого
wcsncpy	Копіює перші n символів одного рядка до іншого
wcspbrk	Відшукує один із символів одного рядка в іншому
wcsrchr	Відшукує символ у рядку

Заголовний файл <string.h> (<cstring>) – роботи з рядками функції у стилі С

Функція	Опис
wcsspn	Відшукує символ одного рядка, відсутній в іншому
wcsstr	Відшукує підрядок у рядку
wcstok	Виокремлює з рядка лексеми
wcstrxfrm	Перетворює рядки на основі поточної локалізації
wmemcpy	Копіює блок пам'яті
wmemmove	Переносить блок пам'яті
wmemset	Заповнює блок пам'яті символом

Заголовний файл <time.h> (<ctime>) – функції для роботи з датою і часом у стилі С

asctime	Перетворює дату-час на рядок
clock	Повертає час виконування програми
ctime	Перетворює час на рядок
difftime	Повертає різницю в часі
gmtime	Ініціалізує структуру tm на основі time_t
localtime	Ініціалізує структуру tm на основі time_t
mktime	Заповнює поля дня тижня і дня року
strftime	Перетворює час у форматі fmt на формат tm
time	Повертає поточні дату/час у вигляді time_t
wcsftime	Повертає час у форматі fmt до формату tm

Заголовний файл <wchar.h> (<cwchar>) – функції для роботи з багатобайтовими символами у стилі С

btowc	Перетворює символ і довге ціле число
fwide	Визначає вигляд потоку
wmemchr	Відшукує перше входження символу до блока пам'яті
wmemcmp	Порівнює блоки пам'яті

Предметний покажчик

E

F

G

T

P

C

T

У

Ф

Ц-Ч

Навчальне видання

ТРОФИМЕНКО Олена Григорівна
ПРОКОП Юлія Віталіївна
ШВАЙКО Ігор Григорович
БУКАТА Людмила Миколаївна
ШАПОВАЛЕНКО Валентина Андріївна
ЛЕОНОВ Юрій Григорович
ЯСИНСЬКИЙ Василь Володимирович

C++

Теорія та практика

Навчальний посібник
за редакцією **О. Г. Трофименко**